ARM Architecture Reference Manual

Debug supplement

ARM®

Copyright © 2006 ARM Limited. All rights reserved.
ARM DDI 0379A

ARM Architecture Reference Manual

Copyright © 2006 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
8 February 2006	A	Non-Confidential	First release.

Proprietary Notice

ARM, the ARM Powered logo, Jazelle, RealView, and Thumb are registered trademarks of ARM Limited.

The ARM logo, AMBA, and CoreSight are trademarks of ARM Limited.

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

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

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

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

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

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

Copyright © 2006 ARM Limited

110 Fulbourn Road Cambridge, England CB1 9NJ

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

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

Copyright © 2006 ARM Limited. All rights reserved.

Contents
ARM Architecture Reference Manual Debug supplement

Preface

About this manual ... x
Conventions ... xii
Further reading ... xiii
Feedback ... xiv

Chapter 1 Introduction

1.1 Overview ... 1-2
1.2 Debug ... 1-3
1.3 Performance counters ... 1-5
1.4 Trace ... 1-6
1.5 Register interfaces ... 1-7

Chapter 2 Debug Events

2.1 Overview ... 2-2
2.2 Invasive debug authentication ... 2-4
2.3 Software Debug events ... 2-6
2.4 Halting Debug events ... 2-18
2.5 Generation of Debug events ... 2-20
2.6 Debug event priority and order ... 2-23

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 3 Debug Exceptions

3.1 Overview 3-2
3.2 Effects of Debug Exceptions on CP15 registers and the WFAR 3-4

Chapter 4 Debug State

4.1 Overview 4-2
4.2 Entering Debug state 4-3
4.3 Behavior of the PC and CPSR in Debug state 4-7
4.4 Executing instructions in Debug state 4-9
4.5 Privilege in Debug state 4-13
4.6 Behavior of non-invasive debug in Debug state 4-18
4.7 Exceptions in Debug state 4-19
4.8 Leaving Debug state 4-21

Chapter 5 Debug Register Interfaces

5.1 About the Debug Register Interface 5-2
5.2 Reset and Power-down support 5-6
5.3 Debug Register Map 5-13
5.4 Synchronization of debug register updates 5-18
5.5 Access permissions 5-20
5.6 Coprocessor interface 5-24
5.7 The Memory-mapped and recommended external debug interfaces 5-34

Chapter 6 Recommended External Debug Interface

6.1 System integration signals 6-2
6.2 Recommended debug slave port 6-10

Chapter 7 Debug Requirements on Memory Systems

7.1 About debug requirements on memory systems 7-2
7.2 Recommended access to specific CP15 registers 7-3
7.3 Debug state Cache/MMU Control Registers 7-4

Chapter 8 Non-invasive debug

8.1 About non-invasive debug 8-2
8.2 Program counter sampling register 8-3
8.3 Non-invasive debug authentication 8-4

Chapter 9 Core-based Performance Counters

9.1 About core-based performance counters 9-2
9.2 Status in the ARM architecture 9-4
9.3 Accuracy of performance counters 9-5
9.4 Behavior on overflow 9-6
9.5 Interaction with Security Extensions 9-7
9.6 Interaction with trace 9-8
9.7 Interaction with power saving operations 9-9

 Copyright © 2006 ARM Limited. All rights reserved.

9.8 Register map ... 9-10
9.9 Access permissions ... 9-12
9.10 Event numbers ... 9-13

Chapter 10 **Debug Register Reference**
10.1 Identification registers ... 10-3
10.2 Control and status registers ... 10-8
10.3 Instruction and data transfer registers ... 10-32
10.4 Breakpoint and watchpoint registers ... 10-39
10.5 Operating-system save and restore registers ... 10-58
10.6 Memory system control registers ... 10-61
10.7 Management registers ... 10-69
10.8 Core-based performance counters registers ... 10-80

Glossary

 Copyright © 2006 ARM Limited. All rights reserved.

 Copyright © 2006 ARM Limited. All rights reserved.

Preface

This preface introduces the *ARM Architecture Reference Manual, Debug supplement*. It contains the following sections:

- *About this manual* on page x
- *Conventions* on page xii
- *Further reading* on page xiii
- *Feedback* on page xiv.

 Copyright © 2006 ARM Limited. All rights reserved.

About this manual

The purpose of this manual is to describe the ARM Debug architecture. It is a supplement to the *ARM Architecture Reference Manual* (ARM DDI 0100, the ARM ARM), and is intended to be used with it.

It is assumed that the reader is familiar with:

- the ARM programmer's model, described in Part A Chapter 2 of the *ARM Architecture Reference Manual*
- the memory system architecture support, described in Part B of the *ARM Architecture Reference Manual*.

This manual is described in the following sections:

- *Intended audience*
- *Using this manual*.

Intended audience

This book is written for all developers designing:

- ARM processors
- hardware using ARM processors
- software for systems using ARM processors.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the purpose of the ARM Debug Architecture, and an overview of how this purpose is achieved.

Chapter 2 *Debug Events*

Read this chapter for information about what Debug events are, and how a processor responds to them.

Chapter 3 *Debug Exceptions*

Read this chapter for information about Debug Exceptions.

Chapter 4 *Debug State*

Read this chapter for information about Debug State.

Chapter 5 *Debug Register Interfaces*

Read this chapter for information about the Debug Register Interfaces.

Chapter 6 *Recommended External Debug Interface*

Read this chapter for details of the recommended external debug interface.

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 7 ***Debug Requirements on Memory Systems***

Read this chapter for details of the requirements placed on memory systems by debug.

Chapter 8 ***Non-invasive debug***

Read this chapter for information about non-invasive debug.

Chapter 9 ***Core-based Performance Counters***

Read this chapter for information about core-based performance counters.

Chapter 10 ***Debug Register Reference***

Read this chapter for reference information about the debug registers.

Copyright © 2006 ARM Limited. All rights reserved.

Conventions

Conventions that this manual can use are described in:

- *Typographic*
- *Signals*.

Typographic

`typewriter`	Is used for assembler syntax descriptions, pseudo-code descriptions of instructions, and source code examples. The `typewriter` font is also used in the main text for instruction mnemonics and for hexadecimal numbers.
italic	Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.
bold	Is used for emphasis in descriptive lists and elsewhere, where appropriate.
SMALL CAPITALS	Are used for a few terms that have specific technical meanings. Their meanings can be found in the *Glossary*.
< **and** >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They appear in normal font in running text. For example: • `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` • The Opcode_2 value selects which register is accessed.

Signals

The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.
Lower-case n	Denotes an active-LOW signal.
Prefix DBG	Denotes debug signals.

 Copyright © 2006 ARM Limited. All rights reserved.

Further reading

This section lists publications that provide additional information on the ARM family of processors.

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

ARM publications

This architecture specification is a supplement to, and must be read in conjunction with, the *ARM Architecture Reference Manual*, ARM DDI 0100.

The following documents contain additional information that is relevant to the information given in this specification, or complement the information given here:

- *ARM Architecture Reference Manual Security Extensions Supplement*, ARM DDI 0309
- *ARM Debug Interface v5 Architecture Specification*, ARM IHI 0031
- *CoreSight Architecture Specification*, ARM IHI 0029
- *Embedded Trace Macrocell Architecture Specification*, ARM IHI 0014.

Other publications

This section lists relevant documents published by third parties:

- *EEE 1149.1-1990 IEEE Standard Test Access Port and Boundary Scan Architecture* (JTAG).
- *JEP106M, Standard Manufacture's Identification Code*, JEDEC Solid State Technology Association.

 Copyright © 2006 ARM Limited. All rights reserved.

Feedback

ARM Limited welcomes feedback on its documentation.

Feedback on this manual

If you have any comments on this manual, send email to `errata@arm.com` giving:

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

ARM Limited also welcomes general suggestions for additions and improvements.

Copyright © 2006 ARM Limited. All rights reserved.

Chapter 1
Introduction

This chapter introduces the ARM Debug Architecture. It contains the following sections:

- *Overview* on page 1-2
- *Debug* on page 1-3
- *Performance counters* on page 1-5
- *Trace* on page 1-6
- *Register interfaces* on page 1-7.

 Copyright © 2006 ARM Limited. All rights reserved.

1.1 Overview

ARMv6 was the first version of the ARM architecture to include debug provisions. The introduction of the ARM Architecture Security Extensions extended the ARMv6 Debug Architecture:

- ARMv6 systems without Security Extensions implement v6 debug
- ARMv6 systems with Security Extensions implement v6.1 debug.

ARMv7 introduces additional extensions to support developments in the debug environment.

The main change in the debug environment is the emergence of new forms of external debug interface. Although ARMv6 did not require a particular debug interface, it was designed with the JTAG scan-chain model in mind. JTAG remains an important and widely used interface. However, systems such as the ARM CoreSight architecture require changes in the debug interface. For more information about the CoreSight architecture see the *CoreSight Architecture Specification*. Some of the aims of the CoreSight architecture, such as a more system-centric view of debug, and improved debug of powered-down systems, are addressed in the ARMv7 core debug architecture.

Another important aspect of v6 Debug is the optional trace extension. This is implemented by an Embedded Trace Macrocell (ETM) compliant to ETMv3. This optional extension is retained in ARMv7, and the first version of the ETM architecture for ARMv7 implementations is ETMv3.3.

ARMv7 debug also introduces an architecture extension to provide core-based performance counters.

Table 1-1 shows the main components of ARMv7 debug, and where they are described.

Table 1-1 ARMv7 Debug sub-architectures

Component	Status	Reference
Debug	Required	Chapter 2 *Debug Events*
		Chapter 3 *Debug Exceptions*
		Chapter 4 *Debug State*
		Chapter 5 *Debug Register Interfaces*
Performance counters	Optional	Chapter 9 *Core-based Performance Counters*
Trace	Optional	*Embedded Trace Macrocell Architecture Specification*

ARM processors implement two types of debug support:

Invasive debug All debug features that allow modification of the processor state.

Non-invasive debug All debug features that allow data and program flow observation, especially trace support.

Copyright © 2006 ARM Limited. All rights reserved.

1.2 Debug

The debug component of the ARM Debug Architecture is primarily intended for run-control debugging. Versions 6 and 7 of the architecture (ARMv6 and ARMv7) provide a software interface that includes:

- a *Debug Identification Register* (DIDR)
- status and control registers, including the *Debug Status and Control Register* (DSCR)
- hardware breakpoint and watchpoint support
- a *Debug Communications Channel* (DCC).

The ARMv7 software interface also includes reset, Power-down and Operating System support features.

The debug architecture also requires an external debug interface that supports access to the programmers' model.

The programmers' model can be used to manage and control Debug events. Watchpoints and breakpoints are two examples of Debug events. Debug events are described in Chapter 2 *Debug Events*.

You can configure the core through the DSCR into one of two Debug-modes:

Monitor Debug-mode

This causes a Debug Exception to occur as a result of a Debug event. Debug Exceptions are serviced through the same exception vectors as the Prefetch and Data Aborts, depending on whether they relate to instruction execution or data access.

Debug Exceptions are described in Chapter 3 *Debug Exceptions*.

Halting Debug-mode

This allows the system to enter a special Debug state when a Debug event occurs. When the system is in Debug state, the processor core ceases to execute instructions from the program counter location, but is instead controlled through the external debug interface and the *Instruction Transfer Register* (ITR) in particular. This allows an external agent, such as a debugger, to interrogate processor context, and control all subsequent instruction execution. Because the processor is stopped, it ignores the external system and cannot service interrupts.

Debug state is described in Chapter 4 *Debug State*.

A debug solution can use a mixture of the two methods, for example to support an OS or RTOS with both:

- *Running System Debug* (RSD) using Monitor Debug-mode
- Halting Debug-mode support available as a fallback for system failure and boot time debug.

The ability to switch between these two Debug-modes is fully supported by the architecture.

You can program the *Vector Catch Register* (VCR) to trap many exceptions. Trapped exceptions cause a Debug event. Untrapped exceptions cause a normal exception in the execution flow.

Copyright © 2006 ARM Limited. All rights reserved.

When both Debug-modes are disabled, debug is restricted to simple monitor solutions. These are usually ROM or Flash-based. Such a monitor might use standard system features, such as a UART or Ethernet connection, to communicate with a debug host. Alternatively, it might use the DCC as an out-of-band communications channel to the host. This minimizes its requirement on system resources.

This forms the basis of the *Debug Programmer's Model* (DPM) for ARMv6 and ARMv7.

1.2.1 Security Extensions and debug

Security Extensions debug allows you to disable invasive debug and non-invasive debug independently in either:

- all Secure modes
- only in Secure Privileged modes.

This is controlled by four input signals and two control bits in the Secure Debug Enable Register:

- the *Debug Enable* signal, **DBGEN**
- the *Non-Invasive Debug Enable* signal, **NIDEN**
- the *Secure Privileged Invasive Debug Enable* signal, **SPIDEN**
- the *Secure Privileged Non-Invasive Debug Enable* signal, **SPNIDEN**
- the *Secure User Invasive Debug Enable* bit, SUIDEN
- the *Secure User Non-invasive Debug Enable* bit, SUNIDEN.

For more information, see:

- *Invasive debug authentication* on page 2-4
- *Non-invasive debug authentication* on page 8-4
- the *ARM Architecture Reference Manual Security Extensions Supplement*, for details of the Secure Debug Enable Register
- *Authentication signals* on page 6-3 for details of the **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN** signals.

Copyright © 2006 ARM Limited. All rights reserved.

1.3 Performance counters

Performance Counters were implemented in several processors before ARMv7, however, before ARMv7 they did not form part of the architecture. The form described here follows those implementations with minor modifications to allow for potential future expansion.

The basic form consists of:

- A cycle counter, with the ability to count cycles or every sixty-fourth cycle.
- A number of performance counters whose events can be programmed. Previous implementations have provided up to four counters, but architecturally space is provided for up to 31 counters allowing for additional expansion (up to 31). The actual number of counters is IMPLEMENTATION DEFINED, and an identification mechanism is provided.
- Controls for enabling and resetting performance counters, to flag overflows and to enable interrupts on overflow. The cycle counter can be independently enabled from the rest of the performance counters.

The set of events that can be monitored split into those that are likely to be consistent across many micro-architectures and the rest, that are likely to be implementation specific. As a result, this architecture defines a core set of events to be used across many micro-architectures, together with a large space reserved for IMPLEMENTATION DEFINED events. The full set of events for any given implementation is IMPLEMENTATION DEFINED. There is no requirement to implement any of the core set of events, but the numbers allocated for the core set of events must not be used except as defined.

 Copyright © 2006 ARM Limited. All rights reserved.

1.4 Trace

Trace support is an architecture extension typically implemented using an *Embedded Trace Macrocell* (ETM). The ETM constructs a real-time trace stream corresponding to the operation of the processor. It is IMPLEMENTATION DEFINED whether the trace stream is stored locally in an Embedded Trace Buffer (ETB) for independent download and analysis, or whether it is exported directly through a trace port to a *Trace Port Analyzer* (TPA) and its associated host based trace debug tools.

Use of the ETM is non-invasive. Development tools can connect to the ETM, configure it, capture trace and download the trace without affecting the operation of the processor in any way. The ETM architecture extension provides an enhanced level of run-time system observation and debug granularity. It is particularly useful in cases where:

- Stopping the core affects the behavior of the system.
- There is insufficient state visible in a system by the time a problem is detected to be able to determine its cause. Trace provides a mechanism for system logging and back tracing of faults.

Trace might also be used to perform analysis of code running on the processor, such as performance analysis or code coverage.

The ETM architecture is documented separately. Licensees and third-party tools vendors should contact ARM Limited to ensure that they have the latest version. The ETM architecture specifies the following:

- the ETM programmers' model
- permitted trace protocol formats
- the physical trace port connector.

The ETM architecture version is defined with a major part and a minor part, in the form ETMvX.Y where X is the major version number and Y is the minor version number. The first ETM version that aligns with ARMv7 is ETMv3.3.

Copyright © 2006 ARM Limited. All rights reserved.

1.5 Register interfaces

This section gives a brief description of the different debug register interfaces defined by ARMv7. The most important distinction is between:

- the external debug interface, that defines how an external debugger can access the ARMv7 debug resources
- the processor interface, that describes how an ARMv7 processor core can access its own debug resources.

ARMv7 recommends an external debug interface based on the *ARM Debug Interface v5 Architecture Specification* (ADIv5). The most significant difference between ADIv5 and the interface recommended by ARMv6 is that ADIv5 supports debug over power-down of the processor core.

Although the ADIv5 interface is not required for compliance with the ARMv7 architecture, ARM's RealView tools require this interface to be implemented.

ADIv5 supports both a JTAG wire interface and a low pin-count *Serial Wire* interface. ARM's RealView tools support either wire interface.

An ADIv5 interface allows a debug object, such as an ARMv7 processor, to abstract a set of resources as a memory-mapped peripheral. Accesses to debug resources are made as 32-bit read/write transfers. Power-down is supported by introducing the abstraction that accesses to certain resources can return an error response when they are unavailable, just as a memory-mapped peripheral can return a slave-generated error response in exceptional circumstances.

The ARMv7 Debug Architecture requires software executing on the processor to be able to access all debug registers. To provide access to a particular basic subset of debug registers ARMv7 requires that the Baseline Coprocessor 14 (CP14) Interface is implemented, see *Baseline CP14 interface* on page 5-24. In order to provide access to the rest of the debug registers ARMv7 allows one of two options:

- An Extended CP14 Interface. This is similar to the requirement of the ARMv6 architecture.
- A Memory-mapped interface.

An implementation can include both of these options.

ARMv7 does not allow all combinations of debug, ETM, and performance monitor interfaces. There are three options for ARMv7 implementations, shown as options A, B, and C in Table 1-2 on page 1-8.

The ETM architecture provides the same implementation options as ARMv7 Debug. It is optional whether Trace registers are implemented but if they are implemented, the interface to them must be as shown in Table 1-2 on page 1-8.

Copyright © 2006 ARM Limited. All rights reserved.

Table 1-2 Options for interfacing to debug in ARMv7

Option	Processor interface to debug registers	Processor interface to trace registers	Processor interface to performance monitor
A	Baseline CP14 + Extended CP14	CP14	CP15
B	Baseline CP14 + Memory-mapped	Memory-mapped	CP15
C	Baseline CP14 + Extended CP14 + Memory-mapped	CP14 + Memory-mapped	CP15

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 2
Debug Events

This chapter contains the following sections:

- *Overview* on page 2-2
- *Invasive debug authentication* on page 2-4
- *Software Debug events* on page 2-6
- *Halting Debug events* on page 2-18
- *Generation of Debug events* on page 2-20
- *Debug event priority and order* on page 2-23.

Copyright © 2006 ARM Limited. All rights reserved.

2.1 Overview

A Debug event can be either:

- A Software Debug event, see *Software Debug events* on page 2-6
- A Hardware Debug event, see *Halting Debug events* on page 2-18.

A processor responds to a Debug event in one of the following ways:

- ignores the Debug event
- takes a Debug Exception, see Chapter 3 *Debug Exceptions*
- enters Debug State, see Chapter 4 *Debug State*.

The response depends on the configuration. This is shown in Table 2-1, and in Figure 2-1 on page 2-3.

Table 2-1 Processor behavior on Debug events

Configuration		**Behavior**			
Invasive debug permitted [a]	**DSCR[15:14]** [b]	**BKPT instruction**	**Other Software Debug event**	**Halting Debug event**	**Debug-mode selected and enabled**
No	bxx [c]	Debug exception[d]	Ignore	Ignore[e]	Disabled, not permitted
Yes	b00	Debug exception[d]	Ignore	Debug state entry[f]	None
Yes	bx1	Debug state entry	Debug state entry	Debug state entry	Halting
Yes	b10	Debug exception	Debug exception or Ignore, but see footnote[g]	Debug state entry[f]	Monitor

a. See *Invasive debug authentication* on page 2-4.
b. See *Halting Debug-mode enable, bit [14]* on page 10-15 and *Monitor Debug-mode enable, bit [15]* on page 10-15.
c. The value of DSCR[15:14] is ignored when invasive debug is not permitted. If **DBGEN** is LOW these bits read as zero.
d. When debug is disabled or not permitted, the BKPT instruction generates a Debug exception rather than being ignored. The DSCR, IFSR and IFAR are set to 1 as if a BKPT Instruction Debug Exception occurred. See *Effects of Debug Exceptions on CP15 registers and the WFAR* on page 3-4.
e. The processor might enter Debug State later, see *Halting Debug events* on page 2-18.
f. In ARMv6, these entries are IMPLEMENTATION DEFINED, see the *ARM Architecture Reference Manual*.
g. Be careful when programming Debug events when Monitor Debug-mode is selected and enabled, because certain conditions can lead to UNPREDICTABLE behavior, see *unpredictable behavior on Software Debug events* on page 2-14.

Copyright © 2006 ARM Limited. All rights reserved.

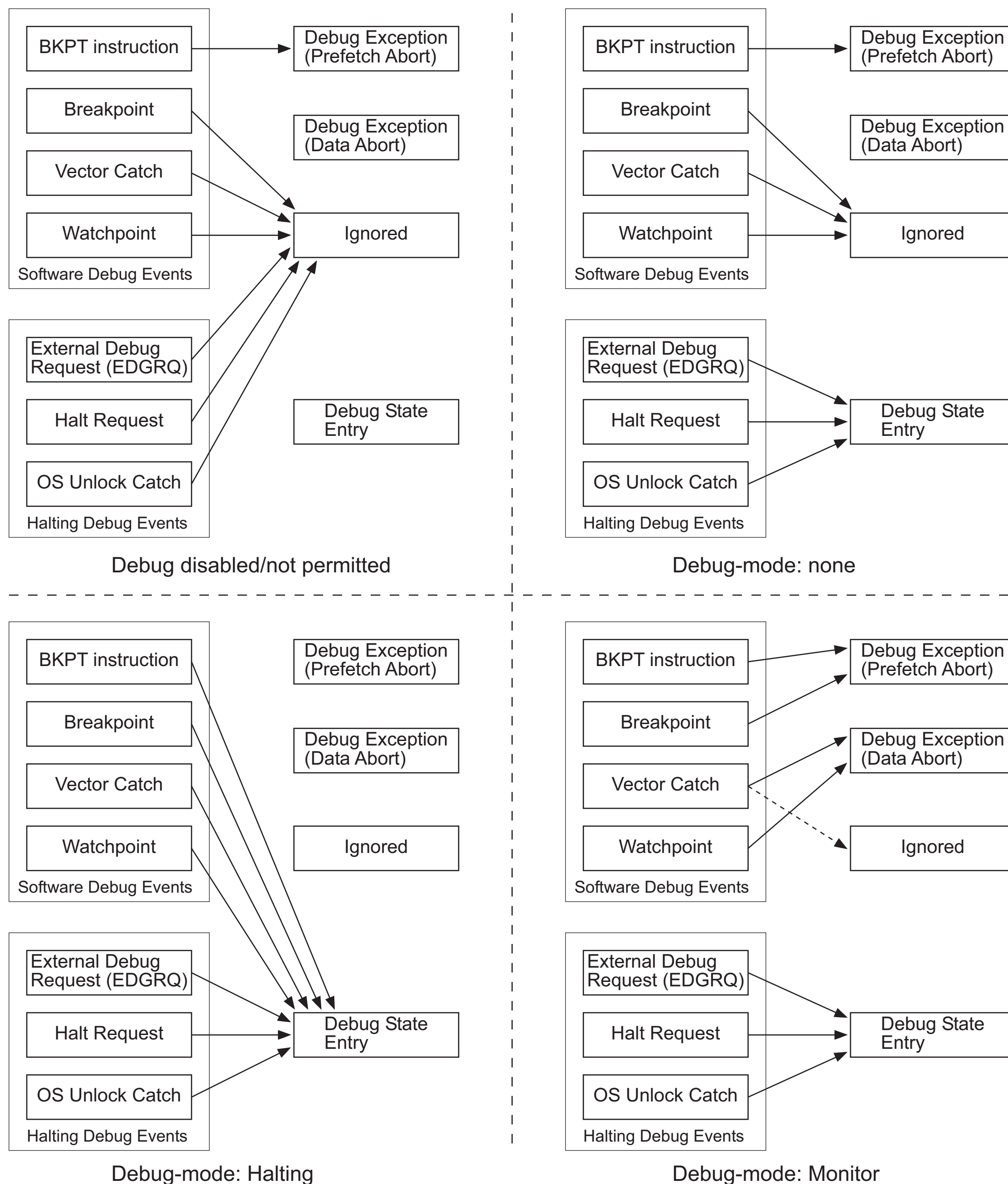


Figure 2-1 Processor behavior on Debug events

 Copyright © 2006 ARM Limited. All rights reserved.

2.2 Invasive debug authentication

Invasive debug can be enabled or disabled. If it is disabled the processor ignores all debug events except BKPT Instruction. This means that debug events other than BKPT Instruction do not cause the processor to enter Debug state or to take a debug exception.

In addition, if a processor implements the Security Extensions, invasive debug can be permitted or not permitted. If invasive debug is not permitted, the processor again ignores all debug events except BKPT Instruction.

The difference between enabled and permitted is that permitted debug operation depends on the security state and the operating mode of the processor. The alternatives for when Invasive debug is permitted are:

- in all processor modes, in both Secure and Nonsecure worlds
- only in Nonsecure world
- in Nonsecure world and Secure User mode.

The external debug interface signals that control the enabling and permitting of Debug events are **DBGEN** and **SPIDEN**. **SPIDEN** is only implemented on processors that implement Security Extensions. See *Authentication signals* on page 6-3.

If **DBGEN** is LOW, all invasive debug is disabled.

On processors that do not implement Security Extensions, if **DBGEN** is HIGH, invasive debug is enabled and permitted in all modes, see Table 2-2.

Table 2-2 Invasive debug authentication, Security Extensions not implemented

DBGEN	Modes in which Invasive debug is permitted
LOW	None
HIGH	All modes

On processors that implement Security Extensions, if both **DBGEN** and **SPIDEN** are HIGH, invasive debug is enabled and permitted in all modes and in both Secure and Nonsecure worlds. If **DBGEN** is HIGH and **SPIDEN** is LOW:

- Invasive debug is enabled and permitted in the Nonsecure world.
- Invasive debug is not permitted in Secure privileged modes.
- Whether invasive debug is permitted in Secure User mode depends on the value of the SUIDEN bit in the Secure Debug Enable (SDE) Register. See the *ARM Architecture Reference Manual Security Extensions Supplement* for details of the Secure Debug Enable Register.

This is shown in Table 2-3 on page 2-5.

 Copyright © 2006 ARM Limited. All rights reserved.

Table 2-3 Invasive debug authentication, Security Extensions implemented

DBGEN	SPIDEN	SUIDEN	Modes in which invasive debug is permitted
LOW	X	X	None
HIGH	LOW	0	All modes in Nonsecure world
HIGH	LOW	1	All modes in Nonsecure world, and Secure User mode
HIGH	HIGH	X	All modes in all worlds

Note

If you only enable invasive debug when the processor is in a Nonsecure mode this only protects your secure processing from direct observation or invasion by an untrusted debugger. System designers must be aware that such configurations might still allow attacks such as denial of service attacks. For example:

- The processor can be forced into Debug state from a Nonsecure mode, preventing processing of secure operations.
- The Interrupts disable bit in the Debug Status and Control Register can be used to prevent servicing of secure interrupts.

ARM Limited recommends that you disable invasive debug in all modes where you are concerned about such attacks.

Copyright © 2006 ARM Limited. All rights reserved.

2.3 Software Debug events

A Software Debug event can be any of the following:

- A Watchpoint Debug event, see *Watchpoint Debug events*
- A Breakpoint Debug event, see *Breakpoint Debug events* on page 2-10
- A BKPT Instruction Debug event, see *BKPT Instruction Debug events* on page 2-12
- A Vector Catch Debug event, see *Vector Catch Debug events* on page 2-13.

If Monitor Debug-mode is selected and enabled, the behavior of certain types of Software Debug events is noted in the following sections as UNPREDICTABLE. See *unpredictable behavior on Software Debug events* on page 2-14 for more information.

2.3.1 Watchpoint Debug events

Watchpoint Debug events are controlled by pairs of registers referred to as Watchpoint Register Pairs (WRPs). An implementation may contain many WRPs. The first two WRPs are identified as WRP0 and WRP1, and this numbering pattern continues for additional WRPs. WRPn refers to any particular pair. WRPn consists of two registers:

- a Watchpoint Control Register, WCRn
- a Watchpoint Value Register, WVRn.

For a given Watchpoint Register Pair, WRPn, a Watchpoint Debug event occurs when all of the following are true:

- The *Data Virtual Address* (DVA) matches the value in WVRn. See *Memory addresses* on page 2-13 for a definition of the DVA.
- At the time when the watchpoint is evaluated, all the conditions of WCRn match.
- The watchpoint is enabled.
- If linking is enabled in WCRn, the linked Context ID matching Breakpoint Register Pair (BRP) must meet the following conditions:
  - the BRP is enabled
  - the value held in the BRP matches the Context ID in CP15 register 13.

  See *Breakpoint Debug events* on page 2-10 for more information about BRPs.
- The instruction that initiated the memory access is committed for execution. Watchpoint Debug events are only generated if the instruction passes its condition code.

All instructions that the *ARM Architecture Reference Manual* defines as memory access instructions can generate Watchpoint Debug events.

Copyright © 2006 ARM Limited. All rights reserved.

It is IMPLEMENTATION DEFINED whether either or both of the memory hint instructions, `PLD` and `PLI`, generate Watchpoint Debug events. When Watchpoint Debug event generation by either or both of the `PLD` and `PLI` instructions is implemented, the behavior must be:

- if `PLI` is implemented, no watchpoint is generated in a situation where the instruction would generate a prefetch abort, if it was a real fetch rather than a hint
- If `PLD` is implemented, no watchpoint is generated in a situation where the instruction would generate a Data Abort, if it was a real data access rather than a hint
- in all other situations a Watchpoint Debug event is generated.

It is IMPLEMENTATION DEFINED whether the following cache maintenance operations generate Watchpoint Debug events:

- clean the data cache by modified virtual address (MVA)
- invalidate the data cache by MVA
- invalidate the instruction cache by MVA
- clean and invalidate the data cache by MVA.

Note

When Watchpoint Debug event generation by these cache maintenance operations is implemented, these operations must generate Watchpoint Debug events on a DVA match, regardless of whether the data is stored in any cache.

For regular data accesses, the Watchpoint Debug event generation includes a comparison of the WCR contents with the size of access. For the purpose of this comparison, the sizes of these operations are IMPLEMENTATION DEFINED:

- memory hints, `PLD` and `PLI`
- cache maintenance operations.

Watchpoint Debug events can be *Precise* or *Imprecise*:

- Precise Watchpoint Debug events act like a precise Data Abort exception on the data access instruction itself
- Imprecise Watchpoint Debug events act like an exception that cancels a later instruction.

For more information see *Precise and Imprecise Watchpoint Debug events* on page 2-8.

For the ordering of debug events, the ARMv7 architecture requires that:

- Regardless of the actual ordering of memory accesses, Watchpoint Debug events must be taken in program order. See *Debug event priority and order* on page 2-23.
- Watchpoint Debug events must behave as if they were evaluated before the memory access is observed, regardless of whether the Watchpoint Debug event is precise or imprecise. See *Generation of Debug events* on page 2-20.

Copyright © 2006 ARM Limited. All rights reserved.

Precise and Imprecise Watchpoint Debug events

ARMv7 allows watchpoints to be either *precise* or *imprecise*. An implementation can implement precise watchpoints, imprecise watchpoints, or both. It is IMPLEMENTATION DEFINED under what circumstances a watchpoint is precise or imprecise.

ARMv6 only allows imprecise watchpoints.

Precise Watchpoint Debug events

A Precise Watchpoint Debug event has the same behavior as a precise Data Abort:

- If Monitor Debug-mode is enabled, R14_abt is set to the address of the watchpointed exception + 8.
- The Debug event occurs before any following instructions or exceptions have altered the state of the processor.
- The value in the base register for the memory access is not updated.

  Note

  The Base Updated Abort Model is not permitted in ARMv7.

- If the instruction was a register load, the data returned is marked as invalid. If the instruction was a single register load, the destination is not updated. If the instruction loaded multiple registers, the values in the destination registers, other than the PC and base register, are UNPREDICTABLE.
- If the instruction is a coprocessor load, the values left in the coprocessor register are UNPREDICTABLE.
- If the instruction is a store, the content of the memory location written to is unchanged.

On a Precise Watchpoint Debug event, the DSCR[5:2] Method of Debug Entry bits are set to Precise Watchpoint Occurred.

If a Precise Watchpoint Debug event is signalled by a memory operation other than the first operation of an instruction that causes multiple operations, such as the `LDM` and `LDC` instructions, to Device or Strongly Ordered memory, the memory access rules may not be maintained.

For example, if the second memory operation of an `STM` instruction signals a Precise Watchpoint Debug event, then when the instruction is re-tried following processing of the Debug event, the first memory operation would be repeated. This behavior is not normally permitted for Device or Strongly Ordered memory.

To avoid this circumstance, debuggers should not set watchpoints on addresses within regions of Device or Strongly Ordered memory which may be accessed in this way. The address masking features of watchpoints may be used to set a watchpoint on an entire region, thereby ensuring the Precise Watchpoint Debug event is taken on the first operation of such an instruction.

Copyright © 2006 ARM Limited. All rights reserved.

Imprecise Watchpoint Debug events

An Imprecise Watchpoint Debug event has similar, but not identical, behavior to an imprecise Data Abort. Its behavior is:

- The state of the processor on the imprecise watchpoint exception must be such that the processor can return to the instruction cancelled by the Debug event.
- If Monitor Debug-mode is enabled, R14_abt is set to the address of the instruction to return to + 8.
- The watchpointed instruction *must* have completed, and other instructions that followed it, in program order, might have completed. For more information see *Recognizing Imprecise Watchpoint Debug events*.
- The watchpoint *must* be taken before any exceptions that occur in program order after the watchpoint is triggered.
- All the registers written by the watchpointed instruction are updated.
- Any memory accessed by the watchpointed instruction is updated.

An imprecise watchpoint is not an External Abort. An imprecise watchpoint:

- is not affected by the EA bit in the Secure Configuration Register (SCR)
- is not ignored when the A flag, bit [8] of the CPSR, is set to 1.

An imprecise watchpoint is also more constrained than an imprecise Data Abort, because it must be possible to return to the program that caused the watchpoint.

On an Imprecise Watchpoint Debug event, the DSCR[5:2] Method of Debug Entry bits are set to Imprecise Watchpoint Occurred.

Recognizing Imprecise Watchpoint Debug events

When an instruction that consists of multiple memory operations is accessing Device or Strongly Ordered memory, and an Imprecise Watchpoint Debug event is signalled by a memory operation other than the first operation of the instruction, the Debug event *must not* cause Debug state entry or a Debug Exception until all the operations have completed. This ensures the memory access rules for Device and Strongly Ordered memory are preserved.

Examples of instructions that cause multiple memory operations are the `LDM` and `LDC` instructions.

Note

To understand why the architecture does not allow the Imprecise Watchpoint Debug event to be taken before the watchpointed instruction completes, consider an `LDM` instruction accessing Device or Strongly Ordered memory, with an Imprecise Watchpoint Debug event signalled after the first word of memory is accessed. If the Debug event was taken immediately, the `LDM` would be re-executed on return from the event handler. This would cause a new access to the first word of memory, breaking the rule that, for Device or Strongly Ordered memory, each memory operation of an instruction is issued precisely once.

Copyright © 2006 ARM Limited. All rights reserved.

2.3.2 Breakpoint Debug events

Breakpoint Debug events are controlled by pairs of registers referred to as *Breakpoint Register Pairs* (BRPs). An implementation may contain many BRPs. The first two BRPs are identified as BRP0 and BRP1, and this numbering pattern continues for additional BRPs. BRPn refers to any particular pair. BRPn consists of two registers:

- a Breakpoint Control Register, BCRn
- a Breakpoint Value Register, BVRn.

For a given Breakpoint Register Pair, BRPn, a Breakpoint Debug event occurs when:

- BCRn is programmed for linked or unlinked *Instruction Virtual Address* (IVA) match or mis-match. See *Memory addresses* on page 2-13 for a definition of the IVA.
- The IVA of an instruction is compared with the BVRn value:
  - — for a match, if BCRn is programmed for matches
  - — for a mis-match, if BCRn is programmed for mis-matches.

  Note

  For more information on address matching and mis-matching, see *Variable length instruction sets* on page 2-11 and *Byte address select, bits [8:5]* on page 10-41.

- At the time when the breakpoint is evaluated, all the conditions specified in BCRn match.
- The breakpoint is enabled.
- If BCRn is programmed for linked Context ID match and, at the time the breakpoint is evaluated:
  - — the linked BRP indicated by BCRn, BRPm, is enabled
  - — the value held in BVRm matches the Context ID held in CP15 register 13.
- The instruction is committed for execution. The Debug event is generated whether the instruction passes or fails its condition code.

A Breakpoint Debug event also occurs when:

- BCRn is programmed for unlinked Context ID match.
- The Context ID, in CP15 register 13, matches the BVRn value.
- At the same time as the breakpoint is evaluated, all the conditions specified in BCRn match.
- The breakpoint is enabled.
- The instruction is committed for execution. The Debug event is generated whether the instruction passes or fails its condition code.

 Copyright © 2006 ARM Limited. All rights reserved.

If Monitor Debug-mode is selected and enabled then the behavior is UNPREDICTABLE if both of the following are true:

- BCRn is programmed for linked or unlinked IVA mismatch, or BCRn is programmed for unlinked Context ID match.
- BCRn is programmed to generate Debug Events in a privileged mode or in any mode.

Breakpoint Debug events are precise. That is, the Debug event acts like an exception that cancels the breakpointed instruction. Breakpoint Debug events must be evaluated before the instruction is executed. See *Generation of Debug events* on page 2-20.

For more information, see *Breakpoint Value Registers (BVRn)* on page 10-39 and *Breakpoint Control Registers (BCRn)* on page 10-40.

Variable length instruction sets

In the ARMv7 Architecture Specification:

- a variable length instruction set is one where instructions comprise one or more units of memory, each of a fixed size.
- a fixed length instruction set is one where instructions are always a single unit of memory of a fixed size.

Thumb-2, Thumb-2EE and Java byte codes are examples of variable length instruction sets. In Thumb-2 and Thumb-2EE, an instruction comprises one or two halfwords. In Java, an instruction comprises one or more bytes.

In the Thumb instruction set, the `BL` instruction consists of two halfwords:

- On processors that do not implement Thumb-2 instructions, implementations can choose to execute the two halfwords separately. In these cases Thumb is considered as a fixed length instruction set
- Alternatively, in an implementation where the Thumb `BL` instruction is executed as a single instruction, Thumb is considered as a variable length instruction set.

The distinction between these two implementations of the Thumb instruction set is whether the second half of the `BL` instruction can be considered as a valid target for an exception return. In other words, Thumb is implemented as a variable length instruction set if an exception can be taken between executing the two halves of a `BL` instruction.

ARM instructions are all a fixed size, and therefore ARM is fixed length instruction set.

In ARMv7, a BRP configured such that a match occurs on an address other than the first unit of the instruction does not cause the instruction to generate a Breakpoint Debug event. For example, on a Thumb-2 instruction, a BRP that matches the second halfword of the instruction will not cause the instruction to generate a Breakpoint Debug event.

Instructions in a fixed length instruction set consist of a single unit, which is therefore the first unit of the instruction. For instructions in a variable length instruction set consisting of more than one unit, the first unit of the instruction is defined as the unit of the instruction with the lowest address in memory.

Copyright © 2006 ARM Limited. All rights reserved.

In ARMv6 the behavior of breakpoints that match on an instruction memory unit other than the first one is IMPLEMENTATION DEFINED. However, ARMv6 specifies that, for the Java instruction set, breakpoints matching on operands do not cause the instruction to be breakpointed. A Breakpoint Debug event is only generated if the BRP is configured to match the opcode. The opcode is always the first memory unit of the instruction.

In all cases, a debugger must configure the BRP such that it matches on all bytes of the first unit of the instruction, otherwise the generation of Breakpoint Debug events is UNPREDICTABLE.

On a ARMv6 architecture processor that does not implement Thumb-2 instructions and that allows an exception to be taken between executing the two halves of a Thumb `BL` instruction, a Debugger must treat the Thumb `BL` as two instructions, and therefore set breakpoints on both halves of the instruction. This might require two BRPs.

Note

To ensure compatibility across ARMv6 implementations, a debugger can always treat `BL` as two instructions when debugging code on an ARMv6 processor that does not implement Thumb-2 instructions.

For example:

- On an ARMv7 processor:
  - To breakpoint on a 32-bit Thumb-2 instruction starting at address `0x8000`, a Debugger must set BVRn = `0x8000` and BCRn[8:5] = b0011. These are the settings for breakpointing on any Thumb-2 instruction, including a `BL`.
  - To breakpoint on a 32-bit Thumb-2 instruction starting at address `0x8002`, a Debugger must set BVRn = `0x8000` and BCRn[8:5] = b1100.
  - To breakpoint on an ARM instruction starting at address `0x8004`, a Debugger must set BVRn = `0x8004` and BCRn[8:5] = b1111.
- On an ARMv6 processor that does not implement Thumb-2:
  - To breakpoint on a Thumb `BL` instruction at address `0xC120`, a Debugger must set BVRn = `0xC120` and BCRn[8:5] = b1111.
  - To breakpoint on a Thumb `BL` instruction at address `0xC122`, a Debugger must set BVRn = `0xC120`, BVRm = `0xC124`, BCRn[8:5] = b1100 and BCRm[8:5] = b0011.

2.3.3 BKPT Instruction Debug events

A BKPT Instruction Debug event occurs when a `BKPT` instruction is committed for execution. `BKPT` is an unconditional instruction.

BKPT Instruction Debug events are precise. That is, the Debug event acts like an exception that cancels the `BKPT` instruction.

See the *ARM Architecture Reference Manual* for details of the `BKPT` instruction and its encodings in the ARM and Thumb instruction sets.

 Copyright © 2006 ARM Limited. All rights reserved.

2.3.4 Vector Catch Debug events

Vector Catch Debug events are controlled by the Vector Catch Register (VCR).

A Vector Catch Debug event occurs when:

- The IVA of an instruction matches a vector location address for the current security world.

  See *Memory addresses* for a definition of the IVA.

- At the same time as the vector catch is evaluated, the corresponding bit of the VCR is set to 1, indicating vector catch enabled.
- The instruction is committed for execution. The Debug event is generated whether the instruction passes or fails its condition code.

Vector Catch Debug events must be evaluated before the instruction is executed. The vector catch behaves exactly like a BRPn set with BVRn[31:2] set to the top 30 bits of the exception vector address, and BCRn[8:5], the Byte Address Select field, set to b1111 and BCRn programmed for unlinked IVA match.

Note

Under this model, any instruction prefetched from an exception vector address that is committed for execution can trigger a Vector Catch Debug event, not just those due to exception entries.

Instruction fetches from non word-aligned addresses within the 4-bytes of the exception vector address also trigger vector catches. For example, a Thumb instruction fetch from the second half-word of the address can trigger a vector catch.

However, unlike breakpoints, if the vector catch address matches a unit of an instruction in a variable length instruction set that is not the first unit of the instruction, vector catch generation is UNPREDICTABLE. See *Variable length instruction sets* on page 2-11 for more information on breakpoint generation and variable length instruction sets.

If Monitor Debug-mode is selected and enabled, and the vector is either the Prefetch abort vector or the Data Abort vector, the Debug event is ignored.

Vector Catch Debug events are precise. That is, the Debug event acts like an exception that cancels the instruction at the caught vector. Vector Catch Debug events must be evaluated before the instruction is executed. See *Generation of Debug events* on page 2-20.

For more information, see *Vector Catch Register (VCR)* on page 10-54.

2.3.5 Memory addresses

On processors that implement the *Virtual Memory System Architecture* (VMSA), and also implement the *Fast Context Switch Extension* (FCSE):

- It is IMPLEMENTATION DEFINED whether the *Instruction Virtual Address* (IVA) used in generating Breakpoint Debug events is the *Modified Virtual Address* (MVA) or *Virtual Address* (VA) of the instruction.

Copyright © 2006 ARM Limited. All rights reserved.

- It is IMPLEMENTATION DEFINED whether the *Data Virtual Address* (DVA) used in generating Watchpoint Debug events is the MVA or VA of the data access.
- The IVA used in generating Vector Catch Debug events is always the VA of the instruction.
- The *Watchpoint Fault Address Register* (WFAR) reads a VA plus an offset dependent on the processor state.
- The *Program Counter Sampling Register* (PCSR) reads a VA plus an offset dependent on the processor state.

Note

The use of the FCSE is deprecated in ARMv7.

On processors that implement the VMSA, and do not implement the FCSE:

- The IVA used in generating Breakpoint Debug events is the VA of the instruction.
- The DVA used in generating Watchpoint Debug events is the VA of the data access.
- The IVA used in generating Vector Catch Debug events is the VA of the instruction.
- The WFAR reads a VA plus an offset dependent on the processor state.
- The PCSR reads a VA plus an offset dependent on the processor state.

On processors that implement the *Protected Memory System Architecture* (PMSA), all addresses are *Physical Addresses* (PAs):

- The IVA used in generating Breakpoint Debug events is the PA of the instruction.
- The DVA used in generating Watchpoint Debug events is the PA of the data access.
- The IVA used in generating Vector Catch Debug events is the PA of the instruction.
- The WFAR reads a PA plus an offset dependent on the processor state.
- The PCSR reads a PA plus an offset dependent on the processor state.

For more information about the WFAR, see *Effects of Debug Exceptions on CP15 registers and the WFAR* on page 3-4, *The effect of entering Debug state on CP15 registers and the WFAR* on page 4-4, and *Watchpoint Fault Address Register (WFAR)* on page 10-22.

For more information about the PCSR, see *Program counter sampling register* on page 8-3 and *Program Counter Sampling Register (PSCR)* on page 10-31.

2.3.6 UNPREDICTABLE behavior on Software Debug events

Vector Catch Debug events on the Prefetch abort and Data Abort vectors are ignored if Monitor Debug-mode is configured, because they would lead to an unrecoverable state.

ARM Limited recommends that VCR[28,27,12,11,4,3] are always programmed as zero when Monitor Debug-mode is configured, see *Vector Catch Register (VCR)* on page 10-54.

Vector Catch Debug events on the Secure Monitor Call vector are not ignored. However, VCR[10] should normally also be programmed as zero; see *Monitor Debug-mode vector catch on Secure Monitor Call* on page 2-17.

Copyright © 2006 ARM Limited. All rights reserved.

In ARMv6 the following events are ignored if Monitor Debug-mode is configured, because they can lead to an unrecoverable state:

- Unlinked Context ID Breakpoint Debug events, if the processor is running in a privileged mode.
- Breakpoint Debug events with BCR[22:21] = b10, if the processor is running in a privileged mode.

In ARMv7, debuggers must avoid these cases by restricting the programming of the BCR if Monitor Debug-mode is enabled and selected. This means that the allowed values of the Privileged Mode control bits, bits [2:1], of BCRn must be restricted in the following cases:

- if BCRn[22:20] = b010, selecting an unlinked Context ID breakpoint
- If BCRn[22:20] = b100 or b101, selecting an IVA mismatch breakpoint.

For these cases, BCRn[2:1] must be programmed to one of:

- b00, selecting match only in User, Supervisor or System Mode
- b10, selecting match only in User Mode.

See *Debug exceptions in abort handlers* for additional points that must be considered before using the b00 setting.

Caution

The following values must not be selected for BCRn[2:1]:

- b01, match in any privileged mode
- b11, match in any mode.

For details of programming the BCR see *Breakpoint Control Registers (BCRn)* on page 10-40.

If these restrictions are not followed, the processor behavior on a resulting Debug event is UNPREDICTABLE. The processor can enter an unrecoverable state, because taking the Debug Exception does not take the processor out of the state in which the Debug event occurs.

Debug exceptions in abort handlers

The previous section indicated that, in ARMv7, a debugger might set BCR[2:1] to b00, match in User, Supervisor and System Modes, to avoid the possibility of reaching an unrecoverable state in the unlinked Context ID and IVA mismatch breakpoint cases when Monitor Debug-mode is selected. However, BCR[2:1] must only be programmed to b00 if you are confident that the abort handler will not switch to one of these modes before saving context that may be corrupted by a additional Debug event. The context that might be corrupted by such an event includes R14_abt, SPSR_abt, IFAR, DFAR, and DFSR.

It is unlikely that an abort handler would switch to User Mode to process an abort before saving these registers, so setting BCR[2:1] to b10, match only in User Mode, is safer

Also, take care when setting a Breakpoint or BKPT Instruction Debug event inside a Prefetch abort or Data Abort handler, or when setting a Watchpoint Debug event on a data address that might be accessed by any of these handlers.

Copyright © 2006 ARM Limited. All rights reserved.

In general, the user must only set Breakpoint or BKPT Instruction Debug events inside an abort handler at a point after the context that would be corrupted by a Debug event has been saved. Breakpoint Debug events in code that may be run by an abort handler can be avoided by setting BCR[2:1] to b00 or b01, as appropriate.

Watchpoint Debug events in abort handlers can be avoided by setting WCR[2:1] for the watchpoint to b10, match only non-privileged accesses, if the code being debugged is not running in a privileged mode.

Failure to follow these guidelines may lead to Debug events occurring before the handler is able to save the context of the abort causing the context to be overwritten, resulting in UNPREDICTABLE software behavior. The context that might be corrupted by such events includes R14_abt, SPSR_abt, IFAR, DFAR, and DFSR.

Debug events in the debug monitor

Because Debug Exceptions are overlaid on top of the Data Abort and Prefetch abort exceptions, the precautions outlined in the section *Debug exceptions in abort handlers* on page 2-15 also apply to debug monitors. The suggested settings for breakpoints and watchpoints that can avoid taking Debug Exceptions within a Data Abort handler can be used to avoid taking Debug Exceptions in the debug monitor.

In addition, particularly on ARMv7 processors that do not implement the Extended CP14 Interface, and particularly those that implement Precise Watchpoint Debug events, when Monitor Debug-mode is enabled debuggers must avoid:

- setting Watchpoint Debug events on the addresses of debug registers
- setting Breakpoint and Vector Catch Debug events on the addresses of instruction within the debug monitor.

In particular, it is unwise to set a watchpoint on the address of the Watchpoint Control Register (WCR) for that watchpoint, or to set a breakpoint on the address of an instruction that disables the breakpoint.

As noted in section *Generation of Debug events* on page 2-20:

- a write to the WCR for a watchpoint set on the address of that WCR, to disable that watchpoint will trigger the watchpoint
- an instruction that disables a breakpoint on that instruction will trigger the breakpoint.

In the first of these cases:

- if watchpoints are imprecise, the write to the WCR still takes place and the watchpoint is disabled. The debug software must then deal with the re-entrant Debug Exception.
- if watchpoints are precise the value in the WCR after the watchpoint is signaled is unchanged, and the Debug event is left enabled.

In the breakpoint case, the Debug Exception will be taken before the Debug event is disabled.

In both the watchpoint and the breakpoint case it might be impossible to recover.

 Copyright © 2006 ARM Limited. All rights reserved.

Monitor Debug-mode vector catch on Secure Monitor Call

Debuggers must also be cautious about programming a Vector Catch Debug event on the Secure Monitor Call (SMC) vector when Monitor Debug-mode is enabled. If such an event is programmed, the following sequence can occur:

1. Nonsecure code executes an `SMC` instruction.
2. The processor takes the `SMC` exception, branching to the Monitor vector in Monitor Mode. SCR[0] = 1, indicating the `SMC` originated in the Nonsecure world.
3. The Vector Catch Debug event is taken. Although SCR[0] = 1, the processor is in the Secure World because it is in Monitor Mode.
4. The processor jumps to the Secure Prefetch abort vector, and sets SCR[0] = 0.

   Note

   Aborts taken in Secure World cause SCR[0] to be set to 0.

5. The abort handler at the Secure Prefetch abort handler can tell a Vector Catch Debug event occurred, and can determine the address of the `SMC` instruction from R14_mon. However, it cannot determine whether that is a Secure or Nonsecure address.

Therefore, ARM Limited advises you not to program a Vector Catch Debug event on the SMC vector when Monitor Debug-mode is enabled.

Note

This is not a security issue, because the sequence given here can only occur if **SPIDEN** is HIGH.

 Copyright © 2006 ARM Limited. All rights reserved.

2.4 Halting Debug events

Halting Debug events are caused by the external debug interface requesting that the processor enters Debug state. The three types of Halting Debug event are:

- Activation of the External Debug Request signal, **EDBGRQ**.

  This signal can be driven by, for example:

  - an external agent such as a cross-triggering unit
  - an ETM.

  See *EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK* on page 6-4.

- A Halt Request Debug event. This occurs following receipt of a Halt Request command.

  The Halt Request command is activated by a debugger writing 1 to the Halt Request bit of the Debug Run Control Register (DRCR). The processor debug control logic holds the request until the processor enters Debug state.

  See *Halt Request, bit [0]* on page 10-23.

- An OS Unlock Catch Debug event. This occurs when both of the following are true:

  - the OS Unlock Catch is enabled in the Event Catch Register
  - the OS Lock transitions from being locked to being unlocked.

  The event is held by the debug control logic until the processor enters Debug state.

  See *OS Unlock Catch, bit [0]* on page 10-57 and *OS Lock Access Register (OSLAR)* on page 10-58 for details.

If debug is disabled when one of these events is detected, the request is ignored and no Halting Debug event occurs. Debug is disabled when the external debug interface signal **DBGEN** is LOW.

If **DBGEN** is HIGH, meaning that debug is enabled, and a Halting Debug event occurs when debug is not permitted, the Halting Debug event is pended. This means that the processor enters Debug state when it transitions to a security world or processor mode where debug is permitted.

While a Halting Debug event is pended:

- For External Debug requests, the requestor holds **EBDGRQ** HIGH until the core enters Debug state.
- For the other Halting Debug events the event is pended internally until the processor enters a security world or processor mode where debug is permitted. However, if **DBGEN** goes LOW before the processor enters the world or mode where debug is permitted, it is UNPREDICTABLE whether the processor keeps the event pended. If the debug logic is reset before the processor enters the permitted world or mode, the processor must remove the pending Halting Debug event.

If a Halting Debug event occurs when debug is enabled and permitted, or debug becomes enabled and permitted whilst a Halting Debug event is pending, it is guaranteed that Debug state is entered by the end of the next Instruction Synchronization Barrier (ISB) operation, exception entry, or exception return.

 Copyright © 2006 ARM Limited. All rights reserved.

In v6 Debug and v6.1 Debug:

- if you are using the recommended ARM Debug Interface v4.0, the Halt Request command is issued by placing the HALT instruction in the IR register and taking the Debug Test Access Port State Machine (DBGTAPSM) through the Run-Test/Idle state
- the OS Unlock Catch Debug event is not supported.

In v6 Debug it is IMPLEMENTATION DEFINED whether Halting Debug events cause entry to Debug state when Halting Debug-mode is not configured and enabled

 Copyright © 2006 ARM Limited. All rights reserved.

2.5 Generation of Debug events

The generation of Breakpoint and Watchpoint Debug events can be dependent on the context of the processor, including:

- the current processor mode
- the contents of the Context ID register
- the Secure world setting, if the processor implements Security Extensions.

The generation of Debug events is also dependent on the state of the debug logic:

- Breakpoint Debug events are dependent on the contents of the relevant Breakpoint Register Pair (BRP)
- Watchpoint Debug events are dependent on the contents of the relevant Watchpoint Register Pair (WRP)
- Linked Breakpoint or Watchpoint Debug events are dependent on the settings of a second BRP
- Vector Catch Debug events are dependent on the settings in the Vector Catch Register (VCR)
- OS Unlock Catch Debug events are dependent on the setting of the Event Catch Register (ECR).

In addition, as shown in Table 2-1 on page 2-2, the generation of Debug events is dependent on:

- the settings of the authentication signals, see *Authentication signals* on page 6-3
- the values in the Debug Status and Control Register (DSCR), see *Halting Debug-mode enable, bit [14]* on page 10-15, and *Monitor Debug-mode enable, bit [15]* on page 10-15.

The following events are guaranteed to take effect on the Debug event generation logic by the end of the next Instruction Synchronization Barrier (ISB) operation, exception entry, or exception return:

- Context changing operations, including:
  - mode changes
  - writes to the Context ID register
  - Secure world changes.
- Operations that change the state of the Debug event generation logic, including:
  - writes to BRP registers, for Breakpoint Debug events, or linked Breakpoint or Watchpoint Debug events
  - writes to WRP registers, for Watchpoint Debug events
  - writes to the VCR register, for Vector Catch Debug events
  - writes to the ECR, for OS Unlock Catch Debug events
  - changes to the authentication signals
  - writes to the DSCR register.

 Copyright © 2006 ARM Limited. All rights reserved.

Exception return sequences are usually also context changing operations, and hence the context change operation is guaranteed to take effect on the breakpoint matching logic by the end of that exception return sequence.

If you require a change in the Debug event generation logic to complete before a particular event or piece of code is debugged then you must ensure there is an explicit synchronization operation after the change in the Debug settings. In the absence of an explicit synchronization operation, the changes take effect as operations drain down the pipeline. This might be acceptable in some situations, if you require only that the debug changes take effect within a number of instructions.

An explicit synchronization operation is one of:

- an exception entry
- a return from exception
- an Instruction Synchronization Barrier (ISB) instruction.

Between a context change operation and the end of the next explicit synchronization operation it is UNPREDICTABLE whether the processing of a Debug event will depend the old or the new context.

Between operations that change the state of the Debug event generation logic and the end of the next ISB, exception entry or exception return, it is UNPREDICTABLE whether Debug event generation depends on the old or the new settings.

Note

The rules for how a write to a debug register through a Memory-mapped interface has side-effects, for example, the enabling or disabling of a Debug event, are defined by the memory ordering model of the ARM Architecture. See *Synchronization of debug register updates* on page 5-18 and the ARM Architecture Reference Manual for more information.

It is not a requirement of this architecture that such changes take effect on instruction fetches from the memory system, or on memory accesses made by the processor, at the same point as they take effect on the debug logic. The only architectural requirement is that such a change executed before an Instruction Synchronization Barrier (ISB) operation must be visible to both the memory system and the debug logic for all instructions executed after the ISB operation. This requirement is described earlier in this section.

Watchpoint Debug events must be evaluated before a memory access operation is observed. Breakpoint and Vector Catch Debug events must be evaluated before the instruction is executed, that is, before the instruction has any effect on the architectural state of the processor. As a result, if the instruction is one that modifies the context in which Debug events are evaluated, the Debug event must be evaluated in terms of the context before the memory access operation is observed or the instruction executes. For example:

- In an ARMv7 implementation that uses the Memory-mapped interface, if the Watchpoint Control Register (WCR) is located in Device or Strongly Ordered memory, a write to the WCR to enable a watchpoint on a Data Virtual Address (DVA) of the WCR itself must not trigger the watchpoint.

 Copyright © 2006 ARM Limited. All rights reserved.

Note

If the WCR is located in Normal memory, the Memory Ordering Model allows for the write to happen out-of-order, and possibly to be repeated. In this case there is a possibility that the write to the WCR can trigger a watchpoint on this instruction.

Conversely, a write to the WCR to disable the same watchpoint must trigger the watchpoint if the WCR is located in Device or Strongly Ordered memory. For more information see *Debug events in the debug monitor* on page 2-16.

- An instruction that writes to a Breakpoint Control Register (BCR) or Vector Catch Register (VCR) to enable a Debug event on the Instruction Virtual Address (IVA) of the instruction itself *must not* trigger the Debug event.

  Conversely, a write to the BCR or VCR to disable the same Debug event *must* trigger the Debug event.

Copyright © 2006 ARM Limited. All rights reserved.

2.6 Debug event priority and order

Table 2-4 shows where Debug events come in the ARM exception priority model.

Table 2-4 Position of Debug events in the ARM exception priority model

Priority order	Exception source	Debug event
Highest	Reset	Not applicable
	Precise Data Abort	Watchpoints, precise or imprecise
	FIQ	Not applicable
	IRQ	Not applicable
	Imprecise Abort	Not applicable
	Prefetch abort	Breakpoints, Vector Catch and Halting Debug events
Lowest	Undefined Instruction, SVC and SMC	BKPT Instruction.

Table 2-4 shows that:

- precise and imprecise Watchpoints have the same priority as Precise Data Aborts
- Breakpoints, Vector Catch and Halting Debug events have the same priority as Prefetch abort
- the BKPT Instruction Debug event has the same priority as Undefined Instruction, SVC and SMC exceptions.

Although Breakpoints, Vector Catch and Halting Debug events are shown as lower priority than Watchpoints, the former are canceling Debug events. This means that the instruction is not executed if the Debug event is taken. Therefore, if one of these Debug events is taken on an instruction that, if executed, would trigger a Watchpoint, the Watchpoint is never triggered. The same is true for Prefetch aborts and Data Aborts.

Breakpoint Debug events (IVA or Context ID match), Vector Catch Debug events, and Halting Debug events have the same priority. If more than one of these events occurs on the same instruction, it is UNPREDICTABLE which event is taken.

BKPT Instruction Debug events have a lower priority than all other Debug events.

Debug events must occur in the execution order of the sequential execution model. This means that if an instruction causes a debug event then that event must be taken before any other Debug event, or any other exception, on any instruction that would execute after that instruction in the sequential execution model.

For example, if an Imprecise Watchpoint Debug event is triggered by the first instruction of a code sequence, then a second Debug event that would be triggered by an instruction which, in the sequential execution model, executes after the first instruction is not taken. This is the case even if the second Debug event is precise, for example, a Breakpoint. It is also the case if the second Debug events is also a Watchpoint, that is, both instructions are memory operations, and the memory operations are not strictly ordered by the architecture.

Copyright © 2006 ARM Limited. All rights reserved.

Table 2-5 shows the priority order of Debug events and Data and Prefetch aborts.

Table 2-5 Relative priorities of Debug events and Data and Prefetch aborts

Priority order	Source
Highest	MMU fault
	Precise Watchpoint Debug event
	Precise External Abort
	Imprecise Watchpoint Debug event, Breakpoint Debug event, Vector Catch Debug event, and all Halting Debug events
Lowest	Imprecise External Abort

Note

This table is not complete. It is only intended to show the relative priority of Watchpoints and external aborts. For a complete abort priority table see the ARM Architecture Reference Manual.

If the watchpointed access is subject to an imprecise Data Abort:

- If the imprecise Data Abort can be associated with the watchpointed access, ARM Limited recommends that the watchpoint exception is taken, rather than the Data Abort.
- If an implementation cannot associate the imprecise Data Abort with the watchpointed access, it is IMPLEMENTATION DEFINED whether the implementation will take the imprecise Data Abort or the watchpoint, because the order in which the events are detected by the processor is IMPLEMENTATION DEFINED
- An implementation must ignore the watchpoint if it takes the imprecise Data Abort.

In v6 Debug, all Debug events have a lower priority than Imprecise Data Aborts.

Note

In the *ARM Architecture Reference Manual*, Imprecise Data Aborts are referred to as Imprecise Aborts, when describing the exception.

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 3
Debug Exceptions

This chapter contains the following sections:

- *Overview* on page 3-2
- *Effects of Debug Exceptions on CP15 registers and the WFAR* on page 3-4.

 Copyright © 2006 ARM Limited. All rights reserved.

3.1 Overview

A Debug exception is taken when Monitor Debug-mode is enabled and permitted and a Software Debug event occurs. You must be careful when programming certain events because you might leave the processor in an unrecoverable state. See *unpredictable behavior on Software Debug events* on page 2-14.

If the cause of the Debug exception is a Breakpoint, BKPT Instruction, or a Vector Catch Debug event, the processor performs the following actions:

- It sets the DSCR[5:2] Method of Debug Entry bits according to Table 10-5 on page 10-10.
- It sets the CP15 IFSR and IFAR registers as described in *Effects of Debug Exceptions on CP15 registers and the WFAR* on page 3-4.
- It performs the same sequence of actions as occur in a Prefetch abort exception. This includes:
  - Update SPSR_abt with the saved CPSR.
  - Set R14_abt to the address of the cancelled instruction + 4:

    If the return from the Debug exception handler is made by an instruction that is intended to return from a Prefetch abort exception and retry the cancelled instruction, Table 3-1 shows the instruction that is retried.

Table 3-1 Retry on return from Debug exception

Cause of Debug exception	On return from Debug event, retries:
Breakpoint Debug event	The breakpointed instruction
BKPT Instruction Debug event	The instruction at the address of the `BKPT` instruction
Vector Catch Debug event	The instruction at the vector

  - If the cancelled instruction is within an IT block, save a value in SPSR_abt so that the instruction resumes with the IT bits in the CPSR set as they were before the instruction was cancelled.
  - Update the CPSR to change to abort mode with normal interrupts and imprecise aborts disabled, the J and IT bits cleared to 0, and T bit set to the value of the TE bit in the CP15 control register.
  - If the security extensions are implemented and the processor is in Monitor Mode, clear bit [0] of the Secure Configuration Register (SCR) to 0.
  - Set the PC to the appropriate Prefetch abort vector.
  - Resume execution.

The Prefetch abort handler is responsible for checking the IFSR bits to find out whether the exception entry was caused by a Debug exception or a Prefetch abort exception. If the cause was a Debug exception, it must branch to the debug monitor.

Copyright © 2006 ARM Limited. All rights reserved.

If the cause of the Debug exception was a Watchpoint Debug event, the processor performs the following actions:

- It sets the DSCR[5:2] Method of Debug Entry bits either to Imprecise Watchpoint Occurred or to Precise Watchpoint Occurred.
- It sets the CP15 DFSR, DFAR, and WFAR registers as described in *Effects of Debug Exceptions on CP15 registers and the WFAR* on page 3-4.
- It performs the same sequence of actions as in a Data Abort exception. This includes:
  - Update SPSR_abt with the saved CPSR.
  - Set R14_abt to the address of the cancelled instruction + 8:

    an instruction that is intended to return from a Data Abort exception to retry the aborted instruction returns from a Watchpoint Debug event exception to retry the cancelled instruction.
  - If the instruction returned to is in an IT block, save a value in SPSR_abt so that the IT block resumes correctly.
  - Update the CPSR to change to abort mode and ARM state with normal interrupts and imprecise aborts disabled, the J and IT bits cleared to 0, and T bit set to the value of the TE bit in the CP15 control register.
  - If the security extensions are implemented and the processor is in Monitor Mode, clear bit [0] of the SCR to 0.
  - Set the PC to the appropriate Data Abort vector.
  - Resume execution.

  For more information see *Precise and Imprecise Watchpoint Debug events* on page 2-8.

The Data Abort handler is responsible for checking the DFSR bits to find out whether the exception entry was caused by a Debug exception or a Data Abort exception. If the cause was a Debug exception, it must branch to the debug monitor:

- the address of the instruction that caused the Watchpoint Debug event is in the WFAR
- the address of (instruction to restart at + 8) is in R14_abt.

This is the standard Data Abort behavior.

Halting Debug events never cause a Debug Exception. The Halting Debug events are External Debug Request Debug event, Halt Request Debug event and OS Unlock Catch Debug event.

Copyright © 2006 ARM Limited. All rights reserved.

3.2 Effects of Debug Exceptions on CP15 registers and the WFAR

There are four CP15 registers that are used to record abort information:

DFAR Data Fault Address Register

IFAR Instruction Fault Address Register

IFSR Instruction Fault Status Register

DFSR Data Fault Status Register

Their usage model for normal operation is described in the *ARM Architecture Reference Manual*.

In v6 Debug the *Watchpoint Fault Address Register* (WFAR) exists in CP15. In implementations of v6.1 Debug this register exists in CP14, and the CP15 alias is deprecated.

In ARMv7 the WFAR is one of the debug registers that can be implemented in the Extended CP14 Interface, and is not implemented in CP15. See *Watchpoint Fault Address Register (WFAR)* on page 10-22 for details.

In Monitor Debug-mode the behavior on Breakpoint, BKPT Instruction, or Vector Catch Debug events is as follows:

- the IFSR is updated with the encoding for a Debug event, IFSR[10,3:0] = b00010
- the IFAR is UNPREDICTABLE following these Debug Exceptions
- the DFSR, DFAR and WFAR are unchanged.

In Monitor Debug-mode the behavior on a Watchpoint Debug event is as follows:

- the IFSR and IFAR are unchanged.
- the DFSR is updated with the encoding for a Debug event (DFSR[10,3:0] = b00010).
- the Domain and Write fields in the DFSR (DFSR[11,7:4]) are UNPREDICTABLE. However, an ARMv6 watchpoint sets the Domain field.
- the DFAR is UNPREDICTABLE.
- the WFAR is updated with the Instruction Virtual Address (IVA) of the instruction that accessed the watchpointed address, plus an offset that depends on the processor state:
  - 8 in ARM state
  - 4 in Thumb and ThumbEE states
  - IMPLEMENTATION DEFINED in Jazelle state.

  See *Memory addresses* on page 2-13 for a definition of the IVA used to update the WFAR.

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 4
Debug State

This chapter contains the following sections:

- *Overview* on page 4-2
- *Entering Debug state* on page 4-3
- *Behavior of the PC and CPSR in Debug state* on page 4-7
- *Executing instructions in Debug state* on page 4-9
- *Privilege in Debug state* on page 4-13
- *Behavior of non-invasive debug in Debug state* on page 4-18
- *Exceptions in Debug state* on page 4-19
- *Leaving Debug state* on page 4-21.

 Copyright © 2006 ARM Limited. All rights reserved.

4.1 Overview

If a Debug event occurs when Halting Debug-mode is enabled and permitted, the processor switches to a special state called Debug state and control passes to an external agent.

Note

This external agent is usually a debugger. However it might be some other agent connecting to the debug port of the processor. This could be another processor in the same System on Chip device. In this architecture specification this agent is often referred to as a debugger.

Halting Debug-mode is configured by setting DSCR[14], see *Halting Debug-mode enable, bit [14]* on page 10-15.

Debug state allows the external agent to control the processor following a Debug event. While in Debug state, the processor behaves as follows:

- The PC and CPSR behave as described in *Behavior of the PC and CPSR in Debug state* on page 4-7.
- Instructions are prefetched from the Instruction Transfer Register (ITR), see *Executing instructions in Debug state* on page 4-9.
- The rules about modes and privileges are different to those in normal execution state, see *Privilege in Debug state* on page 4-13.
- Non-invasive debug features are disabled, see *Behavior of non-invasive debug in Debug state* on page 4-18.
- Exceptions are treated as described in *Exceptions in Debug state* on page 4-19. Other software and hardware Debug events and interrupts are ignored.
- If the processor implements a DMA engine, its behavior is IMPLEMENTATION DEFINED.
- If the processor implements a cache or other local memory that it keeps coherent with other memories in the system during normal operation, it must continue to service coherency requests from the other memories.

Leaving Debug state on page 4-21 describes how to leave Debug state.

Copyright © 2006 ARM Limited. All rights reserved.

4.2 Entering Debug state

The processor switches to Debug state if a Debug event occurs when Halting Debug-mode is enabled and permitted.

A processor can also enter Debug state if a Halting Debug event occurs when Halting Debug-mode is not configured. This is the case if debug has been enabled through the external debug interface and is permitted, see Table 2-1 on page 2-2.

On entering Debug state:

1. In ARMv7 only, the **DBGTRIGGER** signal is driven HIGH.
2. The processor is halted, meaning:
   - The instruction pipeline is flushed and no more instructions are prefetched from memory.
   - The PC and CPSR are frozen.
   - The effect of Debug state entry on other core registers is described in *The effect of entering Debug state on core registers*.
   - The effect of Debug state entry on coprocessor registers is described in *The effect of entering Debug state on CP15 registers and the WFAR* on page 4-4.
   - The processor might:
     - — ensure that all memory operations complete
     - — set the DSCR[19] Imprecise Data Aborts discarded bit to 1
     - — drive the **DBGCPUDONE** signal HIGH.

     However, processor behavior regarding memory accesses outstanding at Debug state entry is IMPLEMENTATION DEFINED, see *Imprecise Data Aborts and entry to Debug state* on page 4-4.
3. The processor signals that it has entered Debug state and is ready for an external agent to take control:
   - the DSCR[0] Core Halted bit is set to 1
   - the DSCR[5:2] Method of Debug Entry bits are set according to Table 10-5 on page 10-10
   - the **DBGACK** signal is driven HIGH.

See *EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK* on page 6-4 for more information about the **DBGTRIGGER**, **DBGACK** and **DBGCPUDONE** signals.

4.2.1 The effect of entering Debug state on core registers

All general-purpose and program status registers, including SPSR_abt and R14_abt, are unchanged on entry to Debug state.

Copyright © 2006 ARM Limited. All rights reserved.

4.2.2 The effect of entering Debug state on CP15 registers and the WFAR

On entry to Debug state, the WFAR is updated with the Instruction Virtual Address (IVA) of the instruction which accessed the watchpointed address, plus an offset that depends on the processor state:

- 8 in ARM state
- 4 in Thumb and ThumbEE states
- IMPLEMENTATION DEFINED in Jazelle state.

See *Memory addresses* on page 2-13 for a definition of the IVA used to update the WFAR.

Note

In ARMv6, the WFAR is accessed as a CP15 register.

In ARMv7 the WFAR is accessed as described in Chapter 5 *Debug Register Interfaces*.

In both cases, on Debug state entry the WFAR is set as described in this section.

In ARMv7, all CP15 registers are unchanged on entry to Debug state. In ARMv6, all CP15 register except for the WFAR are unchanged on entry to Debug state. The unchanged registers include the IFSR, DFSR, DFAR, and IFAR.

On processors that implement Security Extensions, bit [0] of the Secure Configuration Register (SCR) is not changed on entry to Debug state.

4.2.3 Imprecise Data Aborts and entry to Debug state

On entry to Debug state, it is IMPLEMENTATION DEFINED whether a processor ensures that all memory operations complete and that all possible outstanding Imprecise Data Aborts have been recognized before it signals to the external agent that it has entered Debug state.

Behavior in ARMv7

In ARMv7 the behavior on entry to Debug state is signalled by the value of the DSCR[19] bit:

If DSCR[19] = 1

The processor has already ensured that all possible outstanding imprecise Data Aborts have been recognized, and the debugger has no additional action to take.

If the processor logic always automatically sets DSCR[19] to 1 on entry to Debug state, then DSCR[19] is implemented as a RO bit.

If DSCR[19] = 0

The following sequence must occur:

1. The debugger must execute an IMPLEMENTATION DEFINED sequence to determine whether all possible outstanding imprecise Data Aborts have been recognized.

Copyright © 2006 ARM Limited. All rights reserved.

Any imprecise Data Abort recognized as a result of this sequence is not taken immediately. Instead, the processor latches the abort event and its type. The imprecise Data Abort is taken when the processor leaves Debug state.

2. DSCR[19] is set to 1.

   There are two ways this requirement can be implemented:

   - The processor automatically sets this bit on detecting the execution of the IMPLEMENTATION DEFINED sequence. In this case, DSCR[19] is implemented as a RO bit.
   - The IMPLEMENTATION DEFINED sequence sets DSCR[19] to 1, using the processor interface to the debug resources. In this case, DSCR[19] is implemented as a RW bit.

While the processor is in Debug state and DSCR[19] is 1, any memory access that triggers an imprecise Data Aborts cause DSCR[7], the sticky imprecise Data Abort flag, to be set to 1, but has no other effect on the state of the processor. The cause and type of the abort are not recorded. Because the abort is not pended, if the imprecise abort is an external imprecise abort and the Interrupt Status Register (ISR) is implemented, bit [8] of the ISR is not updated. Refer to the *ARM Architecture Reference Manual Security Extensions supplement* for details of the ISR.

Any abort that is latched before or during the entry to Debug state sequence, is not overwritten by any new abort. This means it is not discarded if the processor detects another imprecise Data Abort while DSCR[19] is set to 1. The processor acts on the latched abort on exit from Debug state. If the imprecise abort is an external imprecise abort and the ISR is implemented, bit [8] of the ISR reads as 1 indicating that an external abort is pending.

After writes to memory by the debugger, and before exiting Debug state, the debugger must issue an IMPLEMENTATION DEFINED sequence of operations to ensure that any imprecise Data Aborts have been recognized and discarded.

On exit from Debug state, the processor automatically clears DSCR[19] to 0.

If an imprecise Data Abort occurs before entry to Debug state or between entry to Debug state and DSCR[19] transitioning from 0 to 1, then the processor acts on the imprecise Data Abort on exit from Debug state:

- If the A-bit in the CPSR is 1, the abort is pended, and is taken when the A-bit is cleared to 0.
- If the A-bit in the CPSR is 0, the abort is taken by the processor.

The value of DSCR[19] is reflected at the external debug interface by the signal **DBGCPUDONE**. If the processor sets DSCR[19] automatically on entry to Debug state then the **DBGCPUDONE** signal is redundant, because the **DBGACK** signal has the same properties. See *EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK* on page 6-4 for details of the **DBGCPUDONE** signal.

Copyright © 2006 ARM Limited. All rights reserved.

Behavior in ARMv6

The behavior of imprecise Data Aborts on entry to Debug state differs between v6 Debug and v6.1 Debug:

ARMv6 DSCR[19] not defined. A debugger must always issue a Data Synchronization Barrier (DSB) following entry to Debug state.

It is IMPLEMENTATION DEFINED whether DSCR[7] is set to 1 on imprecise Data Aborts that occur when not in Debug state.

ARMv6.1 A debugger must always issue a Data Synchronization Barrier (DSB) following entry to Debug state. This DSB causes DSCR[19] to be set to 1.

DSCR[7] is set to 1 on any imprecise Data Abort detected while the processor is in Debug state, regardless of the setting of DSCR[19].

Copyright © 2006 ARM Limited. All rights reserved.

4.3 Behavior of the PC and CPSR in Debug state

The PC value is frozen on entry to Debug state. A read of R15 after the processor has entered Debug state returns a value that depends on the previous state of the processor and the type of Debug event. Table 4-1 lists the values that can be returned.

Table 4-1 Value of an R15 read after entering Debug state

Debug event	Previous state of processor			Meaning of return address (RA)[a] obtained from R15 read
	ARM	Thumb or ThumbEE	Jazelle[b]	
Breakpoint	RA + 8	RA + 4	RA + Offset	Breakpointed instruction address
Precise Watchpoint	RA + 8	RA + 4	RA + Offset	Address of the instruction that triggered the watchpoint[c]
Imprecise Watchpoint	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume[d]
BKPT instruction	RA + 8	RA + 4	RA + Offset	BKPT instruction address
Vector Catch	RA + 8	RA + 4	RA + Offset	Vector address
External Debug Request	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume
Halt Request	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume
OS Unlock Catch	RA + 8	RA + 4	RA + Offset	Address of the instruction for the execution to resume

a. Return address (RA) is the address of the first instruction that the processor must execute on exit from Debug state. This enables program execution to continue from where it stopped.
b. *Offset* is an IMPLEMENTATION DEFINED constant and documented value.
c. Returning to RA has the effect of retrying the instruction. This may have implications under the memory ordering model. See *Precise and Imprecise Watchpoint Debug events* on page 2-8.
d. RA is not the address of the instruction that triggered the watchpoint, but one that was executed some number of instructions later. The address of the instruction that triggered the watchpoint can be discovered from the value in the WFAR. See *Watchpoint Fault Address Register (WFAR)* on page 10-22.

On entry to Debug state, the value of the CPSR is the value that the instruction at the return address would have been executed with, if it had not been cancelled by the Debug event. That is, it is the value that would be written to the SPSR_irq if the instruction at the return address was interrupted by an IRQ exception.

Copyright © 2006 ARM Limited. All rights reserved.

Note

This rule also applies to the IT bits in the CPSR. On entry to Debug states these bits apply to the instruction at the return address.

The behavior of the PC and CPSR registers in Debug state is as follows:

- The PC does not increment on instruction execution.
- The IT status bits in the CPSR do not change on instruction execution.
- Predictable instructions that explicitly modify the PC or CPSR operate normally, updating the PC or CPSR.
- After the processor has entered Debug state, if R15 is specified as a source operand for an instruction it returns a value as described in Table 4-1 on page 4-7. The value read from R15 is aligned according to the rules of the instruction set indicated by the J and T execution state bits in the CPSR, regardless of the fact that the core only executes the ARM instruction set in Debug state. For more information see *Executing instructions in Debug state* on page 4-9.
- If a sequence for writing a particular value to the PC is executed while in Debug state, and the processor is later forced to restart without any additional write to the PC or CPSR, the execution starts at the address corresponding to the written value.
- If the CPSR is written to while in Debug state, subsequent reads of R15 return an UNPREDICTABLE value, and if the processor is later forced to restart without having performed a write to the PC, the restart address is UNPREDICTABLE. However, the CPSR can be read correctly while in Debug state.

  Note

  In v6 Debug, the CPSR and PC can be written in a single instruction, for example, `MOVS pc,lr`. In this case, the behavior is as if the CPSR is written first, followed by the PC. That is, if the processor is later forced to restart the restart address is predictable. This does not apply to v6.1 Debug or ARMv7 because such instructions are themselves UNPREDICTABLE in Debug state.

- If the processor is forced to restart without having performed a write to the PC, the restart address is UNPREDICTABLE.
- If the PC is written to while in Debug state, later reads of R15 return an UNPREDICTABLE value.

See also *Executing instructions in Debug state* on page 4-9, for more restrictions on instructions that might be executed in Debug state, including those that access the PC and CPSR.

Copyright © 2006 ARM Limited. All rights reserved.

4.4 Executing instructions in Debug state

In Debug State the processor executes instructions issued through the Instruction Transfer Register in the external debug interface, see *Instruction Transfer Register (ITR)* on page 10-37. This mechanism is enabled through DSCR[13] described in *Execute ARM instruction enable, bit [13]* on page 10-14.

The following rules and restrictions apply to instructions that can be executed in this manner in Debug state:

- The processor execution state always corresponds to the state indicated by the J and T execution state bits in the CPSR. However, the processor always interprets the instructions issued through the ITR as ARM instruction set opcodes, regardless of the setting of the J and T execution state bits. The next point gives more information about the significance of the processor execution state.
- With the exception of those instructions listed in Table 4-2 on page 4-10 as UNPREDICTABLE, the processor can execute any ARM state instruction in Debug state, and, with the exception of the value read for R15, the instructions operate as specified for ARM state.

  The value read for R15 is the return address (RA) plus an offset that depends on the previous state of the processor, as shown in Table 4-1 on page 4-7. This state is indicated by the value of the J and T execution bits in the CPSR at the point of entry to Debug state.
- The IT execution state bits in the CPSR are ignored. This means that instructions issued through the ITR do not fail their condition tests unexpectedly. However, the condition code field in an ARM instruction is honored.

  The IT execution state bits in the CPSR are preserved and do not change when instructions are executed, unless an instruction that modifies those bits explicitly is executed.
- The branch instructions `B`, `BL`, `BLX` (1), and `BLX` (2) are UNPREDICTABLE in Debug state.
- The hint instructions `WFI`, `WFE` and `YIELD` are UNPREDICTABLE in Debug state.
- All memory read and memory write instructions with R15 as the base address register read an UNPREDICTABLE value for the base address.
- Certain instructions that normally update the CPSR can be UNPREDICTABLE in Debug state, see *Writing to the CPSR in Debug state* on page 4-10.
- Instructions that load a value from memory into the PC are UNPREDICTABLE in Debug state.
- Conditional instructions that write explicitly to the PC are UNPREDICTABLE in Debug state.
- There are additional restrictions on data processing instructions that write to the PC. See *Data processing instructions with R15 as the target* on page 4-12.
- The exception generating instructions `SVC`, `SMC` and `BKPT` are UNPREDICTABLE in Debug state.

The result of an instruction that is UNPREDICTABLE in Debug state cannot be relied upon. These instructions or results must not represent security holes, such as putting the processor into a state or mode in which debug is not permitted, or changing the state of registers which cannot be accessed from the current state and mode. UNPREDICTABLE instructions must not halt or hang the processor, or any parts of the system.

Copyright © 2006 ARM Limited. All rights reserved.

4.4.1 Writing to the CPSR in Debug state

Table 4-2 lists all the instructions that normally update the CPSR, and gives their behavior in Debug state. The rule for which instructions are allowed in Debug state depends on the version of the debug architecture that is implemented.

Instructions that only update bits [31:27,19:16] of the CPSR, the N, Z, C, V, Q and GE bits, are excluded from this list and have their normal behavior when executed in Debug state. Instructions that cause exceptions, such as `SVC`, `SMC`, and load or store instructions that cause aborts, are also excluded from this list. Their behavior is described in *Exceptions in Debug state* on page 4-19.

In v6.1 Debug and ARMv7, the debugger must use the `MSR` instruction to update the other bits in the CPSR. The `MSR` instruction must write to all fields in the CPSR, `MSR` instructions that only write to certain fields are UNPREDICTABLE. An `MSR` instruction that writes to a SPSR behaves as it does in normal (non-debug) state.

Note

Table 4-2 only governs the behavior of instructions that update the CPSR. See also *Altering CPSR Privileged bits in Debug state* on page 4-13 for information about what values can be written to the CPSR.

Table 4-2 Instructions that modify the CPSR, and their behavior in Debug state

Instruction	v6 Debug	v6.1 Debug, ARMv7
`BX`	UNPREDICTABLE if the J bit in the CPSR is 1. Use for setting/clearing the T bit in the CPSR.	UNPREDICTABLE
`BXJ`	UNPREDICTABLE if either the J or T bits are 1. Use for setting the J bit in the CPSR to 1.	UNPREDICTABLE
`SETEND`	UNPREDICTABLE.	UNPREDICTABLE
`CPS`	UNPREDICTABLE.	UNPREDICTABLE
SPSR to CPSR transfers (Data processing instructions with S bit 1 and R15 as target)	Use for setting the CPSR to any value.	UNPREDICTABLE
Data processing instructions with S bit 0 and R15 as target	Does not update the CPSR in ARMv6.	See *Data processing instructions with R15 as the target* on page 4-12

Copyright © 2006 ARM Limited. All rights reserved.

Table 4-2 Instructions that modify the CPSR, and their behavior in Debug state (continued)

Instruction	v6 Debug	v6.1 Debug, ARMv7
`MSR CPSR_fsxc`	Use for setting the User-writable and Privileged bits in the CPSR.	Use for setting the CPSR to any value
`MSR CPSR_<not fsxc>`	Use for setting the User-writable and Privileged bits in the CPSR.	UNPREDICTABLE
`RFE`	UNPREDICTABLE.	UNPREDICTABLE

In v6.1 Debug and ARMv7, the behavior of the CPSR forms of the `MSR` and `MRS` instructions in Debug state differs from their behavior in normal state.

When not in Debug state:

- Values written to the Execution State bits in the CPSR by an `MSR` instruction are ignored.
- An `MRS` instruction returns the Execution State bits as zero.

Note

In ARM state, the T, J, and IT bits are all always zero.

However, in Debug state:

- Values written to the Execution State bits in the CPSR are not ignored, and the Execution State bits in the CPSR are updated. A direct modification of the Execution State bits in the CPSR by an `MSR` instruction must be followed by an Instruction Synchronization Barrier (ISB).
- An `MRS` instruction returns the correct values of the Execution State bits for the application being debugged.
- Instructions that do not write to all fields of the CPSR are UNPREDICTABLE.
- If an `MRS` instruction reads the CPSR after an `MSR` writes the Execution State bits, and before an Instruction Synchronization Barrier (ISB) operation, the value returned is UNPREDICTABLE.
- If the processor leaves Debug state after an `MSR` writes the Execution State bits, and before an ISB, the behavior of the processor is UNPREDICTABLE.

Copyright © 2006 ARM Limited. All rights reserved.

4.4.2 Data processing instructions with R15 as the target

In ARMv6, data processing instructions with R15 as the target and the S bit set to 0 do not change the CPSR. Therefore, this section is irrelevant to ARMv6.

In ARMv7, when in normal (non-debug) state:

- In ARM state, data processing instructions with R15 as the target and the S instruction bit set to 0 write the 32-bit result as:
  - Bit [0] is written to CPSR[5], the T bit.
  - Bit [1] must be 0. Behavior is UNPREDICTABLE if bit [1] is not zero.
  - Bits [31:2] are written to the PC.
- In Thumb and ThumbEE states, for data processing instructions with R15 as the target and the S instruction bit set to 0:
  - Bit [0] of the result is ignored. However, when in Debug state, bit [0] of the result must be 1.
  - Bits [31:1] are written to the PC.
- In Jazelle state, there is no equivalent instruction. However, Jazelle state requires that a full 32-bit value can be written to the PC.

In Debug state, when the debugger issues such an instruction, the behavior relating to the result of the data processing operation (alu) is as shown in Table 4-3.

Table 4-3 Rules for data processing instructions that write to the PC

CPSR[24], J bit	CPSR[5], T bit	State	Value of alu[1:0]	Operation	Value written to PC		
					[31:2]	[1]	[0]
0	0	ARM	b00	OK	alu[31:2]	0	0
			bx1	UNPREDICTABLE	-	-	-
			b1x	UNPREDICTABLE	-	-	-
X	1	Thumb or ThumbEE	bx0	UNPREDICTABLE	-	-	-
			bx1	OK	alu[31:2]	alu[1]	0
1	0	Jazelle	bxx	OK	alu[31:2]	alu[1]	alu[0]

Data processing instructions with R15 as the target and the S instruction bit set to 1 are always UNPREDICTABLE, as shown in Table 4-2 on page 4-10.

Copyright © 2006 ARM Limited. All rights reserved.

4.5 Privilege in Debug state

On processors that implement Security Extensions, the processor ignores any attempt to execute privileged instructions, other than certain CP14 and CP15 instructions, if all the following conditions are true:

- the processor is in Debug state
- the processor is in Secure User mode
- debug is not allowed in secure privileged modes (either **DBGEN** or **SPIDEN** is LOW).

When the processor ignores an instruction in these circumstances, it sets DSCR[8], the sticky undefined bit.

On processors that do not implement Security Extensions, any privileged instruction can be executed in Debug state at any time.

4.5.1 Accessing registers and memory

The rules for accessing banked registers and memory are the same in Debug state as in normal state. For example, if the CPSR mode bits indicate the processor is in Supervisor Mode:

- reads of the registers return the Supervisor Mode registers
- normal load/store operations make privileged accesses to memory
- the load/store with User Mode privilege operations, for example `LDRT`, make User Mode privilege accesses.

4.5.2 Altering CPSR Privileged bits in Debug state

On processors that implement Security Extensions, the processor:

- prevents attempts to set the CPSR Mode field to a value that would place the processor in a mode or security world where debug is not permitted
- prevents updates to the Privileged bits of the CPSR in cases where debugging is restricted to User Mode applications in the Secure world.

On processors that do not implement Security Extensions, all CPSR updates that are allowed in a privileged mode when not in Debug state, are allowed in Debug state.

Copyright © 2006 ARM Limited. All rights reserved.

Table 4-4 defines the behavior on writes to the CPSR in Debug state. See *Authentication signals* on page 6-3 for details of **SPIDEN** and **DBGEN**.

Table 4-4 Allowed updates to the CPSR in Debug state

Mode	Secure world [a]	DBGEN && SPIDEN	Update privileged CPSR bits [b]	Modify M[4:0] to Monitor Mode
User	Yes	0	Update ignored	UNPREDICTABLE [c]
Privileged	Yes	0	Allowed	Allowed
Any	No	0	Allowed[d]	UNPREDICTABLE [c]
Any	X	1	Allowed[d]	Allowed

a. The processor is in the Secure world if SCR[0] is 0, or the processor is in Monitor Mode.

b. This column does not apply to changing M[4:0] to Monitor Mode. Apart from this, the CPSR bits are defined in the *ARM Architecture Reference Manual*.

c. The definition of UNPREDICTABLE in the *ARM Architecture Reference Manual* precludes this being a security hole. The behavior of the processor must at least prevent this attempt to enter a mode with higher privilege than the modes and states in which debug is permitted.

d. Regardless of the state of **SPIDEN**:
The F bit in the CPSR is read-only and cannot be updated in the Nonsecure world in Debug state if the FW bit in the Secure Control Register (SCR) is set to 1.
The A bit in the CPSR is read-only and cannot be updated in the Nonsecure world in Debug state if the AW bit in the SCR is set to 1.

Note

It is possible to be in Debug state in a secure privileged mode with **SPIDEN** LOW, see *Generation of Debug events* on page 2-20 and *Changing the Authentication signals* on page 6-3. More generally, it is possible to be in Debug state when the current mode, security world or debug authentication signals indicate that, in normal state, Debug events would be ignored. There are two situations where this can occur:

- Between a change in the debug authentication signals and the end of the next Instruction Synchronization Barrier operation, exception entry, or exception return, it is it is UNPREDICTABLE whether the behavior of Debug events that are generated will follow the old or the new authentication signal settings.
- Because it is possible to change the authentication signals while in Debug state.

For example, the following sequence of events can occur:

1. The processor is in a secure privileged mode. **SPIDEN** and **DBGEN** are both set HIGH.
2. An instruction is prefetched that matches all the conditions for a breakpoint to occur.
3. That instruction is committed for execution.
4. At the same time, an external device writes to the peripheral that controls **SPIDEN** and **DBGEN**, causing **SPIDEN** to be deasserted to LOW.

Copyright © 2006 ARM Limited. All rights reserved.

5. **SPIDEN** changes, but the processor is already committed to entering Debug state.

6. The processor enters Debug state and is in a secure privileged mode, even though **SPIDEN** is LOW.

If this series of events occurs, the processor can change to other secure privileged modes (including Monitor) and update privileged bits in the CPSR, because it is in a privileged mode. However, if the processor leaves Secure world or moves to Secure User mode, it cannot return to a Secure privileged mode.

4.5.3 Changing the NS-bit

The NS-bit (NonSecure state bit) is located in the Secure Configuration Register (SCR), bit [0]. In Debug state, the SCR can only be written to:

- in Secure User mode if **DBGEN** and **SPIDEN** are HIGH
- in a secure privileged mode (including Monitor Mode), regardless of the state of **DBGEN** and **SPIDEN**.

This is the case even if invasive debug is allowed everywhere.

A write to the SCR in any other case is treated as an Undefined Instruction exception. See *Exceptions in Debug state* on page 4-19 for details of how undefined exceptions are handled in Debug state.

This is a particular case of the rules for accessing CP15 registers described in *Coprocessor instructions*.

4.5.4 Coprocessor instructions

The processor accesses external coprocessors in a privileged mode as indicated by the CPSR mode bits. Accesses to the internal coprocessors CP14 and CP15 are as follows:

- Instructions that access CP14 or CP15 registers that are permitted (not UNDEFINED) in User Mode when not in Debug state, are always permitted in Debug state.
- Instructions that access CP14 debug registers (`MCR` and `MRC` instructions with opcode_1 set to b000) that are permitted (not UNDEFINED) in privileged modes when not in Debug state are always permitted in Debug state, regardless of debug permissions and the processor mode and security state.
- ARM Limited recommends that certain CP15 instructions required by a debugger to maintain memory coherency are permitted in Debug state regardless of debug permissions and the processor mode, see *Recommended access to specific CP15 registers* on page 7-3.
- Otherwise, if the debugger is permitted to write to the M[4:0] bits of the CPSR to change to a privileged mode, then instructions that access CP14 or CP15 registers that are not permitted (UNDEFINED) in User Mode when not in Debug state are nonetheless permitted in Debug state. There is no requirement to change to a privileged mode first.

Copyright © 2006 ARM Limited. All rights reserved.

Note

A particular case is when the processor does not implement Security Extensions. In this case the M[4:0] bits of the CPSR can always be changed to a privileged mode and, therefore, the debugger is able to access all CP14 and CP15 registers at all times.

- In every case, permissions to access CP15 registers while in Debug state are never greater than the permissions granted to any privileged mode (except Monitor Mode) when in normal state in the current Security world.
- Any attempt to perform accesses that are not permitted is treated as an undefined exception. See *Exceptions in Debug state* on page 4-19 for details of how undefined exceptions are handled in Debug state.
- On processors that implement Security Extensions, accesses to CP15 registers access the CP15 registers of the current world, Secure or Nonsecure. If the debugger requires access to Secure CP15 registers it must change to the Secure world, and if it requires access to the Nonsecure CP15 registers it must change to the Nonsecure world.

This means, for example, that:

- if the processor is stopped in any nonsecure mode (including stopped in Nonsecure User mode), then the processor has access to the Nonsecure world CP15 registers
- if the processor is stopped in a secure privileged mode other than Monitor Mode then the processor has access to the Secure world CP15 registers
- if the processor is stopped in Secure User mode, and debug is permitted in secure privileged modes (**SPIDEN** HIGH), the processor has access to the Secure world CP15 registers
- if the processor is stopped in Monitor Mode, the normal rules for accessing CP15 registers in Monitor Mode apply.

If the **CP15SDISABLE** input to the core is HIGH, any operation affected by **CP15SDISABLE** in normal state results in an Undefined Instruction exception in Debug state.

If the processor is stopped in the Nonsecure world, then the processor is restricted to access only:

- The Nonsecure world CP15 registers.
- Any Secure world CP15 register that are normally accessible in the Nonsecure world.

  Note

  The rules for accessing Secure registers from the Nonsecure world while in normal state also apply in Debug state. For example, some read/write Secure registers might be read-only when read from the Nonsecure world.

- Those common CP15 registers that are configured to be accessible in the Nonsecure world.

Copyright © 2006 ARM Limited. All rights reserved.

If debug is permitted only in the Nonsecure world, the following conditions therefore apply:

- the debugger cannot access Secure CP15 registers
- the debugger cannot write those bits in Secure world registers that are read-only in the Nonsecure world
- the debugger cannot access those common CP15 registers that have been configured not to be accessible in the Nonsecure world.

These rules are summarized in Table 4-5.

Table 4-5 Allowed accesses to CP15 registers and CP14 debug registers in Debug state

Mode	SCR[0]	DBGEN && SPIDEN	Access to CP14 debug registers	Access to CP15 registers and other CP14 registers			
				Normally accessible in all modes	Normally accessible in privileged modes only		
					Banked [a]	Restricted Access [a]	Configurable Access [a]
User	0	0	Allowed	Secure [b]	UNDEFINED [c]	UNDEFINED [c]	UNDEFINED [c]
User	0	1	Allowed	Secure [b]	Secure [b]	Allowed	Allowed
Monitor	0	X	Allowed	Secure [b]	Secure [b]	Allowed	Allowed
Monitor	1	X	Allowed	Nonsecure [b]	Nonsecure [b]	Allowed	Allowed
PxM [d]	0	X	Allowed	Secure [b]	Secure [b]	Allowed	Allowed
User or PxM [d]	1	X	Allowed	Nonsecure [b]	Nonsecure [e]	UNDEFINED [e]	As configured [e]

a. These registers are defined in the *ARM Architecture Reference Manual Security Extensions supplement*.
b. The accesses are allowed and, if banked, return the register corresponding to the indicated security world. In all these cases the debugger can access the SCR register to change to the other security world and access the other banked register. This changes the processor state. After the state change, a different row of this table applies, and it might not be possible to return to the original state.
c. It is impossible for the debugger to access these registers from this state. It does not have permissions to access them directly from User Mode, and it cannot update CPSR M[4:0] field to promote to a privileged mode.
d. PxM means privileged modes, excluding Monitor Mode.
e. If both **DBGEN** and **SPIDEN** are HIGH the debugger can update the CPSR to change to Monitor Mode. This changes the processor state, and a different row in this table then applies. This permits the debugger to access the Secure banked registers, restricted access registers, and configurable access registers configured for no access in the Nonsecure world.

 Copyright © 2006 ARM Limited. All rights reserved.

4.6 Behavior of non-invasive debug in Debug state

If any non-invasive debug features exit, their behavior in Debug state is broadly the same as when non-invasive debug is not permitted. See *Non-invasive debug authentication* on page 8-4 for details.

Note

When the Force Debug Acknowledge bit in the Debug Status and Control Register (DSCR) is set to 1 and the processor is not in Debug state, the behavior of non-invasive debug features is IMPLEMENTATION DEFINED. Non-invasive debug features behave either as if in Debug state or as if in normal state.

 Copyright © 2006 ARM Limited. All rights reserved.

4.7 Exceptions in Debug state

Note

The information in this section applies to v6.1 Debug and ARMv7 only. Refer to the *ARM Architecture Reference Manual* for details of how exceptions are handled in v6 Debug.

Exceptions are handled as follows when the processor is in Debug state:

Reset The processor leaves Debug state.

Prefetch abort

This exception cannot occur because no instructions are prefetched in Debug state.

SVC The `SVC` instruction is UNPREDICTABLE.

SMC The `SMC` instruction is UNPREDICTABLE.

BKPT The `BKPT` instruction is UNPREDICTABLE.

Debug events Debug events are ignored in Debug state.

Interrupts Interrupt request and fast interrupt request exceptions are ignored in Debug state.

However, if the *Interrupt Status Register* (ISR) is implemented, bits [7:6] of the ISR continue to reflect the values of the IRQ and FIQ inputs to the processor. See the *ARM Architecture Reference Manual Security Extensions supplement* for details of the ISR.

Undefined When an Undefined Instruction exception occurs in Debug state, the core behaves as follows:

- PC, CPSR, SPSR_und, R14_und, SCR[0], and DSCR[5:2] are unchanged.
- The processor remains in Debug state.
- DSCR[8], the sticky undefined bit, is set to 1.

See *Sticky Undefined, bit [8]* on page 10-13 for more information.

Precise Data Abort

When a precise Data Abort occurs in Debug state, the core behaves as follows:

- PC, CPSR, SPSR_abt, R14_abt, SCR[0], and DSCR[5:2] are unchanged.
- The processor remains in Debug state.
- DSCR[6], the sticky precise Data Abort bit, is set to 1.
- If the processor is not in Secure User mode, or if debug is permitted in secure privileged modes, DFSR and DFAR are set. Otherwise it is IMPLEMENTATION DEFINED whether DFSR and DFAR are updated.
- If the ISR is implemented, bit [8] of the ISR is not changed, because no abort is pended.

See also *Sticky Precise Abort, bit [6]* on page 10-11.

Copyright © 2006 ARM Limited. All rights reserved.

Imprecise Data Abort

When an imprecise Data Abort occurs in Debug state, the core behaves as follows:

- The setting of the CPSR A-bit is ignored.
- PC, CPSR, SPSR_abt, R14_abt, SCR[0], and DSCR[5:2] are unaltered.
- The processor remains in Debug state.
- The imprecise Data Abort is not taken and DFSR remains unchanged.
- If DSCR[19] is 1:
  - DSCR[7], the sticky imprecise Data Abort bit, is set to 1.
  - This imprecise Data Abort is not acted upon on exit from Debug state.
  - If the ISR is implemented, bit [8] of the ISR is not changed, because no abort is pended.
- if DSCR[19] is 0:
  - DSCR[7] is unchanged.

    Note

    In v6.1 Debug, DSCR[7] is set to 1.

  - This imprecise Data Abort is acted upon on exit from Debug state.
  - If the imprecise Data Abort is an external imprecise abort, and the ISR is implemented, bit [8] of the ISR is set to 1 indicating that an external abort is pending.
- See also:
  - *Imprecise Data Aborts and entry to Debug state* on page 4-4.
  - *Sticky Imprecise Abort, bit [7]* on page 10-12.
  - *Imprecise Data Aborts discarded, bit [19]* on page 10-16.

Copyright © 2006 ARM Limited. All rights reserved.

4.8 Leaving Debug state

The processor leaves Debug state when a Restart Request command is received.

In ARMv7, the Restart Request bit is set to 1 by writing to the Debug Run Control Register (DRCR). See *Restart Request, bit [1]* on page 10-23 for details.

In ARMv6 using the recommended ARM Debug Interface v4.0, the Restart Request command is issued by placing the RESTART instruction in the IR register and taking the Debug Access Port State Machine (DAPSM) through the Run-Test/Idle state.

ARMv7 also supports the **DBGRESTART** and **DBGRESTARTED** signals in the external interface, that can also be used to generate a restart request. This mechanism allows multiple cores to be restarted in synchrony. See *DBGRESTART and DBGRESTARTED* on page 6-6.

In ARMv6, the DRCR register, **DBGRESTART** and **DBGRESTARTED** signals are not part of the recommended external debug interface.

Note

A number of flags in the Debug Status and Control Register (DSCR) must be set to 0 correctly before leaving Debug state. The flags that must be set to 0 are:

- the sticky exception flags, DSCR[8:6]
- the Execute ARM Instruction Enable bit, DSCR[13].

In ARMv7 the sticky exception flags are cleared to 0 by writing 0 to the appropriate bits of the DRCR. This can be combined with the Restart Request. See *Clear Sticky Exceptions, bit [2]* on page 10-24.

In addition, the debugger must not request the processor to leave Debug state until the Latched Instruction Complete flag, InstrCompl_l, DSCR[24], is set to 1.

If the processor is signaled to leave Debug state without all of these flags set to the correct values the results are UNPREDICTABLE.

On receipt of one of these two restart requests, the processor:

1. Clears the Core Restarted flag in the DSCR to 0 and drives the **DBGRESTARTED** signal LOW. See *Core Restarted, bit [1]* on page 10-10 for details.
2. If the request was made using **DBGRESTART**, the core waits for **DBGRESTART** to be driven LOW.
3. Leaves Debug state:
   a. Clears the Core Halted flag in the DSCR. See *Core Halted, bit [0]* on page 10-10 for details.
   b. Drives the **DBGACK**, **DBGTRIGGER**, and **DBGCPUDONE** signals LOW unless the DbgAck bit in the DSCR is set to 1. See *EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK* on page 6-4 and *Force Debug Acknowledge (DbgAck), bit [10]* on page 10-13 for details.
   c. Sets the Core Restarted flag in the DSCR to 1 and drives the **DBGRESTARTED** signal HIGH.

Copyright © 2006 ARM Limited. All rights reserved.

4. Stops ignoring debug events and starts executing instructions from the address held in the PC, and in the mode and state indicated by the current value of the CPSR. The execution state bits of the CPSR are honored, and the IT bits state machine is restarted (with the current value applying to the first instruction restarted).

For more details of the handshake between **DBGRESTART** and **DBGRESTARTED**, see *DBGRESTART and DBGRESTARTED* on page 6-6.

Copyright © 2006 ARM Limited. All rights reserved.

Chapter 5
Debug Register Interfaces

This chapter contains the following sections:

- *About the Debug Register Interface* on page 5-2
- *Reset and Power-down support* on page 5-6
- *Debug Register Map* on page 5-13
- *Synchronization of debug register updates* on page 5-18
- *Access permissions* on page 5-20
- *Coprocessor interface* on page 5-24
- *The Memory-mapped and recommended external debug interfaces* on page 5-34.

Copyright © 2006 ARM Limited. All rights reserved.

5.1 About the Debug Register Interface

The ARMv7 Debug Architecture defines a set of debug registers. There are several possibilities for implementing the Debug Register Interfaces used by software running on the core and an External Debugger to access these registers.

Four Debug Register Interfaces are described by the Debug Architecture:

- A Baseline CP14 Interface. This is implemented by all processors.
- An external debug interface. This is IMPLEMENTATION DEFINED, but must be implemented.

  ARMv6 and ARMv7 each define their own recommended external debug interfaces

  The External Debugger connects to the external debug interface via a *Debug Access Port* (DAP), as shown in Figure 5-1. For more information about the DAP and the recommended external debug interface see the *ARM Debug Interface v5 Architecture Specification*.
- An Extended CP14 Interface. This is required in ARMv6 and is optional in ARMv7.
- A Memory-mapped interface. This is optional in ARMv7.

An ARMv7 implementation must implement at least one of the Extended CP14 or Memory-mapped interfaces. An ARMv7 processor can implement all four interfaces, as shown in Figure 5-1.

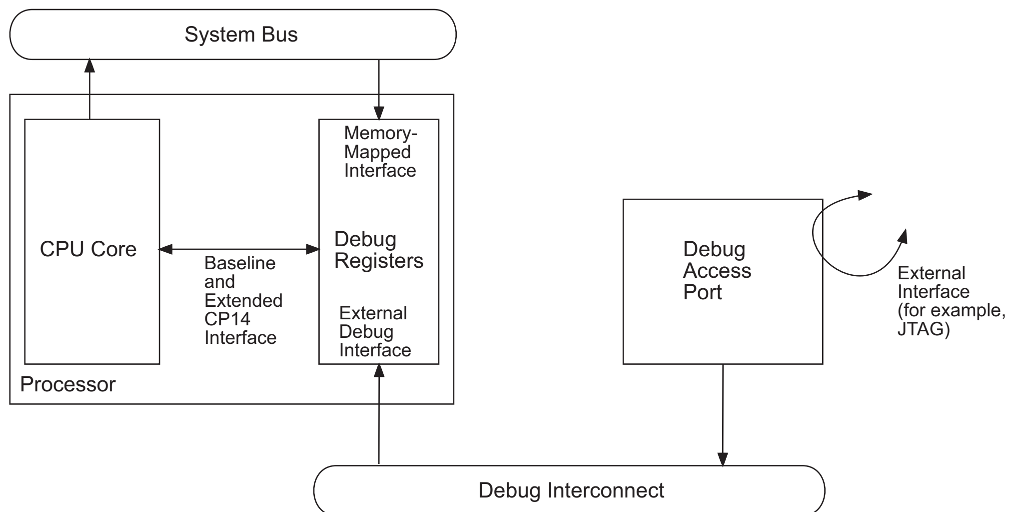


Figure 5-1 ARMv7 system with four Debug Register Interfaces

If a Memory-mapped interface is implemented, the debug registers are accessible through loads or stores to physical memory addresses. ARMv7 does not define how this Memory-mapped interface is implemented at the processor or system level. Some of the debug registers can also be accessed through a coprocessor interface.

Copyright © 2006 ARM Limited. All rights reserved.

One valid ARMv7 configuration is where the ARM processor is not able to access these registers without some level of system support. This means that loads or stores to this register map might go out on the system bus. The system bus might or might not map them back to the same ARM processor on a slave port.

Such a system is shown in Figure 5-2. In this example, the external Debug Access Port has a system access port such that system accesses to debug registers can be multiplexed with debugger accesses to the debug interconnect. In this scenario the external debug interface and Memory-mapped interface might be identical.

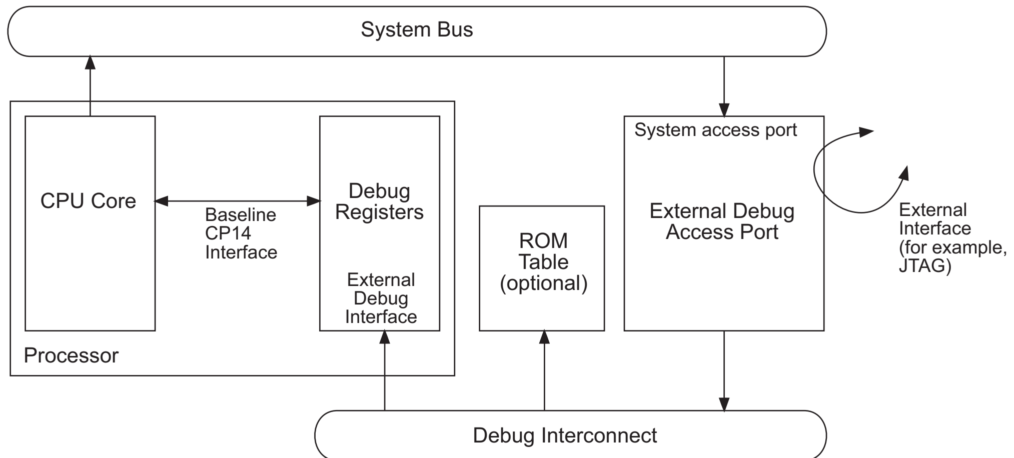


Figure 5-2 ARMv7 system with system-wide debug interconnect

ARMv7 recommends that the combined Memory-mapped and external debug interface for accessing this register map is an APBv3 slave port. Use of a bus standard and system-wide debugging allows this debug subsystem to be extensible and to inter-operate with other debug components, such as the ROM Table shown in Figure 5-2.

Note

The ROM Table holds information about the debug system and its components, see the *ARM Debug Interface v5 Architecture Specification* for more information.

An alternative implementation detects accesses to the debug registers within the processor itself, with the Debug Access Port also contained in the processor. An example of this kind of system is shown in Figure 5-3 on page 5-4:

- The external debug interface consists of:
  - the debug signals to connect the processor to the rest of the system
  - the interface to the External Debugger itself, for example, a JTAG interface.
- The Debug Access Port is part of the external debug interface control logic.

Copyright © 2006 ARM Limited. All rights reserved.

- There is no system-wide access to the debug registers, meaning that devices other than the processor core cannot access these registers directly. However, the registers are Memory-mapped, and the DAP gives access to the debug registers.

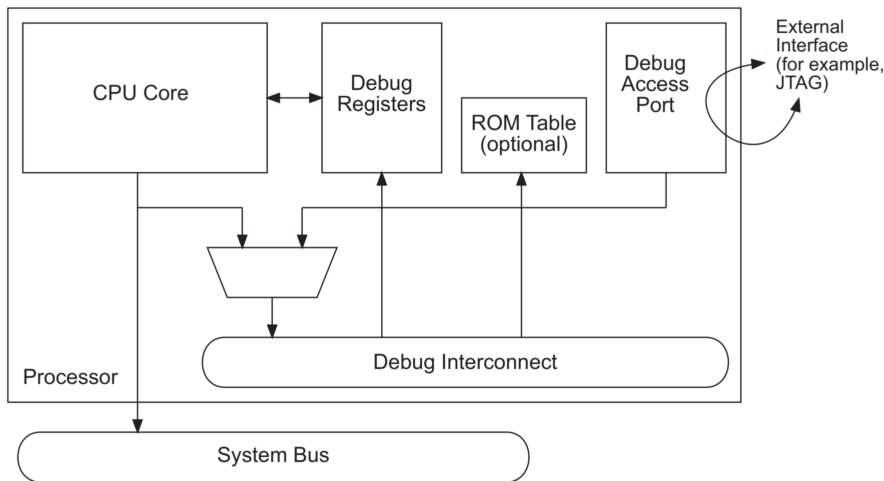


Figure 5-3 ARMv7 system with private debug interconnect

In all ARMv6 implementations and ARMv7 implementations with the Extended CP14 Interface, the debug registers can be accessed through a coprocessor interface in the processor core. In this case the external Debug Access Port has a private interface to the debug registers, but the processor core uses only the CP14 debug interface. Figure 5-4 on page 5-5 shows an ARMv7 implementation of this type, where the Memory-mapped interface is not implemented. The Debug Access Port private interface to the debug registers can be implemented in various ways, including as a bus. As in the previous example, the external debug interface consists of the debug control signals to connect the processor to the rest of the system and the interface to the External Debugger, for example, a JTAG interface. However, in this case the Debug Access Port is part of the external debug interface control logic.

 Copyright © 2006 ARM Limited. All rights reserved.

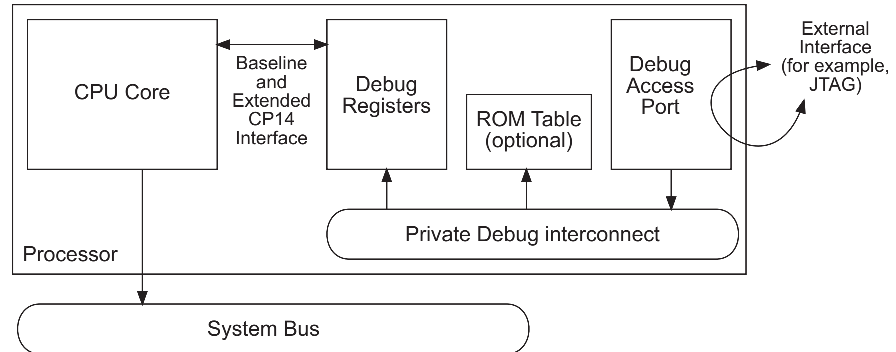


Figure 5-4 ARMv7 system with no Memory-mapped interface

An ARMv7 system can also include the system access port to allow other system components access to the processor's Debug Registers, in which case the structure is more similar to that in Figure 5-2 on page 5-3 above, with a full CP14 interface implemented.

In all cases, the interface to the debug registers by an External Debugger is not defined by the debug architecture. However, ARMv7 recommends a Debug Access Port interface that implements the ARM Debug Interface v5, see the *ARM Debug Interface v5 Architecture Specification*. Although this interface is not required by the architecture, it is required for compatibility with ARM RealView tools.

The ROM Table is required by ADIv5 if more than one debug component is accessed by the Debug Access Port, but might be implemented when there is only one debug component. For more information see the *ARM Debug Interface v5 Architecture Specification*.

 Copyright © 2006 ARM Limited. All rights reserved.

5.2 Reset and Power-down support

This section contains the following subsections:

- *Power domains and debug*
- *Recommended reset scheme* on page 5-12.

5.2.1 Power domains and debug

This section does not apply to ARMv6. ARMv6 only supports a single power domain.

This section discusses how, in ARMv7, certain registers can be split between different power domains to implement support for debug over power-down and re-powering of the processor core.

In ARMv7, it is IMPLEMENTATION DEFINED whether a processor supports debug over power-down. If debug over power-down is not supported, those features associated with debug over power-down are not implemented and a single power domain is usually sufficient. An ARMv7 processor with a single power domain cannot support debug over power-down.

This means that the number of power domains that an ARMv7 processor supports is IMPLEMENTATION DEFINED. However, ARMv7 recommends that at least two are implemented, to provide support for debug over power-down. The two power domains required for this are:

- a debug power domain
- a core power domain.

The debug power domain contains the external debug interface control logic and a subset of the debug resources, determined by physical placement constraints and other considerations, that are explained later in this chapter. Figure 5-5 on page 5-8 shows an example of such a system.

This arrangement is useful for debugging systems where several processors are connected to the same SoC-wide debug bus and where one or more processors can power down at any time.

There are two advantages:

- The debug bus (for example, APBv3 or internal debug bus) is not made unavailable by a processor powering down. If the debugger tries to access the powered-down processor, the external debug interface can return a slave-generated error response instead of locking the system. And if the debugger tries to access another processor, it can proceed normally.

- Some debug registers are unaffected by power-down. This means that a debugger can, for example, identify the SoC while the processor is unavailable.

To have full debug support for power-down and re-powering of the core, the following registers and individual bits need to be in the debug power domain:

ECR This is so the debugger can set the OS Unlock Catch bit to 1 any time and still break on completion of the power-up sequence. If this register was in the core power domain, the power-down event would clear this catch bit to 0. See *OS Unlock Catch, bit [0]* on page 10-57 for details.

Copyright © 2006 ARM Limited. All rights reserved.

DRCR[0] Halt Request bit

This is so the debugger can still request a Debug state entry even if the processor is powered down. Also, if the debugger makes this request before powering-down but it cannot be satisfied, for example because the processor is in Secure state but (**DBGEN** AND **SPIDEN**) = 0, the request remains pending through power-down.

Note

The core needs to be powered up to respond to a pending DRCR[0] halt request or **EDBGRQ**.

OS Save/Restore registers

This is so the lock that the OS sets before saving the debug registers remains set through power-down. See *Operating-system save and restore registers* on page 10-58 for details.

The Device Power-down & Reset registers

These registers have to be in the debug power domain because some of their functions are used for debugging power-down events. See *Device Power-Down and Reset Control Register (PRCR)* on page 10-25, *Device Power-Down and Reset Status Register (PRSR)* on page 10-28.

Lock Access Register

This register has to be in the debug power domain because it is used to enable certain accesses by external debug interface, and this functionality is required when debugging power-down events.

The identification registers and the DIDR

The identification registers are at addresses `0xD00-0xDFC`, and `0xFD0-0xFEC`. See *Management registers* on page 10-69 for details of these registers.

Debugger operation only requires the above registers and bits to be in the debug power domain. However, to rationalize the split between the debug and core power domains in the register map, ARMv7 requires an implementation that supports debug over power-down to have all bits of the following registers in the debug power domain:

DIDR, ECR, and DRCR

No error response is returned on read or write accesses when the core is powered down.

OS Save/Restore registers, and Device Power-down and Reset registers

No error response returned on read or write accesses when the core is powered down. However, accesses to the OS Lock Access Register (OSLAR) and OS Save and Restore Register (OSSRR) are UNPREDICTABLE when the core is powered-down.

All of the management registers, except for the IMPLEMENTATION DEFINED integration registers

The management registers are the registers in the address range `0xD00-0xFFC`. Requiring all these registers to be in the debug power domain simplifies the decoding of register addresses for the registers in the debug power domain.

Copyright © 2006 ARM Limited. All rights reserved.

Note

The CP15 c0 register (`0xD00-0xDFC`) is also included in this category.

Note

ARMv7 requires all bits of a register to be in the same power domain. Therefore, the requirement that DRCR[0] is in the debug power domain means that the DRCR must be implemented in the debug power domain.

It is IMPLEMENTATION DEFINED whether an IMPLEMENTATION DEFINED register is in the core or debug power domain. In this context, the Integration Registers are IMPLEMENTATION DEFINED registers.

All other registers must be in the core power domain.

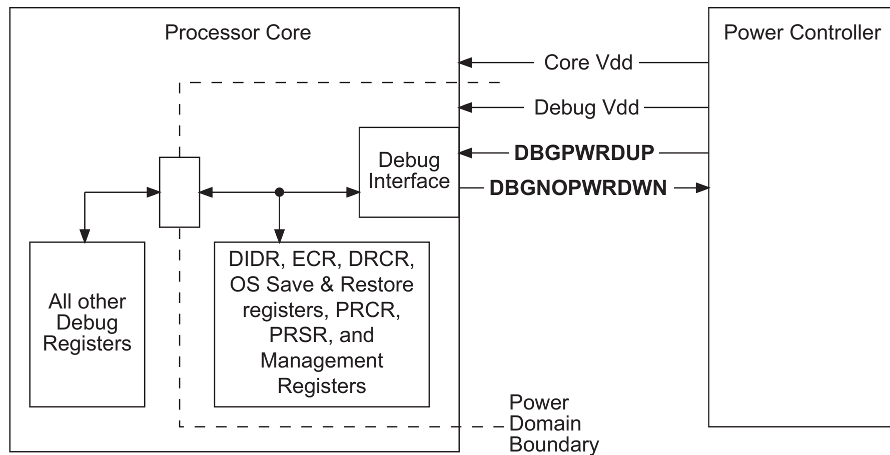


Figure 5-5 Recommendation for Core/Debug power domain split

The signals **DBGNOPWRDWN** and **DBGPWRDUP** shown in Figure 5-5 above form an interface between the debug logic of the core in the debug power domain and the power controller such that:

- the External Debugger can request the Power Controller emulates power-down, simplifying the requirements on software by sacrificing entirely realistic behavior
- the external debug interface knows when the core is powered down, and can communicate this information to the External Debugger.

See *DBGNOPWRDWN* on page 6-7 and *DBGPWRDUP* on page 6-7 for details of these signals,

If the core power domain is not being powered down at the same time as the debug power domain then the authentication signal **DBGEN** must be pulled LOW before power is removed from the debug power domain. The behavior of the debug logic, and in particular the generation of Debug events, is UNPREDICTABLE when the debug power domain is not powered if **DBGEN** is not LOW. Pulling **DBGEN** LOW ensures that Debug events are ignored by the core.

 Copyright © 2006 ARM Limited. All rights reserved.

Reads and writes of debug registers when the debug logic is powered down are UNPREDICTABLE.

The core-based performance monitors must be implemented in the core power domain, and must continue to operate when debug power is removed.

The rest of this document assumes that two power domains are implemented as described in this section. However, the features that are not required for an ARMv7 processor with a single power domain are marked as *SinglePower*, with a description of the differences in behavior.

5.2.2 Operating System Save and Restore support

The Operating System Save and Restore Registers (OSSRR) enable an operating system to save the debug registers before power-down and restore them when power is restored. This extends the support for debug over power-down, and permits debug tools to work at a higher level of abstraction where there are no power-down events.

In ARMv7:

- If an implementation supports debug over power-down, then it must support the OS Save and Restore Registers. If the implementation does not support debug over power-down, these registers are implemented as read-as-zero, write-ignored
- On a *SinglePower* implementation, it is IMPLEMENTATION DEFINED whether the OS Save and Restore Registers are implemented.

In ARMv6, these registers are not defined.

Implementations that support debug over power down do so, in part, by providing support for an operating system to save and restore the debug registers over a power-down. Mechanisms are also provided to allow a debugger to detect that a processor has powered-down. See *Permissions in relation to power-down* on page 5-22 for more information.

The save and restore mechanism is provided by three registers:

- OS Save and Restore Register (OSSRR), see *OS Save and Restore Register (OSSRR)* on page 10-60
- OS Lock Access Register (OSLAR), see *OS Lock Access Register (OSLAR)* on page 10-58
- OS Lock Status Register (OSLSR), see *OS Lock Status Register (OSLSR)* on page 10-59.

You can read the OSLSR to detect whether the OS Save and Restore mechanism is implemented.

The OSSRR works in conjunction with an internal sequence counter, so that a series of reads or writes of this register return or restore the complete debug state of the processor that would be lost when the core is powered down.

The number of accesses required, and the order and interpretation of the data are IMPLEMENTATION DEFINED.

The internal sequence counter is reset to the beginning by writing the key, `0xC5ACCE55`, to the OS Lock Access Register, see *OS Lock Access Register (OSLAR)* on page 10-58. This write also resets the internal sequence counter for the OS save or restore process.

Copyright © 2006 ARM Limited. All rights reserved.

The first access to the OSSRR following the reset of the internal sequence counter must be a read, which returns the number of registers to be saved or restored. The result of issuing a write to the OSSRR following a reset of the internal sequence counter is UNPREDICTABLE.

Note

If the OS Save and Restore mechanism is not implemented, this read will return zero, correctly indicating to software that no registers are to be saved or restored.

The subsequent accesses to the OSSRR must be either all reads or all writes. UNPREDICTABLE behavior results if:

- reads and writes are mixed
- more accesses are performed than the number of registers to be saved or restored, as returned by first read.

If the OS Lock is cleared before the full set of accesses has been completed, the sequence will be restarted the next time the OS Lock is set.

If this register is read or written while the core is powered down or when the OS Lock Access Register is not set to its locked value, the results are UNPREDICTABLE.

The complete debug state of the processor

If the core/debug power domain split described in *Power domains and debug* on page 5-6 is implemented then the complete debug state of the processor consists of:

- The WFAR.
- The BVRs, BCRs, WVRs, WCRs, and VCR.
- the DSCCR and DSMCR.
- If DTRTXfull is set to 1 when the OS save is performed, then the value of DTRTX is guaranteed to be saved and restored. If DTRRXfull is set to 1 when the OS save is performed, then the value of DTRRX is guaranteed to be saved and restored. If either of these flags is not set to 1 when the OS save is performed then the value of the corresponding register is UNPREDICTABLE after the OS restore sequence.

  Note

  The save and restore sequences must not stall reading the values of DTRTX and DTRRX, and must not cause any instructions to be issued to the core, regardless of the settings of the DTR Access Mode bits in the DSCR.

- The DTR status flags themselves:
  - DTRTXfull, bit[29]
  - DTRTXfull_l, bit[26]

Copyright © 2006 ARM Limited. All rights reserved.

- DTRRXfull, bit[30]
- DTRRXfull_l, bit[27].

Note

Unlike a read of EXT-DSCR, when the value of DSCR is saved through OSSRR the values of DTRTXfull_l and DTRRXfull_l are not changed.

- All other writable flags in the DSCR:
  - Method of Debug Entry (MOE), bits[5:2]
  - Force Debug Acknowledge (DbgAck), bit[10]
  - Interrupts Disable (IntDis), bit[11]
  - User Mode Access to Comms Channel Enable, bit[12]
  - Execute ARM Instruction Enable, bit[13]
  - Halting Debug-mode Enable, bit[14]
  - Monitor Debug-mode Enable, bit[15]
  - EXT-DTR Access Mode, bits[21:20].

The save and restore sequence does not preserve:

- Any debug state that is not lost when the core is powered down. For example, if the split between the core and debug power domains described in *Power domains and debug* on page 5-6 is implemented, this includes the ECR, OSLSR, Device Power-down and Reset registers, and all of the management registers.
- The sticky exception flags in the DSCR, and the contents of the ITR.
- The read-only processor status flags in the DSCR:
  - Core halted, bit[0]
  - Core restarted, bit[1]
  - Secure Privileged Debug Disabled, bit[16]
  - Secure Privileged Non-invasive Debug Disabled, bit[17]
  - Nonsecure world status bit, bit[18]
  - Discard imprecise abort, bit[19]
  - Latched instruction complete, bit[24]
  - Sticky pipeline advance, bit[25].
- The core-based performance counters registers. See Chapter 9 *Core-based Performance Counters*.
- The ETM registers.

The restore sequence always overwrites the debug registers with the values that were saved. In particular, the values of the DTRTX and DTRRX registers, and of the DTR status flags DTRTXfull, DTRTXfull_l, DTRRXfull, and DTRRXfull_l after the restore will be the saved values.

If there were valid values in the DTRTX or DTRRX registers immediately before the restore then those values are lost.

 Copyright © 2006 ARM Limited. All rights reserved.

5.2.3 Recommended reset scheme

The processor reset scheme is IMPLEMENTATION DEFINED.

ARMv7 recommends the following four reset signals for an implementation that supports different core and debug power domains:

nSYSPORESET This signal must be driven LOW on power-up of both the core and debug power domains. It sets parts of the processor logic, including debug logic, to a known state.

nCOREPORESET If the core is powered down while the system is still powered up, this signal must be driven LOW when the core is powered back up. It sets parts of the processor logic in the core power domain to a known state. The debug registers that are placed on core power are also initialized by this reset.

nRESET This signal is driven LOW to generate a soft reset, that is, when the system wants to set the processor to a known state but the reset has nothing to do with any power-down, for example a watchdog reset. It sets parts of the non-debug processor logic to a known state. A debug session must be unaffected by this reset.

PRESETDBGn The debugger drives this signal LOW to set parts of the debug logic to a known state. This signal must be driven LOW on power-up of the debug logic.

ARMv6 systems do not support multiple power domains and therefore a less flexible reset scheme is recommended, consisting of only **nSYSPORESET** and **nRESET**. The debug logic is only reset on **nSYSPORESET** and has no independent reset signal.

In the recommended ARMv7 reset scheme, a separate **PRESETDBGn** reset signal can be asserted at any time, not just at power-up. This new signal has similar effects to **nSYSPORESET**, that is, it clears all debug registers, unless otherwise noted by the register definition. See Chapter 6 *Recommended External Debug Interface* for details.

Table 5-1 Recommended reset scheme, ARMv7

	Debug power domain	Core power domain	
Signal	**Debug logic**	**Debug logic**	**Non-debug logic**
nSYSPORESET	Reset	Reset	Reset
nCOREPORESET	Not reset	Reset	Reset
nRESET	Not reset	Not reset	Reset
PRESETDBGn	Reset	Reset	Not reset

For ARMv7 *SinglePower* systems, only **nSYSPORESET**, **nRESET**, and **PRESETDBGn** are recommended.

 Copyright © 2006 ARM Limited. All rights reserved.

5.3 Debug Register Map

The complete list of debug registers is defined in Table 5-2. Full details of each register can be found in the referenced section.

The number of BVR/BCR and WVR/WCR pairs is IMPLEMENTATION DEFINED, see *Number of Breakpoint Register Pairs implemented, bits [27:24]* on page 10-5 and *Number of Watchpoint Register Pairs (WRPs) implemented, bits [31:28]* on page 10-5. An implementation can have up to 16 of each. In Table 5-2, *n* refers to the index of the BVR, BCR, WVR, or WCR register. If *n* is more than the number of breakpoint or watchpoint pairs implemented, the register is Reserved.

The interpretation of the information in the *Access* column depends on the interface used to access the register, coprocessor or Memory-mapped.

Registers 832-1023 are collectively known as the Management Registers.

Table 5-2 Debug register map

Register number	Offset	Access	Versions[a]	Name and reference to description
0	`0x000`	RO	All	*Debug ID Register (DIDR)* on page 10-3.
N/A[b]	-	RO	v7 only	*Debug ROM Address Register (DRAR)* on page 10-5.
N/A[b]	-	RO	v7 only	*Debug Self Address Offset Register (DSAR)* on page 10-6.
1-5	-	RAZ	-	Reserved.
6	`0x018`	RW	v7 [c]	*Watchpoint Fault Address Register (WFAR)* on page 10-22.
7	`0x01C`	RW	All	*Vector Catch Register (VCR)* on page 10-54.
8	-	RAZ	-	Reserved.
9	`0x024`	RW	v7 only	*Event Catch Register (ECR)* on page 10-57.
10	`0x028`	RW	v6.1, v7	*Debug State Cache Control Register (DSCCR)* on page 10-62.
11	`0x02C`	RW	v6.1, v7	*Debug State MMU Control Register (DSMCR)* on page 10-65.
12-31	-	RAZ	-	Reserved.
32	`0x080`	RW	v7 [d]	DTRRX external view[e]. See *Host to Target Data Transfer Register (DTRRX)* on page 10-32.

Copyright © 2006 ARM Limited. All rights reserved.

Table 5-2 Debug register map (continued)

Register number	Offset	Access	Versions[a]	Name and reference to description
33	`0x084`	W	v7 [d]	*Instruction Transfer Register (ITR)* on page 10-37.
		UNP	v7 [d]	*Program Counter Sampling Register (PSCR)* on page 10-31.
34	`0x088`	RW	v7 [d]	DSCR external view[e]. See *Debug Status and Control Register (DSCR)* on page 10-8.
35	`0x08C`	RW	v7 [d]	DTRTX external view[e]. See *Target to Host Data Transfer Register (DTRTX)* on page 10-35.
36	`0x090`	RAZ/WO	v7 only	*Debug Run Control Register (DRCR)* on page 10-23.
37-63	-	RAZ	-	Reserved.
64-79	`0x100-0x13C`	RW/-	All	*Breakpoint Value Registers (BVRn)* on page 10-39 / Reserved.
80-95	`0x140-0x17C`	RW/-	All	*Breakpoint Control Registers (BCRn)* on page 10-40 / Reserved.
96-111	`0x180-0x1BC`	RW/-	All	*Watchpoint Value Registers (WVRn)* on page 10-48 / Reserved.
112-127	`0x1C0-0x1FC`	RW/-	All	*Watchpoint Control Registers (WCRn)* on page 10-49 / Reserved.
128-191	-	RAZ	-	Reserved.
192	`0x300`	RAZ/WO	v7 only	*OS Lock Access Register (OSLAR)* on page 10-58.
193	`0x304`	RO	v7 only	*OS Lock Status Register (OSLSR)* on page 10-59.
194	`0x308`	RW	v7 only	*OS Save and Restore Register (OSSRR)* on page 10-60.
195	-	RAZ	-	Reserved.
196	`0x310`	RW	v7 only	*Device Power-Down and Reset Control Register (PRCR)* on page 10-25.
197	`0x314`	RW	v7 only	*Device Power-Down and Reset Status Register (PRSR)* on page 10-28.
198-511	-	RAZ	-	Reserved.
512-575	`0x800-0x8FC`	-	v7 only	IMPLEMENTATION DEFINED.

 Copyright © 2006 ARM Limited. All rights reserved.

Table 5-2 Debug register map (continued)

Register number	Offset	Access	Versions[a]	Name and reference to description
576-831	-	RAZ	-	Reserved.
832-895	`0xD00-0xDFC`	RO	v7 only	*Processor Identification Registers* on page 10-69.
896-927	-	RAZ	-	Reserved.
928-959	`0xE80-0xEFC`	R/RW	v7 only	IMPLEMENTATION DEFINED Integration registers. See the *CoreSight Architecture Specification*.
960	`0xF00`	RW	v7 only	*Integration Mode Control Register (ITCTRL)* on page 10-70.
961-999	`0xF04- 0xF9C`	-	v7 only	Reserved for Management Registers expansion.
1000	`0xFA0`	RW	v7 only	*Claim Tag Set Register (CLAIMSET)* on page 10-71.
1001	`0xFA4`	RW	v7 only	*Claim Tag Clear Register (CLAIMCLR)* on page 10-71.
1002-1003	-	RAZ	-	Reserved.
1004	`0xFB0`	RAZ/W	v7 only	*Lock Access Register (LAR)* on page 10-72.
1005	`0xFB4`	RO	v7 only	*Lock Status Register (LSR)* on page 10-73.
1006	`0xFB8`	RO	v7 only	*Authentication Status Register (AUTHSTATUS)* on page 10-74.
1007-1009	-	RAZ	-	Reserved.
1010	`0xFC8`	RAZ	v7 only	Device Identifier (DEVID). Reserved.
1011	`0xFCC`	RO	v7 only	*Device Type Register (DEVTYPE)* on page 10-75.
1012-1019	`0xFD0-0xFEC`	RO	v7 only	*Peripheral Identification Registers (PERIPHERALID)* on page 10-76.
1020-1023	`0xFF0-0xFFC`	RO	v7 only	*Component Identification Registers (COMPONENTID)* on page 10-79.

a. An entry of *All* in the *Versions* column indicates that the register is implemented in v6 Debug, v6.1 Debug, and ARMv7.

b. Not applicable. These registers are only implemented through the Baseline CP14 Interface and do not have register numbers or offsets.

c. The method of accessing the WFAR register is different in v6 Debug, v6.1 Debug and ARMv7. See *Watchpoint Fault Address Register (WFAR)* on page 10-22 for details.

d. In ARMv6 these registers are recommended as part of the external debug interface, and are not implemented through the ARMv6 Extended CP14 Interface. In ARMv7 these registers are required.

e. Internal views of the DTRRX, DTRTX, and DSCR are implemented through the Baseline CP14 Interface. This is explained in *Internal and external views of DSCR and DTR* on page 5-16.

Copyright © 2006 ARM Limited. All rights reserved.

5.3.1 Internal and external views of DSCR and DTR

Each of the three registers DSCR, DTRTX and DTRRX have two views denoted by the INT- and EXT- prefixes. The differences between these aliases relate to the handling of the Debug Communications Channel (DCC), and in particular the DTRTXfull and DTRRXfull status flags. The nomenclature *internal* and *external* derives from the intended usage model.

Accesses to INT-DSCR, INT-DTRRX or INT-DTRTX are always made through the Baseline CP14 Interface described in *Baseline CP14 interface* on page 5-24. INT-DSCR is read only in ARMv7

Accesses to EXT-DSCR, EXT-DTRRX or EXT-DTRTX can be made through:

- the Extended CP14 interface, if implemented
- the Memory-mapped interface, if implemented
- the external debug interface.

However, if at any given time you attempt to access the EXT-DSCR, EXT-DTRRX and EXT-DTRTX registers through more than one interface the behavior is UNPREDICTABLE. If an implementation provides a single port to handle external debug interface and the Memory-mapped interface accesses, that port might serialize accesses to the registers from the two interfaces. However, the effects of reads and writes to these registers are such that the behavior observed from either interface will appear as UNPREDICTABLE.

Copyright © 2006 ARM Limited. All rights reserved.

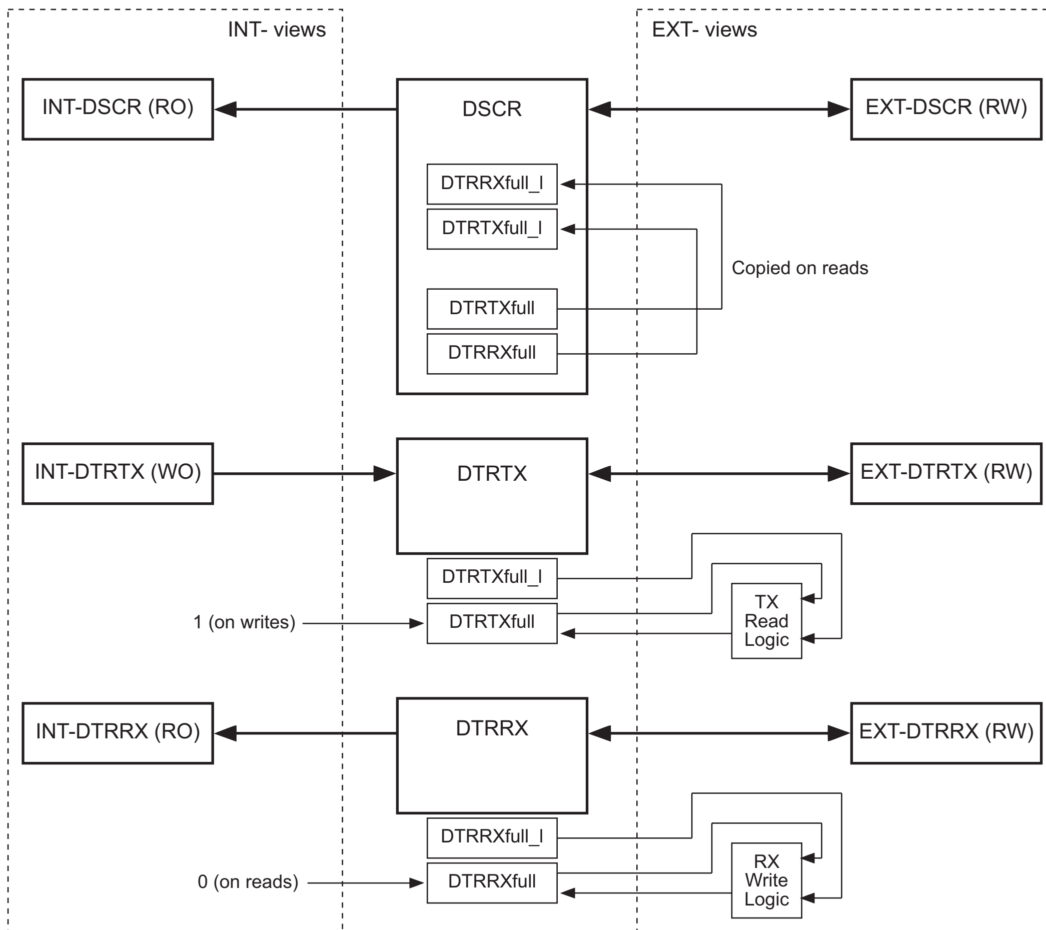


Figure 5-6 Internal (INT-) and External (EXT-) views of DSCR, DTRTX and DTRRX, for ARMv7

Note

INT-DSCR and EXT-DSCR, INT-DTRRX and EXT-DTRRX, and INT-DTRTX and EXT-DTRTX respectively only provide different views onto the same underlying registers, DSCR, DTRRX and DTRTX.

See also *Host to Target Data Transfer Register (DTRRX)* on page 10-32 and *Target to Host Data Transfer Register (DTRTX)* on page 10-35.

5.3.2 Banking of debug registers

On processors that implement Security Extensions, no debug registers are banked between the Secure and Nonsecure worlds.

 Copyright © 2006 ARM Limited. All rights reserved.

5.4 Synchronization of debug register updates

Debug registers can be programmed by software running on the processor through the CP14 coprocessor, and also, optionally in ARMv7, through the Memory-mapped interface.

- Where debug registers are programmed through a CP14 interface they are updated immediately following the CP14 instruction that performs the update.

  Any CP14 register changes caused by instructions that appears in program order after an explicit memory operation are guaranteed not to alter the effect on Debug event generation of that memory operation. For example, a CP14 instruction that enables debug is guaranteed not to cause the previous instruction to trigger a watchpoint.

- Where debug registers are programmed through the Memory-mapped interface, writes to the debug registers from the processor must be followed by a Data Synchronization Barrier (DSB) operation to ensure they have been updated.

  The region of memory occupied by the debug registers must not be marked as Normal, as the Memory Order Model allows accesses to such memory locations that are not appropriate for debug register accesses, for example repeated, speculative, or cached accesses. Reads and writes of debug registers may have side-effects.

  If the region of memory occupied by the debug registers is marked as Device, then writes to the debug registers must normally be preceded by a Data Memory Barrier (DMB) operation to ensure that previous memory accesses complete before the write to the debug register takes place. This is particularly important where the write has effects on the generation of Watchpoint Debug events.

  If the region of memory occupied by the debug registers is marked as Strongly Ordered, the DMB is not required.

  The effect of a debug register write, for example the enabling of a Debug event, is guaranteed to be achieved only from the point when the write access completes. Therefore you must use a DSB operation to ensure that a change in debug configuration has been made.

  All writes to debug registers that are explicitly ordered after any memory operations are guaranteed not to affect those preceding memory operations. When a debug register write is issued after a particular memory access, the debug register write is said to be explicitly ordered after the memory access if either:

  — a DMB or DSB operation is performed between the memory accesses and the debug register writes

  — the debug register write is to Strongly-Ordered memory.

  See the *ARM Architecture Reference Manual* for details of the relative ordering of writes.

See also *Generation of Debug events* on page 2-20.

All writes to debug registers, whether a CP14 write or a memory write to Device or Strongly Ordered memory followed by a DSB, are guaranteed to be visible to instructions and memory operations that appear in program order after the write only after the execution of an Instruction Synchronization Barrier (ISB) operation, the taking of an exception, or the return from an exception.

Copyright © 2006 ARM Limited. All rights reserved.

Note

This applies equally to writes affecting the generation of Watchpoint Debug events as it does those affecting the generation of other Debug events. See *Generation of Debug events* on page 2-20.

The synchronization between register updates made through the external debug interface and updates made by software running on the processor is IMPLEMENTATION DEFINED. However, if the external debug interface is implemented through the same port as the Memory-mapped interface, then such updates have the same properties as updates made through the Memory-mapped interface.

Copyright © 2006 ARM Limited. All rights reserved.

5.5 Access permissions

This section describes the basic concepts governing the access permissions for debug registers on ARMv7 processors. The actual rules for each interface, and for ARMv6 implementations, are given in the section describing the register interface:

- *CP14 debug registers access permissions* on page 5-28
- *Access permissions for External Debug and Memory-mapped interfaces* on page 5-37

The restrictions for accessing the registers can be divided into three categories:

Privilege of the access

Accesses from processors in the system to the memory-mapped registers, and accesses to coprocessor registers, may be required to be privileged.

Locks You can lock out different parts of the register map so they cannot be accessed.

Power-down Access to registers inside the core power domain is not possible when the core is powered down.

When permission to access a register is not granted, an error is returned. The nature of this error depends on the interface:

- For coprocessor interfaces, the error is an Undefined Instruction exception
- For the Memory-mapped interface, the error is a slave-generated error response, for example **PSLVERRDBG**. The error is normally taken as an Abort Exception.
- For the external debug interface, the error is signaled to the debugger by the Debug Access Port.

The behavior of the Memory-mapped or external debug interface is not affected by the core being held in soft reset, whether through the external **nRESET** signal or through the Device Power-down and Reset Control Register (PRCR).

The Hold Internal Reset control bit of the PRCR allows an external debugger to keep the core in soft reset while programming other debug registers. See *Device Power-Down and Reset Control Register (PRCR)* on page 10-25 for details.

5.5.1 Permissions in relation to the privilege of the access

The majority of debug registers can only be accessed by privileged code. The exception to this general requirement is a small subset of the registers, defined in *Baseline CP14 interface* on page 5-24. Using the coprocessor interface, privileged code can disable User Mode access to this subset of registers.

For the Memory-mapped interface, it is IMPLEMENTATION DEFINED whether restricting debug register access to privileged code is implemented by the processor or must be implemented by the system designer at the system level. The behavior of the disallowed access is IMPLEMENTATION DEFINED. It can either be ignored or aborted.

 Copyright © 2006 ARM Limited. All rights reserved.

Note

- The recommended Memory-mapped interface port is based on the AMBA Advanced Peripheral Bus (APB), which does not support signaling of access privileges. Therefore in this case the system must prevent the access.
- This access restriction applies to the privilege of the initiator of the access, not the current mode of the processor being accessed. The privilege of accesses made by a Debug Access Port is IMPLEMENTATION DEFINED.

The system designer can impose additional restrictions. However, ARM Limited strongly discourages restrictions such as only allowing secure privileged accesses, and will not support such restrictions in its tool chain.

5.5.2 Permissions in relation to locks

The registers can be locked by a Debugger or by an operating system so that access to debug registers is restricted.

There are three locks, although some of these locks only apply to certain interfaces:

Software Lock

The Software Lock only applies to accesses made through the Memory-mapped interface.

By default, software is locked out so the debug registers settings are not modified. A debug monitor must leave this lock set when not accessing the debug registers, to reduce the chance of erratic code modifying its settings. When this lock is set, writes to these registers from the Memory-mapped interface are ignored. See *Lock Access Register (LAR)* on page 10-72 and *Lock Status Register (LSR)* on page 10-73 for details on how to set and check this lock.

OS Lock

An OS can set this lock on the debug registers so that the debug registers cannot be read or written while the OS is in the middle of a save or restore sequence. When this lock is set, accesses to some registers return errors. Only the OS Save and Restore Registers can be accessed safely.

Note

An External Debugger can clear this lock at any time, even if an OS save or restore operation is in progress.

See *OS Lock Access Register (OSLAR)* on page 10-58 and *OS Lock Status Register (OSLSR)* on page 10-59 for details on how to set or check the status of this lock.

Debug Software Enable

A debugger can lock out the processor and other potential debug bus masters from the debug bus entirely, guaranteeing that the debug registers cannot be modified by a debug monitor or other software running on the system. The Debug Software Enable is a required function of the Debug Access Port, and is implemented as part of the ARM Debug Interface v5.

Copyright © 2006 ARM Limited. All rights reserved.

ARMv7 processors that implement the Extended CP14 Interface also require a **DBGSWENABLE** input so that the CP14 interface can be locked out. See *DBGSWENABLE* on page 6-8.

Note

The states of the software lock and the OS lock are held in the debug power domain, and the Debug Software Enable is in the Debug Access Port. Therefore, these locks are unaffected by the core powering down. Also, all of these locks are set to their reset values only on reset of the debug power domain, that is. on a **PRESETDBGn** or **nSYSPORESET** reset.

On *SinglePower* systems, these locks are lost over a power-down.

5.5.3 Permissions in relation to power-down

Accesses cannot be made through the coprocessor interface when the core is powered-down.

Access to registers inside the core power domain is not possible when the core is powered down, and accesses return a slave-generated error response such as **PSLVERRDBG**. The Memory-mapped and external debug interfaces ignore accesses to powered-down registers. This means that reads return an UNPREDICTABLE value and writes do not have any effect.

Note

Returning this error response, rather than simply ignoring writes, means that the debugger and the debug monitor detect the debug session interruption as soon as it occurs. This makes re-starting the session, after power-up, considerably easier.

When the processor powers down, the Sticky Power-down bit , bit [1] of the Device Power-down and Reset Status Register, is set to 1. This bit remains set to 1 until the debugger clears it by reading this register after the processor has powered up. If the register is read whilst the processor is still powered down, the bit remains set to 1. When this bit is 1 the Memory-mapped interface behavior is as if the core is powered down, meaning it ignores accesses to registers inside the core power domain and returns a slave-generated error response.

This behavior is useful because when the external debugger tries to access a register whose contents might have been lost by a power-down, it gets the same response regardless of whether the core is currently powered down or has powered back up. This means that, if the external debugger does not access the external debug interface during the window where the core is powered down, the processor still reports the occurrence of the power-down event.

Access to all debug registers is not possible if the debug logic is powered down. In this situation, the system or Debug Access Port (DAP), as applicable, must ignore or abort the accesses.

Accesses through the coprocessor interface are UNPREDICTABLE when debug logic is powered down.

Copyright © 2006 ARM Limited. All rights reserved.

On a *SinglePower* implementations, the system must return an error response to all accesses made through the Memory-mapped or external debug interface while the processor is powered-down. The tables in the section *Permissions summary for SinglePower (debug and core in single power domain)* on page 5-40 summarize the required behavior in this case.

5.5.4 Access to Reserved and IMPLEMENTATION DEFINED locations

It is UNPREDICTABLE whether or not the processor returns an error response if an access is made to a Reserved register, other than a Reserved register in the management registers space (`0xD00-0xFFC`), while any of the following are true:

- the processor is powered-down
- the sticky powered-down flag is set to 1
- either the OS Lock or the Software Lock is set.

Note

- There are no Reserved registers in the Baseline CP14 Interface.
- Reserved registers in the management register space, `0xD00-0xFFC`, always Read-as-zero and ignore writes (WI), even if one or more of the listed conditions applies. Accesses to these registers never return an error response.

The response of the processor to accesses to IMPLEMENTATION DEFINED registers under these conditions is IMPLEMENTATION DEFINED. This also applies to accesses to the integration registers.

When none of these conditions apply, that is during normal operation, reads from Reserved locations return zero and writes to these locations are ignored.

Unused registers in the IMPLEMENTATION DEFINED spaces (`0x800-0x8FC` and `0xE80-0xEFC`) must have the same behavior as that described for Reserved locations.

Note

Unimplemented breakpoint and watchpoint registers are Reserved locations.

Copyright © 2006 ARM Limited. All rights reserved.

5.6 Coprocessor interface

This section contains the following subsections:

- *Baseline CP14 interface*
- *Extended CP14 interface* on page 5-25
- *CP14 debug registers access permissions* on page 5-28.

5.6.1 Baseline CP14 interface

Table 5-3, lists the set of valid CP14 debug instructions for accessing the debug registers.

Additional MRC and MCR instructions with cp_num = b1110 and opcode_1 = b000 not listed below are defined in *Extended CP14 interface* on page 5-25. All other such instructions are Reserved for use by the Debug Architecture. The behavior of Reserved instructions is defined in *CP14 debug registers access permissions* on page 5-28.

All MRC and MCR instructions with cp_num = b1110 and opcode_1 = b001 are used by the trace extension; other values of opcode_1 are not used by the Debug Architecture.

All LDC and STC instructions with cp_num =b1110 that are not listed below are Reserved for use by the Debug Architecture and are currently UNDEFINED. All CDP, MRC2, MCR2, LDC2, STC2, LDCL, STCL, LDC2L, and STC2L instructions with cp_num = b1110 are UNDEFINED.

Instructions that access registers that are only available in ARMv7 are UNDEFINED in earlier versions of the Debug Architecture. For example, MRC p14,0,Rd,c1,c0,0 (read from DRAR) is UNDEFINED in ARMv6, but is allowed in ARMv7.

Rd refers to any of the general purpose registers R0-R14. Use of R15 (PC) is UNPREDICTABLE except where stated.

Table 5-3 Baseline CP14 debug instructions

Instruction	Mnemonic	Version	Name and reference to description
MRC p14,0,Rd,c0,c0,0	DIDR	All	*Debug ID Register (DIDR)* on page 10-3
MRC p14,0,Rd,c1,c0,0	DRAR	v7 only	*Debug ROM Address Register (DRAR)* on page 10-5
MRC p14,0,Rd,c2,c0,0	DSAR	v7 only	*Debug Self Address Offset Register (DSAR)* on page 10-6
MRC p14,0,Rd,c0,c5,0 STC p14,c5,<addr_mode>	INT-DTRRX	All[a]	DTRRX internal view. See *Host to Target Data Transfer Register (DTRRX)* on page 10-32
MCR p14,0,Rd,c0,c5,0 LDC p14,c5,<addr_mode>	INT-DTRTX	All[a]	DTRTX internal view. See *Target to Host Data Transfer Register (DTRTX)* on page 10-35
MRC p14,0,Rd,c0,c1,0 MRC p14,0,PC,c0,c1,0 [b]	INT-DSCR	All[a]	DSCR internal view. See *Debug Status and Control Register (DSCR)* on page 10-8

a. See the register description for more information.
b. DSCR[31:28] are transferred to the CPSR flags. See the *ARM Architecture Reference Manual* for details.

 Copyright © 2006 ARM Limited. All rights reserved.

5.6.2 Extended CP14 interface

In ARMv6 all debug registers can be accessed through CP14, and implementations must provide an external access mechanism for debuggers, the details of which are not covered in the architecture. See *ARMv6 Debug Architecture specifics* on page 5-27.

The Extended CP14 Interface to the debug registers is optional in ARMv7.

The Baseline CP14 Interface is sufficient to boot-strap access to the register file, and allows software to distinguish between the Extended CP14 and Memory-mapped interfaces.

See *ARMv7 Debug Architecture specifics* on page 5-26.

If the Extended CP14 Interface is not implemented, the Memory-mapped interface must be implemented. See section *The Memory-mapped and recommended external debug interfaces* on page 5-34.

This section does not apply if the Extended CP14 Interface is not implemented.

The full list of debug registers is given in Table 5-2 on page 5-13 and is not repeated here.

With some exceptions, listed in *ARMv7 Debug Architecture specifics* on page 5-26 and *ARMv6 Debug Architecture specifics* on page 5-27, the debug registers, including those in the IMPLEMENTATION DEFINED space, are mapped to the following coprocessor instructions, with CRn <= b0111 and the mapping shown in Figure 5-7:

- `MRC p14,0,Rd,CRn,CRm,opcode_2  ; Read`
- `MCR p14,0,Rd,CRn,CRm,opcode_2  ; Write`

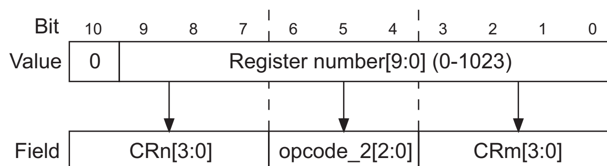


Figure 5-7 Mapping from register number to CP14 instruction

For example, the instruction:

```
MRC p14,0,Rd,c0,c0,5
```

reads the value of BCR0, that is register 80, b000 101 000.

Copyright © 2006 ARM Limited. All rights reserved.

ARMv7 Debug Architecture specifics

Table 5-4 lists the exceptions in the ARMv7 Extended CP14 Interface to this canonical mapping. The Instruction Transfer Register (ITR), PC Sample Register (PCSR) and Lock Access/Status Registers are not for the use of software running on the target. In addition, the Identification Registers in the Management Register space are not canonically mapped as they can be read through CP15.

Table 5-4 Exceptions in the canonical mapping, ARMv7 with Extended CP14 Interface

Register number	Name	Canonical mapping	Note
33	Program Counter Sampling Register	`MRC p14,0,Rd,c0,c1,2`	Returns an UNPREDICTABLE value in Rd.
	Instruction Transfer Register	`MCR p14,0,Rd,c0,c1,2`	Has UNPREDICTABLE behavior
832-895	Processor Identification Registers	`MRC p14,0,Rd,c6,c0,4 to` `MRC p14,0,Rd,c6,c15,7`	Returns an UNPREDICTABLE value in Rd. Use the CP15 identification registers.
1004-1005	Lock Access and Lock Status Registers	`MRC p14,0,Rd,c7,c12,6`	Returns zero to Rd.
		`MRC p14,0,Rd,c7,c13,6`	No lock is required on coprocessor accesses.
		`MCR p14,0,Rd,c7,c12,6`	These instructions are no-ops.
		`MCR p14,0,Rd,c7,c13,6`	No lock is required on coprocessor accesses.

Accesses to EXT-DSCR, EXT-DTRRX and EXT-DTRTX (external views) can be made through the canonical mapping of these registers, in addition to the instructions to access INT-DSCR, INT-DTRRX and INT-DTRTX (internal views) in the Baseline CP14 Interface. See *Internal and external views of DSCR and DTR* on page 5-16.

 Copyright © 2006 ARM Limited. All rights reserved.

ARMv6 Debug Architecture specifics

Table 5-5 lists the exceptions in the ARMv6 Extended CP14 Interface to this canonical mapping. All the instructions listed are UNDEFINED in ARMv6.

Table 5-5 ARMv6 exceptions in the canonical mapping

Register number	Name	Canonical mapping, all UNDEFINED
32	Host to Target Data Transfer Register	`MRC p14,0,Rd,c0,c0,2`
		`MCR p14,0,Rd,c0,c0,2`
33	Program Counter Sampling Register	`MRC p14,0,Rd,c0,c1,2`
	Instruction Transfer Register	`MCR p14,0,Rd,c0,c1,2`
34	Debug Status and Control Register	`MRC p14,0,Rd,c0,c2,2`
		`MCR p14,0,Rd,c0,c2,2`
35	Target to Host Data Transfer Register	`MRC p14,0,Rd,c0,c3,2`
		`MCR p14,0,Rd,c0,c3,2`

See also footnote [d] on Table 5-2 on page 5-13, regarding registers 32, 33, 34 and 35.

No registers in ARMv6 map to CP14 instructions with `CRn` != b0000.

In addition, an instruction for making an internal access write to the DSCR is defined in Table 5-6.

Table 5-6 Additional ARMv6 CP14 debug instruction

Instruction	Mnemonic	Name
`MCR p14,0,Rd,c0,c1,0`	INT-DSCR	*Debug Status and Control Register (DSCR)* on page 10-8

Copyright © 2006 ARM Limited. All rights reserved.

5.6.3 CP14 debug registers access permissions

By default, certain CP14 debug registers can be accessed from User Mode. However, the processor can be programmed to disallow User Mode access to these CP14 registers. See *User Mode access to Comms Channel disable, bit [12]* on page 10-14 for details.

All CP14 debug registers can be accessed if the processor is in Debug state.

Note

When the software lock (LAR) is implemented for a Memory-mapped interface, it does not affect the behavior of CP14 instructions.

Baseline CP14 debug registers access permissions

Access to the baseline CP14 registers is governed only by the processor mode, Debug state and the setting of DSCR[12], as shown in Table 5-7.

Table 5-7 Access to baseline CP14 debug registers

Conditions			Baseline CP14 instructions[a]
Debug state	**Processor mode**	**DSCR[12]** [b]	
Yes	X	X	Proceed
No	User	0	Proceed
No	User	1	UNDEFINED
No	Privileged	X	Proceed

a. Read DIDR, DSAR, DRAR, INT-DSCR, read/write INT-DTR.
b. DCC user accesses disable.

Note

The recommended DbgSwEnable control in the Debug Access Port has no effect on the baseline CP14 instructions. See *DBGSWENABLE* on page 6-8.

See also *Access permissions* on page 5-20 for more information on access permissions and restrictions. Instructions that access the debug registers are UNPREDICTABLE if the debug power domain is powered down.

 Copyright © 2006 ARM Limited. All rights reserved.

ARMv7 CP14 debug registers access permissions, Extended CP14 Interface not implemented

Table 5-8 summarizes the complete set of CP14 instructions if the Extended CP14 Interface is not implemented, that is, only the Baseline CP14 Interface is implemented.

Table 5-8 Access to unallocated CP14 debug registers, ARMv7 with no Extended CP14 Interface

Conditions		Unallocated MCR and MRC instructions with cp_num == b1110 and opcode_1 == b000	
Debug state	**Processor mode**	**CRn <= b0111**	**CRn >= b1000**
Yes	X	UNPREDICTABLE	UNPREDICTABLE
No	User	UNDEFINED	UNDEFINED
No	Privileged	UNPREDICTABLE	UNPREDICTABLE

 Copyright © 2006 ARM Limited. All rights reserved.

ARMv7 CP14 debug registers access permissions, Extended CP14 Interface implemented

If the Extended CP14 Interface is implemented, the **DBGSWENABLE** input can be used to disallow access to registers other than the DIDR, DSCR, DTR, DSAR, DRAR, OSLAR, OSLSR and OSSRR. For more information see *DBGSWENABLE* on page 6-8.

Table 5-9 summarizes the access permissions to the CP14 debug registers, and Table 5-10 on page 5-31gives additional information about access to the Extended CP14 Interface debug registers.

Table 5-9 Access to CP14 debug registers, ARMv7 with Extended CP14 Interface

Conditions				Baseline CP14 instructions[a]	Other CP14 debug instructions[b]	
Debug state	Processor mode	Enable[c]	DSCR[12][d]		CRn <= b0111[e]	CRn >= b1000
Yes	X	0	X	Proceed	UNDEFINED	UNPREDICTABLE
Yes	X	1	X	Proceed	See Table 5-10 on page 5-31	UNPREDICTABLE
No	User	X	0	Proceed	UNDEFINED	UNDEFINED
No	User	X	1	UNDEFINED	UNDEFINED	UNDEFINED
No	Privileged	0	X	Proceed	UNDEFINED[f]	UNPREDICTABLE
No	Privileged	1	X	Proceed	See Table 5-10 on page 5-31	UNPREDICTABLE

a. Read accesses to DIDR, DSAR, DRAR, and INT-DSCR, read/write accesses to INT-DTR.

b. All other `MRC` and `MCR` instructions with cp_num == b1110 and opcode_1 == b000

c. **DBGSWENABLE** signal driven HIGH to enable debug access.

d. DCC User accesses disable

e. Where indicated in this column, see Table 5-10 on page 5-31 for a more detailed description of access permissions to the other registers defined by the debug architecture. In addition, there is more information about access to Reserved and IMPLEMENTATION DEFINED registers in *Access to Reserved and implementation defined locations* on page 5-23.

f. Except for the OS Save and Restore Registers, OSLAR, OSLSR, and OSSRR. These registers are always accessible through the Extended CP14 Interface, regardless of the setting of **DBGSWENABLE**. Access to these registers must always be provided, even on implementations that do not support debug over power down. If the implementation does not support debug over power down these registers are RAZ.

Copyright © 2006 ARM Limited. All rights reserved.

Table 5-10 Access to Extended CP14 Interface debug registers

Conditions		Registers:				
Sticky Power-down set	**OS Lock Set**	**ECR, DRCR, OSLAR[a], OSLSR[a], PRCR, PRSR**	**OSSRR[a]**	**Other Debug[b]**	**All Reserved[c]**	**Other mgmt[d]**
No	No	OK	UNP[e]	OK	OK, RAZ	OK
No	Yes	OK	OK	UNDEFINED	UNP[e]	OK
Yes	X	OK	UNP[e]	UNDEFINED	UNP[e]	OK

a. If the OS Save and Restore Registers are not implemented, these registers addresses behave as Reserved locations.
b. Registers in the range `0x000` to `0x1FC`, except for the ECR, DRCR, the registers defined as baseline registers, and Reserved registers. See Footnote [a] to Table 5-9 on page 5-30 for details of the baseline registers.
c. See also *Access to Reserved and implementation defined locations* on page 5-23.
d. Other management registers. This means registers in the range `0xD00` to `0xFFC`, except for the IMPLEMENTATION DEFINED locations, see *Access to Reserved and implementation defined locations* on page 5-23.
e. UNPREDICTABLE.

The behavior of Extended CP14 Interface `MRC` and `MCR` instructions in ARMv7 also depends on access type of the register, as shown in Table 5-2 on page 5-13. All CP14 debug instructions are defined, that is, all instructions with opcode_1 = b000 and CRn <= b0111. The behavior is summarized in Table 5-11.

Table 5-11 Behavior of CP14 MRC and MCR instructions, ARMv7 with Extended CP14 Interface

Access	MCR p14,0,Rd,CRn,CRm,op2	MRC p14,0,Rd,CRn,CRm,op2
RAZ (Reserved)	Ignored (No-op)	Returns zero in Rd
RO	Ignored (No-op)	Returns register value in Rd
RAZ / WO	Writes value in Rd to register	Returns zero in Rd
RW	Writes value in Rd to register	Returns register value in Rd

For example, the following instruction reads the value of WVR7, register 103, if at least 8 watchpoints are implemented, and is a no-op otherwise:

```
MRC p14,0,Rd,c0,c7,6
```


Copyright © 2006 ARM Limited. All rights reserved.

Note

The access permissions in Table 5-9 on page 5-30 and Table 5-10 on page 5-31 have precedence over the behavior in Table 5-11 on page 5-31. For example, even if at least 8 watchpoints are implemented, the following instruction is UNDEFINED in User Mode, and is UNPREDICTABLE in privileged modes when **DBGSWENABLE** is 0:

```
MRC p14,0,Rd,c0,c7,6
```

ARMv6 Debug CP14 debug registers access permissions

In ARMv6, Debug access to registers other than the DIDR, DSCR and DTR is disallowed if Halting Debug-mode is selected. **DBGSWENABLE**, Sticky power-down and the OS Lock are not implemented, and there are fewer CP14 Debug registers than in the ARMv7 Extended CP14 Interface.

Table 5-12 Access to CP14 debug registers, ARMv6

Conditions				Baseline CP14 instructions[a]	Write INT-DSCR[b]	Other CP14 debug instructions[c]
Debug state	Processor mode	DSCR[15:14][d]	DSCR[12][e]			
Yes	X	XX	X	Proceed	Proceed	Proceed
No	User	XX	0	Proceed	UNDEFINED	UNDEFINED
No	User	XX	1	UNDEFINED	UNDEFINED	UNDEFINED
No	Privileged	b00 (None)	X	Proceed	Proceed	UNDEFINED
No	Privileged	bx1 (Halting)	X	Proceed	Proceed	UNDEFINED
No	Privileged	b10 (Monitor)	X	Proceed	Proceed	Proceed

a. Read accesses to DIDR and INT-DSCR, read/write accesses to INT-DTR.
b. In ARMv6 only certain bits in the DSCR can be written through the coprocessor interface. These are described in the section *Debug Status and Control Register (DSCR)* on page 10-8.
c. All other instructions with opcode_1 == b000 and CRn == b0000. See also Table 5-13 on page 5-33.
d. Debug-mode enabled and selected.
e. DCC User accesses disable

The behavior of CP14 `MRC` and `MCR` instructions in ARMv6 also depends on access type of the register shown in Table 5-2 on page 5-13. The behavior is summarized in Table 5-13 on page 5-33.

 Copyright © 2006 ARM Limited. All rights reserved.

Table 5-13 Behavior of CP14 MRC and MCR instructions in ARMv6

Access	MCR p14,0,Rd,c0,CRm,op2	MRC p14,0,Rd,c0,CRm,op2
RAZ (Reserved)	UNDEFINED	UNDEFINED
RO	UNDEFINED	Returns register value in Rd
RAZ / WO	Writes value in Rd to register	UNDEFINED
RW	Writes value in Rd to register	Returns register value in Rd

For example, the following instruction reads the value of WVR7 (register 103) if at least 8 watchpoints are implemented, and is UNDEFINED otherwise:

```
MRC p14,0,Rd,c0,c7,6
```

Note

The access permissions in Table 5-12 on page 5-32 have precedence over those in Table 5-13. For example, even if at least 8 watchpoints are implemented, the following instruction is UNDEFINED in User Mode, and is also UNDEFINED in privileged modes when Halting Debug-mode is enabled:

```
MRC p14,0,Rd,c0,c7,6
```


Copyright © 2006 ARM Limited. All rights reserved.

5.7 The Memory-mapped and recommended external debug interfaces

The ARMv6 recommended external debug interface is IMPLEMENTATION DEFINED and is not described in this document.

The Memory-mapped interface to the debug registers is optional in ARMv7.

The Baseline CP14 Interface is sufficient to boot-strap access to the register file, and allows software to distinguish between the Extended CP14 and Memory-mapped interfaces.

The Memory-mapped interface is defined in terms of an addressable register file mapped over a region of memory. The recommended external debug interface is also defined in terms of an addressable register file.

This section describes the memory map for both the processor's own view of the debug registers through the Memory-mapped interface, and the memory map of the recommended external debug interface.

If the Memory-mapped interface is not implemented, the Extended CP14 Interface must be implemented. See *Extended CP14 interface* on page 5-25.

5.7.1 Register map

The register map occupies 4KB of physical address space. The base address is IMPLEMENTATION DEFINED and must be aligned to a 4KB boundary.

Each register is mapped at an offset that is the register number multiplied by 4 (the size of a word). For example, WVR7 (register 103) is mapped at offset `0x19C` (412).

The complete list of registers is defined in *Debug register map* on page 5-13, and is not repeated here.

5.7.2 Shared interface port for the Memory-mapped and external debug interfaces

Because the Memory-mapped interface and external debug interface share the same memory map and many of the same properties, it is possible to implement both interfaces using a single memory slave port on the processor.

If the Memory-mapped interface and external debug interface share the same port, External Debugger accesses must be distinguishable from those of software running on a processor, including the ARM processor itself, in the target system. A debug monitor is an example of software that might be running on a processor. For the recommended Memory-mapped or external debug interface this is achieved using the **PADDRDBG**[31] signal, see *PADDRDBG* on page 6-12.

Note

This scheme can permit the External Debugger to make accesses through the combined Memory-mapped or external debug interface port that appear to be Memory-mapped interface accesses. This behavior is permitted by the ARMv7 Debug Architecture.

Through the remainder of this section it is assumed that such a combined port and **PADDRDBG**[31] signal is implemented. However, ARMv7 does not require this arrangement.

Copyright © 2006 ARM Limited. All rights reserved.

5.7.3 Endianness

The recommended Memory-mapped and external debug interface port, referred to as the debug port, is permitted to only allow word accesses. Data presented or returned on the interface is always 32-bits and is in a fixed byte order:

- bits [7:0] of the debug register are mapped to bits [7:0] of the connected data bus
- bits [15:8] of the debug register are mapped to bits [15:8] of the connected data bus
- bits [23:16] of the debug register are mapped to bits [23:16] of the connected data bus
- bits [31:24] of the debug register are mapped to bits [31:24] of the connected data bus.

The debug port ignores bits [1:0] of the address. These signals are not present in the debug port interface.

The Debug Access Port (DAP) and the interface between it and the debug port together form part of the external debug interface, and must support word accesses from the External Debugger to these registers. The recommended ARM Debug Interface v5.0 (ADIv5) supports word accesses. Where this interface is used the implementation must ensure that a 32-bit access by the debugger through the Debug Access Port has the same 32-bit value, in the same bit order, as the corresponding access to the debug registers. This is a requirement for tools support using the ARM Debug Interface v5.0.

If a Memory-mapped interface is implemented, the debug port normally connects to the system interconnect fabric via some form of bridge component. Such system interconnect fabrics normally support byte accesses. The system must support word-sized accesses to the debug registers. The behavior of smaller-than-word-sized accesses to debug registers is UNPREDICTABLE.

The detailed behavior of this bridge and of the system interconnect is outside the scope of the architecture.

Accesses to registers made through the debug port are not affected by the endianness configuration of the processor in which the registers reside. However, they will be affected by the endianness configuration of the bus master making the access, and by the nature and configuration of the fabric that connects the two.

In an ARMv7 processor, the E bit in the CPSR controls the endianness. With some assumptions, described later in this section, the operation of the E bit is:

E bit set to 0, for little-endian operation

If the processor reads its own DIDR with an `LDR` instruction, the system ensures that the value returned in the destination register will be in the same bit order as the DIDR register itself.

E bit set to 1, for big-endian operation

If the processor reads its own DIDR with an `LDR` instruction, the system ensures that:

- bits [7:0] of the DIDR are read into bits [31:24] of the destination register
- bits [15:8] of the DIDR are read into bits [23:16] of the destination register
- bits [23:16] of the DIDR are read into bits [15:8] of the destination register
- bits [31:24] of the DIDR are read into bits [7:0] of the destination register.

Similarly the bytes of a data value written to a debug register, for example the DSCR, are reversed in big-endian configuration.

Copyright © 2006 ARM Limited. All rights reserved.

If an ARMv7 processor, with the E-bit set for little-endian operation, reads the DIDR of a second ARMv7 processor with an `LDR` instruction, then bits [7:0] of the DIDR of the second processor will be read into bits [7:0] of the destination register of the `LDR`, on the first processor. Similarly, the other bytes of the DIDR will be copied to the corresponding bytes of the destination register. However, if the E-bit of the first processor is set for big-endian operation the bytes are reversed during the `LDR` operation, with bits [31:24] of the DIDR of the second processor being read to bits [7:0] of the destination register of the `LDR`.

Note

The ordering of the bytes in the destination register on the first processor is in no way affected by the setting of the E-bit in the CPSR of the second processor.

These examples assume that no additional manipulation of the data occurs in the interconnect fabric of the system. For example, an interconnect might perform byte transposition for accesses made across a boundary between a little-endian subsystem and a big-endian subsystem. Such transformations are beyond the scope of the architecture.

5.7.4 Permission summaries for Memory-mapped and external debug interface

Access permissions for External Debug and Memory-mapped interfaces on page 5-37 describes the restrictions for accessing the Memory-mapped and external debug interface port. This section gives summaries of the permission controls and their effects for different implementations of ARMv7 debug systems. It contains the following sections:

- *Meanings of terms and abbreviations used in this section*
- *Permissions summary for separate debug and core power domains* on page 5-39
- *Permissions summary for SinglePower (debug and core in single power domain)* on page 5-40.

Meanings of terms and abbreviations used in this section

The following terms and abbreviations are used in the tables that summarize the access permissions:

X Don't care. The outcome does not depend on this condition.

0 The condition is false.

1 The condition is true. See Table 5-14 on page 5-37 for more information.

IG/ABT The system or DAP, as applicable, ignores or aborts the access.

Proceed The system or DAP, as applicable, allows the access. However, the processor might return an error response.

NPOSS Accessing the debug registers while the processor is powered down is Not Possible if a single power domain is implemented. The response is system dependent and IMPLEMENTATION DEFINED.

Error Slave-generated error response. Writes ignored and reads return an UNPREDICTABLE value.

OK Access (read or write) succeeds. Writes to RO locations are ignored. Reads from RAZ/WO locations return zero.

UNP The access has UNPREDICTABLE results. Reads return UNPREDICTABLE value.

LAR Lock Access Register, see *Lock Access Register (LAR)* on page 10-72. This is one of the Management registers.

Copyright © 2006 ARM Limited. All rights reserved.

Table 5-14 lists the control conditions used in this section, and tells you where you can find more information about each of these controls. These conditions can be given an argument of X, 0 or 1, as defined at the start of this section. The table gives more information about the meaning when the argument = 1 for each condition.

Table 5-14 Meaning of (Argument = 1) for the control condition

Control condition	Meaning of (Argument = 1)	For details see
DBGPWRDUP	The **DBGPWRDUP** signal is HIGH, indicating the processor is powered up	*DBGPWRDUP* on page 6-7.
Sticky Power-down	PRSR[1] = 1	*Permissions in relation to power-down* on page 5-22.
OS Lock	OSLSR[1] = 1	*Permissions in relation to locks* on page 5-21.
Software Lock	LSR[1] = 1	*Permissions in relation to locks* on page 5-21.
Debug Software Enable	The recommended function of the Debug Access Port (DAP) is enabled.	*Permissions in relation to locks* on page 5-21[a].

a. For more information about the Debug Software Enable function of the Debug Access Port see *ARM Debug Interface v5 Architecture Specification*.

Access permissions for External Debug and Memory-mapped interfaces

Table 5-15 summarizes the access permissions for the External Debug and Memory-mapped interfaces.

At the system level, certain Memory-mapped accesses must be prohibited. An implementation can either ignore or abort these accesses.

Table 5-15 Memory-mapped and external debug interface registers access permissions

Conditions							
Debug logic powered?	Indicated Interface[a]	Debug Software Enable	Access Privilege	Software Lock	DAP or system response	Processor response?	Side-effects?
No	X	X	X	X	IG/ABT	No	-
Yes	Ext. Dbg.	X	X	X	Proceed	Yes[b]	Yes
Yes	Mem-map	0	X	X	IG/ABT	No	-
Yes	Mem-map	X	User	X	IG/ABT	No	-

Copyright © 2006 ARM Limited. All rights reserved.

Table 5-15 Memory-mapped and external debug interface registers access permissions (continued)

Conditions					DAP or system response	Processor response?	Side-effects?
Debug logic powered?	Indicated Interface[a]	Debug Software Enable	Access Privilege	Software Lock			
Yes	Mem-map	1	Privileged	0	Proceed[c]	Yes[b]	Yes
Yes	Mem-map	1	Privileged	1	Proceed[c]	Yes[b]	LAR only[d]

a. In the recommended combined External Debug and Memory-mapped interface port, this indication is given by the **PADDRDBG[31]** signal, which is:
   HIGH to indicate an external debug interface access, indicated by an entry of Ext. Dbg. in the column
   LOW to indicate a Memory-mapped interface access, indicated by an entry of Mem-map. in the column.
   The external debug interface might be able to simulate a Memory-mapped interface access. In this case the access must behave as if Debug Software Enable is set to 1 and the access is privileged.
b. For details of the response by the processor see:
   Permissions summary for separate debug and core power domains on page 5-39
   Permissions summary for SinglePower (debug and core in single power domain) on page 5-40.
c. With a *SinglePower* implementation:
   the access will Proceed if the core is powered up
   if the core is powered down the system response is IMPLEMENTATION DEFINED.
d. Writes are ignored and reads, such as reads of EXT-DSCR, have no side-effects. Writes to LAR have the defined side-effect, if permitted.

When no Memory-mapped interface is implemented, the Software Lock cannot be implemented and the access permissions table simplifies, as shown in Table 5-16.

Table 5-16 External debug interface registers access permissions when Memory-mapped interface not implemented

Conditions				System response	Processor response?	Side-effects?
Debug logic powered?	Indicated Interface[a]	Debug Software Enable[b]	Access Privilege[b]			
No	Ext. Debug	X	X	IG/ABT	No	-
Yes	Ext. Debug	X	X	Proceed	Yes[c]	Yes

a. The indication of the External Debug (Ext. Debug) Interface is implied by the fact that the Memory-mapped interface is not implemented.
b. These control conditions are ignored when the Memory-mapped interface is not implemented.
c. For details of the response by the processor see:
   Permissions summary for separate debug and core power domains on page 5-39
   Permissions summary for SinglePower (debug and core in single power domain) on page 5-40.

Copyright © 2006 ARM Limited. All rights reserved.

Permissions summary for separate debug and core power domains

For implementations with separate Debug and Core power domains, the effects of permissions on access to Memory-mapped debug registers is shown in the following tables:

- Table 5-17 for access to Debug and Management registers
- Table 5-18 for access to the OS Lock and Power-down Control registers.

See Table 5-14 on page 5-37 for more information about the conditions that control access to these registers, and *Meanings of terms and abbreviations used in this section* on page 5-36 for details of the table entries.

Table 5-17 Debug and Management register access for separate debug and core power domains

Conditions			Registers:			
DBGPWRDUP	Sticky Powerdown	OS Lock	DIDR, ECR, DRCR	Other debug[a, c]	Management[b, c]	Reserved[c]
0	X	X	OK	Error	OK	UNP
1	0	0	OK	OK	OK	OK, RAZ/WI
1	0	1	OK	Error	OK	UNP
1	1	X	OK	Error	OK	UNP

a. Registers in the memory region `0x000` - `0x1FC`, except for the DIDR, ECR, and DRCR, and Reserved and IMPLEMENTATION DEFINED locations.

b. Registers in the memory region `0xD00` - `0xFFC`, except for IMPLEMENTATION DEFINED registers, see *Access to Reserved and implementation defined locations* on page 5-23.

c. For details of the behavior of accesses to Reserved and IMPLEMENTATION DEFINED registers see *Access to Reserved and implementation defined locations* on page 5-23.

Table 5-18 OS Lock and Power-down register access for separate debug and core power domains

Conditions			Registers:		
DBGPWRDUP	Sticky Powerdown	OS Lock	OSLSR[a] PRCR, PRSR	OSLAR[a]	OSSRR[a]
0	X	X	OK	UNP	UNP
1	0	0	OK	OK	UNP
1	0	1	OK	OK	OK
1	1	X	OK	OK	UNP

a. If the OS Save and Restore Registers are not implemented, these registers behave as Reserved locations. For details of the behavior of accesses to Reserved and IMPLEMENTATION DEFINED registers see *Access to Reserved and implementation defined locations* on page 5-23.

Copyright © 2006 ARM Limited. All rights reserved.

Permissions summary for *SinglePower* (debug and core in single power domain)

For implementations with a single Debug and Core power domain, when the processor is powered down, it is not possible to access the debug register map, and now error responses can be generated. The Sticky Power-down bit is not present.

The effects of permissions on access to Memory-mapped debug registers is shown in Table 5-19.

See Table 5-14 on page 5-37 for more information about the conditions that control access to these registers, and *Meanings of terms and abbreviations used in this section* on page 5-36 for details of the table entries.

Table 5-19 Register accesses for single Debug and Core power domain

Conditions		Registers:					
DBGPWRDUP	OS Lock	DIDR, ECR, DRCR, OSLSR[a], PRCR, PRSR	OSLAR[a]	OSSRR[a]	Other debug[b, d]	Mgmt[c, d]	Reserved[d]
0	X	NPOSS	NPOSS	NPOSS	NPOSS	NPOSS	NPOSS
1	0	OK	OK	UNP	OK	OK	OK, RAZ/WI
1	1	OK	OK	OK	Error	OK	UNP

a. If the OS Save and Restore Registers are not implemented, these registers behave as Reserved locations, see also footnote [d].

b. Registers in the memory region `0x000` - `0x1FC`, except for the DIDR, ECR, and DRCR, and Reserved and IMPLEMENTATION DEFINED locations.

c. Management Registers, that is, registers in the memory region `0xD00` - `0xFFC`, except for IMPLEMENTATION DEFINED registers.

d. For details of the behavior of accesses to Reserved and IMPLEMENTATION DEFINED registers see *Access to Reserved and implementation defined locations* on page 5-23.

5.7.5 Registers not implemented in the Memory-mapped or external debug interface

The following registers are not implemented in any debug architecture version through the Memory-mapped or external debug interfaces:

DRAR *Debug ROM Address Register (DRAR)* on page 10-5

DSAR *Debug Self Address Offset Register (DSAR)* on page 10-6.

These registers are not required by an external debugger.

In addition, there is no interface to access to INT-DSCR, INT-DTRRX or INT-DTRTX through the Memory-mapped or external debug interface. These operations are only available through the Baseline CP14 Interface.

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 6
Recommended External Debug Interface

This chapter contains the following sections:

- *System integration signals* on page 6-2
- *Recommended debug slave port* on page 6-10.

 Copyright © 2006 ARM Limited. All rights reserved.

6.1 System integration signals

The signals recommended in ARMv7 are shown in Table 6-1.

Table 6-1 Miscellaneous debug signals

Name	Direction	Versions	Description	Section
DBGEN	In	v6, v6.1, v7	Debug Enable	*Authentication signals* on page 6-3
NIDEN	In	v6, v6.1: optional v7: required	Non-invasive Debug Enable	
SPIDEN	In	v6.1, v7	Secure Privileged Invasive Debug Enable	
SPNIDEN	In	v6.1, v7	Secure Privileged Non-Invasive Debug Enable	
EDBGRQ	In	v6, v6.1, v7	External Debug Request	*EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK* on page 6-4
DBGACK	Out	v6, v6.1, v7	Debug Acknowledge signal	
DBGTRIGGER	Out	v7 only, optional	Debug Acknowledge signal	
DBGCPUDONE	Out	v7 only, optional	Debug Acknowledge signal	
DBGRESTART	In	v7 only	External restart request	*DBGRESTART and DBGRESTARTED* on page 6-6
DBGRESTARTED	In	v7 only	Handshake for **DBGRESTART**	
COMMRX	Out	v6, v6.1, v7	DTRRX full signal	*COMMRX and COMMTX* on page 6-6
COMMTX	Out	v6, v6.1, v7	DTRTX empty signal	
DBGOSLOCKINIT	In	v7 only	Initialize O/S Lock on reset	*DBGOSLOCKINIT* on page 6-7
DBGNOPWRDWN	Out	v6, v6.1: optional v7: required	No power-down request signal	*DBGNOPWRDWN* on page 6-7
DBGPWRDUP	In	v7 only	Processor powered up	*DBGPWRDUP* on page 6-7
DBGROMADDR[31:12]	In	v7 only	Debug ROM physical address	*DBGROMADDR and DBGROMADDRV* on page 6-8
DBGROMADDRV	In	v7 only	Debug ROM physical address valid	

Copyright © 2006 ARM Limited. All rights reserved.

Table 6-1 Miscellaneous debug signals (continued)

Name	Direction	Versions	Description	Section
DBGSELFADDR[31:12]	In	v7 only	Debug self-address offset	*DBGSELFADDR and DBGSELFADDRV* on page 6-8
DBGSELFADDRV	In	v7 only	Debug self-address offset valid	
DBGSWENABLE	In	v7 only	Debug software access enable	*DBGSWENABLE* on page 6-8
PRESETDBGn	In	v7 only	Debug logic reset	*PRESETDBGn* on page 6-9

6.1.1 Authentication signals

DBGEN, **NIDEN**, **SPIDEN** and **SPNIDEN** are the authentication signals.

NIDEN and **SPNIDEN** can be omitted if no non-invasive debug features are implemented.

SPIDEN and **SPNIDEN** can be omitted if Security Extensions are not implemented.

When **DBGEN** is LOW (debug disabled), the processor behaves as if DSCR[15:14] equals b00 (see *Debug Status and Control Register (DSCR)* on page 10-8) with the exception that Halting Debug events are ignored when this signal is LOW.

See *Invasive debug authentication* on page 2-4 and *Non-invasive debug authentication* on page 8-4 for details of how these signals control enabling of invasive and non-invasive debug.

Note

The ARMv7 Debug Architecture authentication signal interface described here is compatible with the CoreSight architecture requirements for the authentication interface of a debug component. However the CoreSight architecture places additional requirements on other components in the system, see the *CoreSight Architecture Specification* for more information.

SPIDEN also controls permissions in Debug state. See *Privilege in Debug state* on page 4-13 for details.

See also *Authentication Status Register (AUTHSTATUS)* on page 10-74.

Changing the Authentication signals

In ARMv7, the **NIDEN**, **DBGEN**, **SPIDEN**, and **SPNIDEN** authentication signals can be controlled dynamically, meaning that they might change while the processor is running, or while the processor is in Debug state.

Note

In ARMv6 debut **DBGEN** is a static signal and must be changed only while the processor is in reset.

Copyright © 2006 ARM Limited. All rights reserved.

Normally, these signals are driven by the system, meaning that they are driven by a peripheral connected to the ARM processor. If the software running on the ARM processor has to change any of these signals it must follow this procedure:

1. Execute an implementation specific sequence of instructions to change the signal value. For example, this might be an instruction to write a value to a control register in a system peripheral.
2. If step 1 involves any memory operation, issue a Data Synchronization Barrier (`DSB`).
3. Poll the debug registers for the processor view of the signal values. This is required because the processor might not see the signal change until several cycles after the DSB completes.
4. Issue an Instruction Synchronization Barrier, exception entry or exception return.

The software cannot perform debug or analysis operations that rely on the new value until this procedure has been completed. The same rules apply for instructions executed through the ITR while in Debug state.

The processor view of the **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN** signals can be polled through DSCR[17:16].

Note

Exceptionally, the core might be in Debug state even though the mode, Secure world and authentication signal settings are such that, in normal state, Debug events would be ignored. This can occur because:

- it is UNPREDICTABLE whether the behavior of Debug events that are generated between a change in the authentication signals and the next Instruction Synchronization Barrier, exception entry or exception return follow the behavior of the old or new settings
- it is possible to change the authentication signals while the core is in Debug state.

See also *Generation of Debug events* on page 2-20 and *Altering CPSR Privileged bits in Debug state* on page 4-13.

6.1.2 EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK

EDBGRQ is a request to cause the processor to enter Debug state. If this happens, the DSCR[5:2] Method of Debug Entry bits are set to b0100.

EDBGRQ, **DBGTRIGGER**, **DBGCPUDONE** and **DBGACK** are all active HIGH. Asserting one of these signals means the signal is driven HIGH.

Once **EDBGRQ** is asserted it must be held HIGH until the Debug Acknowledge signal is asserted HIGH.

Before ARMv7, only one Debug Acknowledge signal is recommended, **DBGACK**.

The ARMv7 architecture defines three acknowledge signals, **DBGTRIGGER**, **DBGCPUDONE**, and **DBGACK**. Either **DBGACK** or **DBGTRIGGER** can be used as an acknowledge for **EDBGRQ**.

An ARMv7 implementation can choose not to implement **DBGTRIGGER** or **DBGCPUDONE** if these signals would have identical behavior to **DBGACK**.

 Copyright © 2006 ARM Limited. All rights reserved.

The processor asserts **DBGTRIGGER** to indicate that it is committed to entering Debug state. Therefore, **DBGTRIGGER** can be used as the handshake for the **EDBGRQ** signal. **DBGTRIGGER** should be asserted as early as possible, so that it can be used in the system to signal to other devices that the core is entering Debug state. For example, **DBGTRIGGER** can be used for cross-triggering, meaning it can be used as a halt signal to other processor cores within a multi-processor system when one core halts.

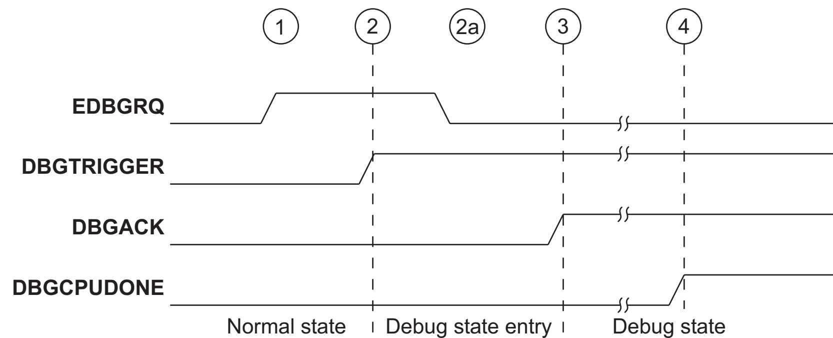
See Chapter 4 *Debug State* for the definition of Debug state.

The processor asserts **DBGACK** to indicate that it has entered Debug state. Therefore, **DBGACK** might be used as a handshake for the **EDBGRQ** signal, instead of using **DBGTRIGGER**. **DBGACK** reflects the state of the status flag DSCR[0].

In ARMv6, **DBGACK** can be used for cross-triggering.

The signal **DBGCPUDONE** is only asserted after the core has completed a data synchronization barrier operation. Therefore, it can be used to guarantee that memory transactions, issued by the core are as a result of operations issued by a debugger, have completed. **DBGCPUDONE** reflects the state of the status flag DSCR[19].

Figure 6-1 shows the external debug request handshaking. It is diagrammatic only, and no timings are implied.



See the text about the ordering of transition 2*a*.

Figure 6-1 External debug request handshake using DBGTRIGGER, DBGACK and DBGCPUDONE

In Figure 6-1these events must occur in order:

1 The peripheral asserts **EBDGRQ** and waits for **DBGTRIGGER** to go HIGH.

2 The core takes the Debug event and starts the Debug state entry sequence. The core drives **DBGTRIGGER** HIGH.

3 The core completes the Debug state entry sequence and drives **DBGACK** HIGH.

4 The core completes a data synchronization barrier operation and drives **DBGCPUDONE** HIGH. This might only be done after intervention by an external debugger, see *Imprecise Data Aborts and entry to Debug state* on page 4-4.

Event 2a, the peripheral driving **EDBGRQ** LOW to deassert the signal, can occur at any time after the peripheral recognizes event 2. Event 2a is not ordered relative to events 3 and 4.

 Copyright © 2006 ARM Limited. All rights reserved.

The **DBGTRIGGER**, **DBGCPUDONE**, and **DBGACK** signals are also driven HIGH when the DbgAck bit in the DSCR is set to 1, see *Force Debug Acknowledge (DbgAck), bit [10]* on page 10-13 for details.

If the DbgAck bit is not set to 1, **DBGTRIGGER**, **DBGCPUDONE**, and **DBGACK** are all driven LOW on exit from Debug state.

6.1.3 DBGRESTART and DBGRESTARTED

Asserting **DBGRESTART** HIGH causes the core to exit from Debug state. Once **DBGRESTART** is asserted, it must be held HIGH until **DBGRESTARTED** is deasserted.

DBGRESTARTED reflects bit [1] of the DSCR. See *Core Restarted, bit [1]* on page 10-10 for details.

DBGRESTART and **DBGRESTARTED** form a four-phase handshake, as shown in Figure 6-2.

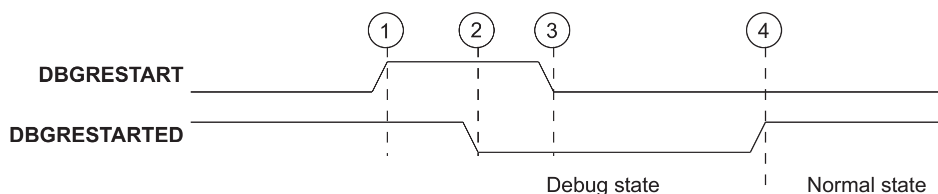


Figure 6-2 DBGRESTART / DBGRESTARTED handshake

Figure 6-2 is diagrammatic only, and no timings are implied. The numbers in Figure 6-2 have the following meanings:

1. If **DBGRESTARTED** is asserted (HIGH) the peripheral asserts **DBGRESTART** HIGH and waits for **DBGRESTARTED** to go LOW
2. The core drives **DBGRESTARTED** LOW to deassert the signal and waits for **DBGRESTART** to go LOW
3. The peripheral drives **DBGRESTART** LOW to deassert the signal.
4. The core leaves Debug state and asserts **DBGRESTARTED** HIGH.

In the process of leaving Debug state the core normally clears the **DBGACK** and **DBGTRIGGER** signals to LOW. It is IMPLEMENTATION DEFINED when this change occurs relative to the changes in **DBGRESTART** and **DBGRESTARTED**.

6.1.4 COMMRX and COMMTX

COMMRX and **COMMTX** reflect the state of DSCR[30:29] through the external debug interface:

- **COMMTX** is the inverse of DSCR[29] (DTRTXfull). Processor is ready to transmit. See *DTRTX register full (DTRTXfull), bit [29]* on page 10-21.
- **COMMRX** is equivalent to DSCR[30] (DTRRXfull). Processor has data to receive. See *DTRRX register full (DTRRXfull), bit [30]* on page 10-22.

 Copyright © 2006 ARM Limited. All rights reserved.

These signals are active HIGH indicators of when the DTR needs processing by the target system. The purpose of these status signals is to allow interrupt-driven communications over the DTR. By connecting these signals to an interrupt controller, software using the debug communications channel can be interrupted whenever there is new data on the channel or when the channel is clear for transmission.

Note

There can be race conditions between reading the DTR flags through a read of EXT-DSCR and reads/writes to the INT-DTR through the Baseline CP14 Interface. However the timing of these signals with respect to the DTR must be such that target code executing off an interrupt triggered off either of these signals must be able to write to INT-DTR and read INT-DTR without race conditions.

6.1.5 DBGOSLOCKINIT

DBGOSLOCKINIT is not required in ARMv6.

DBGOSLOCKINIT is a configuration signal that determines the state of the OS Lock immediately after a debug registers reset. On a debug registers reset:

- if **DBGOSLOCKINIT** is HIGH then the OS Lock is set from the reset
- if **DBGOSLOCKINIT** is LOW then the OS Lock is clear from the reset.

Normally, **DBGOSLOCKINIT** is tied off HIGH.

See *Recommended reset scheme* on page 5-12 for a description of debug registers reset, and *Operating-system save and restore registers* on page 10-58 for details on the OS Lock.

See also *Permissions in relation to locks* on page 5-21.

6.1.6 DBGNOPWRDWN

DBGNOPWRDWN is optional in ARMv6.

DBGNOPWRDWN is equivalent to the value of bit [0] of the Device Power-Down and Reset Control Register. The processor power controller must work in emulate mode when this signal is HIGH.

See *No Power-down (DBGNOPWRDWN), bit [9]* on page 10-13 for details.

6.1.7 DBGPWRDUP

DBGPWRDUP is not required in ARMv6.

DBGPWRDUP is not required in a *SinglePower* system (a single power domain design).

The **DBGPWRDUP** input signal is HIGH when the processor is powered up, and LOW otherwise. The **DBGPWRDUP** signal is reflected in bit [0] of the Device Power-Down and Reset Status Register.

See also *Power-up Status, bit [0]* on page 10-28 and *Permissions in relation to power-down* on page 5-22.

 Copyright © 2006 ARM Limited. All rights reserved.

6.1.8 DBGROMADDR and DBGROMADDRV

DBGROMADDR and **DBGROMADDRV** are not required in ARMv6. They are required in ARMv7 if the Memory-mapped interface is implemented.

DBGROMADDR specifies bits [31:12] of the debug ROM table physical address. This is a configuration input. It must be either a tie-off or change while the processor is in reset. In a system with multiple debug ROM tables, this address must be tied off to the top-level ROM address.

In a system with no debug ROM tables this address must be tied off with the physical address where the debug registers are memory-mapped. Debug software can use the Component Identification Registers at the end of the 4KB block addressed by **DBGROMADDR** to distinguish a ROM table from a processor.

Note

If more than one debug component, for example a processor and an ETM, are implemented in the system, a debug ROM table must be provided.

DBGROMADDRV is the valid signal for **DBGROMADDR**. If the address cannot be determined, **DBGROMADDR** must be tied off to zero and **DBGROMADDRV** tied LOW.

The format of debug ROM tables is defined in *ARM Debug Interface v5 Architecture Specification*.

6.1.9 DBGSELFADDR and DBGSELFADDRV

DBGSELFADDR and **DBGSELFADDRV** are not required in ARMv6.

In ARMv7, **DBGSELFADDR** and **DBGSELFDDRV** are required if the Memory-mapped interface is implemented. If **DBGROMADDR** and **DBGROMADDRV** are not implemented, **DBGSELFADDR** and **DBGSELFADDRV** must not be implemented.

DBGSELFADDR specifies bits [31:12] of the 2's complement signed offset from the debug ROM table physical address to the physical address where the debug registers are Memory-mapped. This is a configuration input. It must be either a tie-off, or change only while the processor is in reset.

If there is no debug ROM table, **DBGROMADDR** must be configured as described in section 8.1.8 above, and **DBGSELFADDR** must be tied off to zero with **DBGSELFADDRV** tied HIGH.

DBGSELFADDRV is the valid signal for **DBGSELFADDR**. If the offset cannot be determined, **DBGSELFADDR** must be tied off to zero and **DBGSELFADDRV** tied LOW.

6.1.10 DBGSWENABLE

In ARMv6, **DBGSWENABLE** is not required.

In ARMv7, **DBGSWENABLE** is required if the Extended CP14 Interface is implemented. **DBGSWENABLE** must be driven by the Debug Access Port. See the *ARM Debug Interface v5 Architecture Specification* for details.

The same control is recommended within the Debug Access Port that locks the system out from the Memory-mapped interface.

 Copyright © 2006 ARM Limited. All rights reserved.

DBGSWENABLE is a signal that can be asserted HIGH by the debugger by some IMPLEMENTATION DEFINED means to block system access to the debug register file. This gives the debugger full control over the debug registers in the processor.

In the ARM Debug Interface v5, **DBGSWENABLE** is asserted HIGH by writing to the DbgSwEnable control bit in the Access Port Control/Status Word register (CSW). See the *ARM Debug Interface v5 Architecture Specification* for details. This signal must normally be asserted HIGH at reset and taken LOW under debugger control.

When this signal is asserted HIGH, Extended CP14 operations become UNDEFINED instructions. See *CP14 debug registers access permissions* on page 5-28.

6.1.11 PRESETDBGn

PRESETDBGn is not required on ARMv6 systems. The debug logic is only reset on system power-up reset.

The reset signal resets all debug registers. See also section *Recommended reset scheme* on page 5-12.

Copyright © 2006 ARM Limited. All rights reserved.

6.2 Recommended debug slave port

This slave port is not required on ARMv6.

The Memory-mapped interface is optional on ARMv7. This section describes the recommended slave port (APBv3). It provides both the Memory-mapped and external debug interfaces.

A valid external debug interface for ARMv7 is any access mechanism that allows the external debugger to complete reads or writes to the Memory-mapped registers described in *The Memory-mapped and recommended external debug interfaces* on page 5-34.

In ARMv7 a Memory-mapped interface can be provided for access to the debug registers using load and store operations. Such an interface is sufficient for the requirements of the external debug interface, and hence it is possible to implement both the Memory-mapped and external debug interfaces using a single memory slave port on the processor.

This section describes the ARMv7 recommendations for a memory slave port (APBv3) as part of the external debug interface. In addition, a Debug Access Port capable of mastering an APBv3 bus and compatible with the ARM Debug Interface v5 (ADIv5) is recommended. Figure 6-3 shows the recommendations.

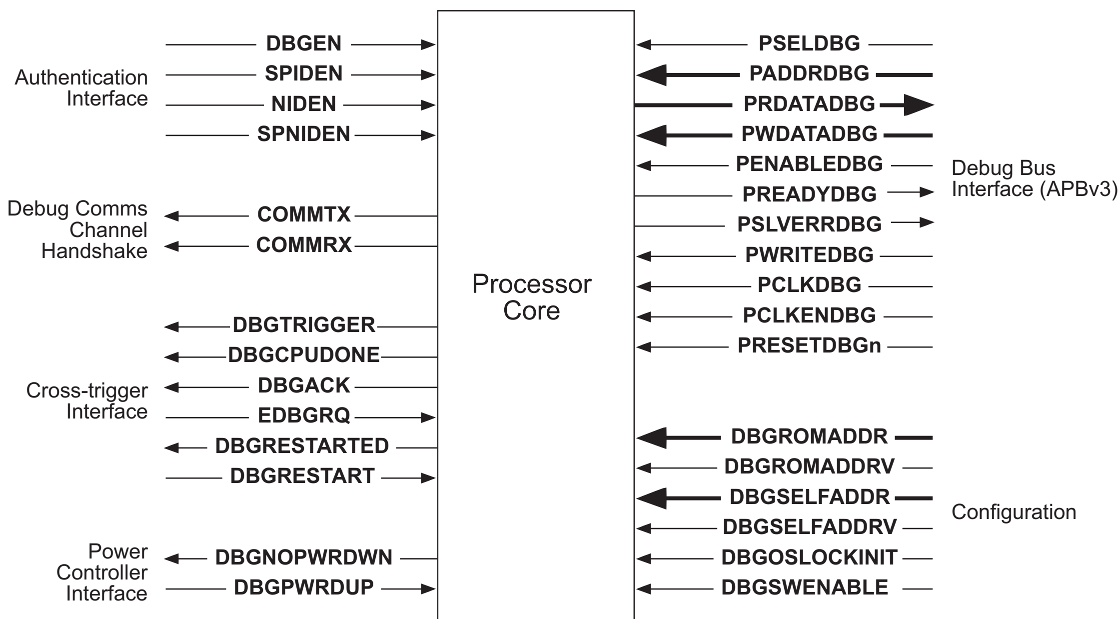


Figure 6-3 Recommended external debug interface (including APBv3 slave port)

In Figure 6-3, signals with a lower-case n suffix are active LOW and all other signals are active HIGH.

Copyright © 2006 ARM Limited. All rights reserved.

ARMv7 recommends that the debug registers are accessible through an ARM AMBA 3 Peripheral Bus version 1 (APBv3) external debug interface. This APBv3 interface:

- is 32-bits wide
- supports only 32-bit reads and writes
- has stallable accesses
- has slave-generated aborts
- has 10 address bits ([11:2]) mapping 4KB of memory.

An extra **PADDRDBG**[31] signal informs the debug slave port about the source of the access.

Table 6-2 describes the external debug interface signals.

Table 6-2 Recommended external debug interface signals

Name	Direction	Description
PSELDBG	In	Selects the external debug interface
PADDRDBG[31,11:2]	In	Address. see *PADDRDBG* on page 6-12
PRDATADBG[31:0]	Out	Read data
PWDATADBG[31:0]	In	Write data
PENABLEDBG	In	Indicates a second and subsequent cycle of a transfer
PREADYDBG	Out	Used to extend a transfer (that is, to insert wait states)
PSLVERRDBG	Out	Slave-generated error response, see *PSLVERRDBG* on page 6-12
PWRITEDBG	In	Distinguishes between a read (LOW) and a write (HIGH)
PCLKDBG	In	Clock
PCLKENDBG	In	Clock enable for **PCLKDBG**

Copyright © 2006 ARM Limited. All rights reserved.

6.2.1 PADDRDBG

PADDRDBG selects the register to read or write.

In the recommended debug slave port that implements both the external debug interface and the Memory-mapped interface, the complete register set is aliased twice. The first view (the Memory-mapped interface view) starts at `0x0`. The second view (the external debug interface view) starts at `0x8000 0000`. That is, **PADDRDBG**[31] is used to distinguish accesses from the External Debugger (set, 1) from accesses from the system as a whole (clear, 0).

Note

Only bits 31 and 11 through 2 of **PADDRDBG** are specified. bits [1:0] are not required because all registers are word-sized. bit [31] is used as above. Because some HDL languages do not allow partial buses to be specified in this way an actual implementation can use a different name for **PADDRDBG**[31], such as **PADDRDBG31**.

6.2.2 PSLVERRDBG

PSLVERRDBG has the same timing as the ready response, **PREADYDBG**. Under the ARMv7 model, accesses are only aborted (return **PSLVERRDBG**) in power-down related scenarios. See also *Access permissions for External Debug and Memory-mapped interfaces* on page 5-37.

 Copyright © 2006 ARM Limited. All rights reserved.

Chapter 7
Debug Requirements on Memory Systems

This chapter contains the following sections:

- *About debug requirements on memory systems* on page 7-2
- *Recommended access to specific CP15 registers* on page 7-3
- *Debug state Cache/MMU Control Registers* on page 7-4.

 Copyright © 2006 ARM Limited. All rights reserved.

7.1 About debug requirements on memory systems

The Debug Architecture places requirements on the memory system. There are two general guidelines:

- Memory coherency has to be maintained during debugging.
- It is best if debugging is non-intrusive. This requires a way to preserve, for example, the contents of memory caches and translation lookaside buffers (TLBs), so the state of the target application is not altered.

In Debug state, it is strongly recommended that the caches and TLBs, where implemented, behave as follows. For preservation purposes it is strongly recommended that it be possible to:

- disable cache evictions and line fills, so that cache accesses (read or write) do not cause the contents of caches do not change.
- disable TLB evictions and replacements, so that translations do not cause the contents of TLBs to change.

These facilities must be accessible by the external debugger, but are only required when in Debug state. In ARMv7, the Debug State Cache Control Register (DSCCR) and the Debug State MMU Control Register (DSMCR) are used for this purpose.

However, in Debug state, no instruction fetches occur, and therefore in systems that implement separate instruction and data caches, and/or separate instruction and data TLBs, there can be no instruction cache or instruction TLB evictions or replacements while in Debug state.

In Debug state, caches must behave as follows, for memory coherency purposes:

- Reads behave as in normal operation (cache reads return data from the cache, cache misses fetch from external memory).
- Writes update all levels of memory. That is, all caches and all external memory where the address written to exists.

It must be possible to reset the processor's memory system to a known safe and coherent state. Similarly, it must be possible to reset any caches of meta-information (such as branch predictor arrays) to a safe and coherent state.

It is also recommended, for debugging purposes, that TLBs can be disabled such that all TLB accesses are read from the main translation tables, and not from the TLB. This allows a debugger to access memory without using the virtual to physical memory mapping of the application (where implemented).

Copyright © 2006 ARM Limited. All rights reserved.

7.2 Recommended access to specific CP15 registers

Mechanisms must exist to allow these requirements to be met when debugging in Secure User mode when debug of secure privileged modes is not permitted. In such a case, access to the CP15 cache and TLB control registers would normally be prohibited.

To allow these requirements to be met, it is recommended that the rules for accessing CP15 registers on processors that implement Security Extensions do not apply for certain register access operations when the processor is in Debug state. See *Coprocessor instructions* on page 4-15 for more information.

The set of instructions depends on the version of the ARM Architecture implemented, and is listed in Table 7-1.

Table 7-1 CP15 operations allowed from User Mode in Debug state

Versions [a]	Operation	Description
v7	`MCR p15,0,Rd,c7,c5,0`	Invalidate All I-Cache and flush Branch Predictor arrays
v7	`MCR p15,0,Rd,c7,c5,1`	Invalidate I-Cache by MVA
v6Z	`MCRR p15,0,Rd,Rn,c5`	Invalidate I-cache by VA range
v6Z, v7	`MCR p15,0,Rd,c7,c5,6`	Flush entire Branch Predictor array

a. These are ARM architecture versions, not the Debug architecture versions.

These instructions must be executable in Debug state regardless of any processor setting. However, they can generate aborts if I-cache lockdown is in use.

In ARMv6 and on other devices that do not implement Security Extensions, or when debugging in a state where privileged CP15 operations can be executed, the debugger can use any CP15 operations. These include, but are not limited to, those operations listed in Table 7-2.

Table 7-2 Privileged CP15 operations used to maintain instruction/data coherency

Operation	Description
`MCR p15,0,Rd,c7,c5,0`	Invalidate All I-Cache and flush Branch Predictor arrays
`MCR p15,0,Rd,c7,c5,1`	Invalidate I-Cache by MVA
`MCR p15,0,Rd,c7,c5,6`	Flush entire Branch Predictor array.
`MCR p15,0,Rd,c7,c5,7`	Invalidate Branch Predictor array by MVA.

Copyright © 2006 ARM Limited. All rights reserved.

7.3 Debug state Cache/MMU Control Registers

In v6 Debug, a Cache Behavior Override Register (CBOR) and a TLB Debug Control Register (TDCR) were recommended in the IMPLEMENTATION DEFINED region of the CP15 register space. The Debug state MMU Control Register (DSMCR) and Debug state Cache Control Register (DSCCR) are not defined in v6 Debug.

In v6.1 Debug, both the CP15 registers (CBOR and TDCR) and CP14 registers (DSMCR and DSCCR) are recommended.

In ARMv7, the DSMCR and DSCCR registers are required, but there can be IMPLEMENTATION DEFINED limits on their behavior. The CBOR and TDCR registers are no longer recommended, but the relevant CP15 instructions remain IMPLEMENTATION DEFINED.

The ARMv7 Debug Architecture requires a pair of additional debug control registers that allow a debugger to disable cache behavior only when in Debug state. There can be IMPLEMENTATION DEFINED limits on their behavior, but must follow as far as possible the behavior described in this architecture.

For details of these registers, see *Memory system control registers* on page 10-61.

Copyright © 2006 ARM Limited. All rights reserved.

Chapter 8
Non-invasive debug

This chapter contains the following sections:

- *About non-invasive debug* on page 8-2
- *Program counter sampling register* on page 8-3
- *Non-invasive debug authentication* on page 8-4.

Copyright © 2006 ARM Limited. All rights reserved.

8.1 About non-invasive debug

Non-invasive debug includes all debug features that allow data and program flow to be observed, but that do not allow modification of the main processor state.

The ARMv7 Debug Architecture implements three areas of non-invasive debug:

- A Program Counter Sampling Register, see *Program counter sampling register* on page 8-3.
- Core-based performance counters. These are described in detail in Chapter 9 *Core-based Performance Counters*
- Instruction and data trace. Trace support is an architecture extension typically implemented using an *Embedded Trace Macrocell* (ETM). The ETM architecture is described in the *Embedded Trace Macrocell Architecture Specification*.

Other forms of non-invasive debug might be implemented in a system.

Copyright © 2006 ARM Limited. All rights reserved.

8.2 Program counter sampling register

In ARMv6, the *Program Counter Sampling Register* (PCSR) is an optional part of the recommended external debug interface. It is not defined by the architecture.

In ARMv7, the PCSR is an IMPLEMENTATION DEFINED feature. It is an optional extension to the debug architecture, that provides a mechanism for course-grained profiling of code executing on the processor core without changing the behavior of that code. See *Program Counter Sampling Register (PSCR)* on page 10-31 for details.

If the program counter sampling register is implemented, bit [13] of the *Debug Identification Register* (DIDR) is set to 1, see *Program Counter Sampling Register implemented, bit [13]* on page 10-4.

When read, the PCSR returns one of the following:

- the address of an instruction *recently executed* by the ARM core
- `0xFFFF FFFF` if the processor is in Debug state, or in a state and mode where non-invasive debug is not permitted.

Note

There is no architectural definition of *recently executed*. The delay between an instruction being executed by the core and its address appearing in the PCSR is not defined. For example, if a piece of code reads the PCSR of the processor it is running on, there is no guaranteed relationship between the program counter for that piece of code and the value read. The PCSR is intended only for use by an external agent to provide statistical information for code profiling.

The value always references a committed instruction. It is IMPLEMENTATION DEFINED whether instructions that do not pass their condition codes are sampled, however, ARM Limited recommends that they are. Implementations must not sample values that reference instructions that are fetched but not committed for execution.

 Copyright © 2006 ARM Limited. All rights reserved.

8.3 Non-invasive debug authentication

Non-invasive debug is enabled though the external debug interface. On processors that implement Security Extensions, this can be conditional on the state of the processor, and the alternatives for when Non-invasive debug is permitted are:

- in all processor modes, in both Secure and Nonsecure worlds
- only in Nonsecure world
- in Nonsecure world and Secure User mode.

Whether Non-invasive debug is permitted in Secure User mode depends on the value of the SUNIDEN bit in the Secure Debug Enable (SDE) Register.

The external debug interface signals that control enabling of non-invasive debug are **DBGEN**, **SPIDEN**, **NIDEN** and **SPNIDEN**.

DBGEN and **SPIDEN** also control invasive debug.

SPIDEN and **SPNIDEN** are only implemented on processors that implement Security Extensions. **NIDEN** is an optional signal in ARMv6.

For more information see:

- the *ARM Architecture Reference Manual Security Extensions Supplement* for details of the Secure Debug Enable Register
- *Authentication signals* on page 6-3 for details of the **DBGEN**, **SPIDEN**, **NIDEN** and **SPNIDEN** signals.

Note

In ARMv6, if **NIDEN** is not present the device behaves as if **NIDEN** is present and tied HIGH. **NIDEN** might be implemented on some non-invasive debug components and not on others. For example, the performance monitoring unit for a processor might implement **NIDEN** when the ETM for the same processor does not.

See *ARMv6 non-invasive debug authentication* on page 8-8 for more information about ARMv6 non-invasive debug.

If both **DBGEN** and **NIDEN** are LOW, all non-invasive debug is disabled.

Non-invasive debug authentication is described in the following sections:

- *Non-invasive debug authentication, Security Extensions not implemented* on page 8-5
- *Non-invasive debug authentication, Security Extensions implemented* on page 8-5.

The behavior of the non-invasive debug components when non-invasive debug is not enabled or not permitted is described in the following sections. These sections also describe the behavior when the core is in Debug state:

- *Core-based performance counters* on page 8-7
- *Program Counter Sampling Register* on page 8-7
- *Trace* on page 8-7.

Copyright © 2006 ARM Limited. All rights reserved.

Note

Only enabling non-invasive debug in Nonsecure modes prevents non-invasive debug features directly observing code processed in Secure modes. However, indirect effects, such as the effect of cache interference between Nonsecure and Secure code, might still be observed. ARM Limited recommends that non-invasive debug is disabled in all modes where such attacks are a concern.

8.3.1 Non-invasive debug authentication, Security Extensions not implemented

On processors that do not implement Security Extensions, if **NIDEN** is asserted HIGH, non-invasive debug is enabled and permitted in all modes.

In ARMv7, if the Security Extensions are not implemented then if **DBGEN** is asserted HIGH the system behaves as if **NIDEN** is asserted HIGH, regardless of the actual state of the **NIDEN** signal.

Table 8-1 shows the required behavior on ARMv7.

Non-invasive debug authentication for ARMv6 systems that do not implement the Security Extensions are described in *ARMv6 non-invasive debug authentication* on page 8-8.

Table 8-1 ARMv7 Non-invasive debug authentication, Security Extensions not implemented

DBGEN	NIDEN	Modes in which non-invasive debug is permitted
LOW	LOW	None
X	HIGH	All modes
HIGH	LOW	All modes

8.3.2 Non-invasive debug authentication, Security Extensions implemented

On processors that implement Security Extensions:

- if both **NIDEN** and **SPNIDEN** are asserted HIGH, non-invasive debug is enabled and permitted in all modes and worlds.
- If **NIDEN** is HIGH and **SPNIDEN** is LOW:
  - — non-invasive debug is enabled and permitted in the Nonsecure world
  - — non-invasive debug is not permitted in Secure privileged modes

Copyright © 2006 ARM Limited. All rights reserved.

— whether non-invasive debug is permitted in Secure User mode depends on the value of the SUNIDEN bit in the SDE Register.

In ARMv7:

- if **DBGEN** is HIGH, the system behaves as if **NIDEN** is HIGH, regardless of the actual state of the **NIDEN** signal
- if **SPIDEN** is HIGH, the system behaves as if **SPNIDEN** is HIGH, regardless of the actual state of the **SPNIDEN** signal.

Table 8-2 shows the non-invasive debug authentication for ARMv7 processors that implement the security extensions.

Non-invasive debug authentication for ARMv6 systems that implement the Security Extensions are described in *ARMv6 non-invasive debug authentication* on page 8-8.

Table 8-2 ARMv7 Invasive debug authentication, Security Extensions implemented

Signals				SUNIDEN[a]	Modes in which non-invasive debug is permitted
DBGEN	**NIDEN**	**SPIDEN**	**SPNIDEN**		
LOW	LOW	X	X	X	None
LOW	HIGH	LOW	LOW	0	All modes in Nonsecure world
LOW	HIGH	LOW	LOW	1	All modes in Nonsecure world Secure User mode
LOW	HIGH	LOW	HIGH	X	All modes in all worlds
LOW	HIGH	HIGH	X	X	All modes in all worlds
HIGH	X	LOW	LOW	0	All modes in Nonsecure world
HIGH	X	LOW	LOW	1	All modes in Nonsecure world Secure User mode
HIGH	X	LOW	HIGH	X	All modes in all worlds
HIGH	X	HIGH	X	X	All modes in all worlds

a. Value of the SUNIDEN bit in the SDE Register.

The value of the SUIDEN bit in the SDE register does not have any effect on non-invasive debug.

Non-invasive debug authentication for ARMv6 systems that implement the Security Extensions are described in *ARMv6 non-invasive debug authentication* on page 8-8.

 Copyright © 2006 ARM Limited. All rights reserved.

8.3.3 Core-based performance counters

Performance counters provide a non-invasive debug feature, and are controlled by the non-invasive debug authentication signals. For more information see Chapter 9 *Core-based Performance Counters*.

The cycle counter, CCNT, is not controlled by the non-invasive debug authentication signals. Disabling the cycle counter in Secure states does not add to the security of the processor. However, a control register flag is provided so that CCNT counting can be disabled in regions of code where the performance counters are disabled. See *Disable CCNT when prohibited, bit [5]* on page 10-82 for details.

Table 8-3 describes the behavior of the performance counters when non-invasive debug is disabled or not permitted, and in Debug state.

Table 8-3 Behavior of performance counters when non-invasive debug not permitted

Debug state	Non-invasive debug permitted and enabled	PMNC[5]	Performance counters enabled and exported	CCNT enabled
Yes	X	X	No	No
No	Yes	X	Yes	Yes
No	No	0	No	Yes
No	No	1	No	No

The performance counters are not intended to be completely accurate, see *Accuracy of performance counters* on page 9-5. In particular, some inaccuracy is permitted at the point of changing security world. However, to avoid the leaking of information from the Secure world, the permitted inaccuracy that non-prohibited transactions can be uncounted. Prohibited transactions must not be counted.

Entry to and exit from Debug state can also disturb the normal running of the processor that causes additional inaccuracy in the performance counters. Disabling the counters while in Debug state limits the extent of this inaccuracy. Implementations can limit this inaccuracy to a greater extent, for example by disabling the counters as soon as possible during the Debug state entry sequence.

8.3.4 Program Counter Sampling Register

When the core is in a mode where non-invasive debug is not permitted, or in Debug state, the PC sample register always reads `0xFFFF FFFF`. See *Program Counter Sampling Register (PSCR)* on page 10-31 for details.

8.3.5 Trace

When the core is in a mode where non-invasive debug is not permitted, or in Debug state, all instructions and data transfers are ignored by the Trace device.

 Copyright © 2006 ARM Limited. All rights reserved.

8.3.6 ARMv6 non-invasive debug authentication

An ARMv6 processor might implement the ARMv7 non-invasive debug authentication signalling described in *Non-invasive debug authentication, Security Extensions not implemented* on page 8-5 and *Non-invasive debug authentication, Security Extensions implemented* on page 8-5. However, in ARMv6 some signal combinations might have IMPLEMENTATION DEFINED alternative behavior that can prevent non-invasive debug. This alternative behavior is described in:

- Table 8-4 for ARMv6 processors that do not implement the Security Extensions
- Table 8-5 on page 8-9 for ARMv6 processors that implement the Security Extensions.

There is no mechanism that a debugger can use to determine whether an ARMv6 processor implements the alternative behavior of the non-invasive debug authentication signals. Therefore, any debug system that incorporates an ARMv6 processor must avoid setting the signal combinations that prevent non-invasive debug.

Table 8-4 shows the IMPLEMENTATION DEFINED alternative behavior of the non-invasive debug authentication signals on ARMv6 processors that do not implement the Security Extensions, and includes all possible combinations of these signals.

Table 8-4 ARMv6 signals with alternative behavior, Security Extensions not implemented

DBGEN	NIDEN	Modes in which non-invasive debug is permitted, ARMv6 alternative behavior
LOW	LOW	None
X	HIGH	All modes
HIGH	LOW	None

Table 8-5 on page 8-9 shows the IMPLEMENTATION DEFINED alternative behavior for some signal combinations on ARMv6 processors that implement the Security Extensions. There is no alternative behavior for signal combinations not listed in the table, and these combinations always act as shown in Table 8-2 on page 8-6.

Copyright © 2006 ARM Limited. All rights reserved.

Table 8-5 ARMv6 signals with alternative behavior, Security Extensions implemented

Signals				SUNIDEN[a]	Modes in which non-invasive debug is permitted, ARMv6 alternative behavior
DBGEN	NIDEN	SPIDEN	SPNIDEN		
HIGH	LOW	LOW	LOW	0	None.
HIGH	LOW	LOW	LOW	1	All modes in Nonsecure world.
HIGH	LOW	X	HIGH	X	None[b].
X	HIGH	HIGH	LOW	0	All modes in Nonsecure world.
X	HIGH	HIGH	LOW	1	All modes in Nonsecure world. Secure User mode.
HIGH	LOW	HIGH	LOW	X	None[b].

a. Value of the SUNIDEN bit in the SDE Register.

b. As noted earlier in this section, in ARMv6 the **NIDEN** signal is optional and the ARMv6 processor might behave as if **NIDEN** is asserted. As a result, some forms of non-invasive debug might be enabled in Nonsecure modes and, if the SUNIDEN flag is set to 1 in the SDE register, in Secure User modes.

In ARMv6, the authentication can vary between types of non-invasive debug. For example, a performance monitoring unit might follow the alternative behavior and the ETM follow the ARMv7 behavior. However, the behavior of a each non-invasive debug component must choose between the ARMv7 prescribed behavior and the ARMv6 alternative behavior. For this reason, and because it is IMPLEMENTATION DEFINED whether **NIDEN** is present, systems incorporating ARMv6 components must avoid certain combinations of **DBGEN**, **NIDEN**, **SPIDEN** and **SPNIDEN**. These forbidden combinations are identified in:

- Table 8-4 on page 8-8, for ARMv6 systems that do not implement the Security Extensions,
- Table 8-5, for ARMv6 systems that implement the Security Extensions.

Copyright © 2006 ARM Limited. All rights reserved.

Copyright © 2006 ARM Limited. All rights reserved.

Chapter 9
Core-based Performance Counters

This chapter contains the following sections:

- *About core-based performance counters* on page 9-2
- *Status in the ARM architecture* on page 9-4
- *Accuracy of performance counters* on page 9-5
- *Behavior on overflow* on page 9-6
- *Interaction with Security Extensions* on page 9-7
- *Interaction with trace* on page 9-8
- *Interaction with power saving operations* on page 9-9
- *Register map* on page 9-10
- *Access permissions* on page 9-12
- *Event numbers* on page 9-13.

 Copyright © 2006 ARM Limited. All rights reserved.

9.1 About core-based performance counters

The basic form of the core-based performance counters consists of:

- A cycle counter. This can be incremented either on every cycle, or once every 64 cycles.
- A number of performance counters whose events can be programmed. Space is provided in the architecture for up to 31 counters. The actual number of counters is IMPLEMENTATION DEFINED, and there is an identification mechanism for the counters.
- Controls for enabling the counters, resetting the counters, flagging overflows, and enabling interrupts on counter overflow.

  The cycle counter can be enabled independently of the rest of the performance counters.

The counters are held in a set of registers that can be accessed in coprocessor space. This means the counters can be accessed from the operating system running on the core, enabling a number of uses, including:

- dynamic compilation techniques
- energy management.

In addition, you can provide access to the counters from application code, if required. This allows applications to monitor their own performance with fine grain control without requiring operating system support. For example, an application might implement per-function performance monitoring.

There are many situations where performance monitoring features integrated into the core are valuable for applications and for application development. When an operating system does not use the performance counters itself, ARM Limited recommends that it enables application code access to the performance counters. However an implementations can choose not to implement any performance counters.

To allow interaction with external monitoring, an implementation might consider additional enhancements, including:

- Providing a set of events, from which a section can be exported onto a bus for use as external events. For very high frequency operation, this might introduce unacceptable timing requirements, but the bus could be interfaced to the ETM or another closely coupled resource.
- Providing the ability to trace external events. Here, again, there are clock frequency issues between the core and the system. A suitable approach might be to edge-detect changes in the signal and to use those changes to increment a counter. This requires the core to implement a set of external event input pins.
- Providing memory-mapped access to the performance counter registers, to allow the counter resources to be used for system monitoring in systems where they are not used by the software running on the core. Such a memory-mapped access is not described in this document, but might follow the structure of the debug memory map described in *The Memory-mapped and recommended external debug interfaces* on page 5-34.

The set of events that might be monitored splits into:

- events that are likely to be consistent across many micro-architectures
- implementation specific events.

Copyright © 2006 ARM Limited. All rights reserved.

Therefore, this architecture defines a core set of events to be used across many micro-architectures, together with a large space reserved for IMPLEMENTATION DEFINED events. The full set of events for any given implementation is IMPLEMENTATION DEFINED, and there is no requirement to implement any of the core set of events.

Copyright © 2006 ARM Limited. All rights reserved.

9.2 Status in the ARM architecture

The status of the core-based performance counters block is that it is an IMPLEMENTATION DEFINED space for ARMv7, but ARM Limited recommends implementers to use this approach.

 Copyright © 2006 ARM Limited. All rights reserved.

9.3 Accuracy of performance counters

The performance counters are designed to provide approximately accurate performance count information, but to keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. There is no exact definition of reasonable degree of inaccuracy, but the following guidelines are recommended:

- Under normal operating conditions, the counters must present an accurate value of the count.
- In exceptional circumstances, such as changes in Security state or other boundary conditions, it is acceptable for the count to be inaccurate.
- Under very unusual non-repeating pathological cases counts can be inaccurate. These are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is vanishingly unlikely.

Note

Implementations must not introduce inaccuracies that can be triggered systematically by normal pieces of code that are running. For example, dropping a branch count in a loop due to the structure of the loop gives a systematic error that makes the count of branch behavior very inaccurate, and this is not reasonable. However, the dropping of a single branch count as the result of a rare interaction with an interrupt is acceptable.

The permissibility of inaccuracy limits the possible uses of the performance counters. In particular, the point in a pipeline where the performance counter is incremented is not defined relative to the point where a read of the performance counters is made, so allowing for some imprecision due to pipelining effects.

Implementations must document any particular scenarios where significant inaccuracies are expected.

Copyright © 2006 ARM Limited. All rights reserved.

9.4 Behavior on overflow

On counter overflow:

- An overflow status flag bit is set to 1. See *Overflow Flag Status Register (FLAG)* on page 10-87.
- An interrupt is generated if the processor is configured to generate counter overflow interrupts. See *Interrupt Enable Set Register (INTENS)* on page 10-83 and *Interrupt Enable Clear Register (INTENC)* on page 10-84 for details.
- The counter wraps to zero and continues counting events. Counting continues as long as the counters are enabled, regardless of any overflows.

The counter always resets to zero and overflows after 32 bits of increment. To allow a more frequent generation of interrupts, the counters can be written to. For example, an interrupt handler can reset the overflowed counter to `0xFFFF 0000` (–65536) to generate another overflow interrupt after 16 bits of increment.

The interrupt handler must cancel the interrupt by clearing the overflow flag.

Copyright © 2006 ARM Limited. All rights reserved.

9.5 Interaction with Security Extensions

Performance counters provide a non-invasive debug feature, and therefore are controlled by the non-invasive debug authentication signals. *Non-invasive debug authentication* on page 8-4 describes how non-invasive debug interacts with Security Extensions.

Core-based performance counters on page 8-7 describes the behavior of the performance counters when non-invasive debug is not permitted.

Note

Additional controls in the PMNC register can also disable the performance counters and the CCNT. The PMNC register controls take precedence over the authentication controls.

The performance counters are not banked, and are always accessible regardless of the values of the authentication signals and SUNIDEN. The purpose of authentication is to control whether the counters count events, not to control access to the performance counter registers.

Copyright © 2006 ARM Limited. All rights reserved.

9.6 Interaction with trace

It is IMPLEMENTATION DEFINED whether events are exported to the Embedded Trace Macrocell or other external monitoring agents to provide triggering information. The form of the exporting is also IMPLEMENTATION DEFINED. If implemented, this exporting might be enabled as part of the performance monitoring control functionality.

Similarly, ARM Limited recommends that a mechanism for importing a set of external events to be counted is implemented, but such a feature is IMPLEMENTATION DEFINED. When implemented, this feature allows the Trace module to pass in events to be counted.

Copyright © 2006 ARM Limited. All rights reserved.

9.7 Interaction with power saving operations

All counters are subject to any changes in clock frequency, including clock stopping caused by the `WFI` and `WFE` instructions.

Copyright © 2006 ARM Limited. All rights reserved.

9.8 Register map

The performance counter registers are mapped into part of the CP15 register map. The registers are described in *Core-based performance counters registers* on page 10-80.

Table 9-1 Recommended reset scheme

Instruction[a]	Name or notes
MRC|MCR p15,0,Rd,c9,c12,0	*Performance Monitor Control Register (PMNC)* on page 10-80
MRC|MCR p15,0,Rd,c9,c12,1	*Count Enable Set Register (CNTENS)* on page 10-85
MRC|MCR p15,0,Rd,c9,c12,2	*Count Enable Clear Register (CNTENC)* on page 10-86
MRC|MCR p15,0,Rd,c9,c12,3	*Overflow Flag Status Register (FLAG)* on page 10-87
MRC|MCR p15,0,Rd,c9,c12,4	*Software Increment Register (SWINCR)* on page 10-89
MRC|MCR p15,0,Rd,c9,c12,5	*Performance Counter Selection Register (PMNXSEL)* on page 10-90
MRC|MCR p15,0,Rd,c9,c12,<n>	<n> is 6 or 7. Reserved
MRC|MCR p15,0,Rd,c9,c13,0	*Cycle Count Register (CCNT)* on page 10-89
MRC|MCR p15,0,Rd,c9,c13,1	*Event Select Register (EVTSELX)* on page 10-90
MRC|MCR p15,0,Rd,c9,c13,2	*Performance Count Registers (PMNX)* on page 10-91
MRC|MCR p15,0,Rd,c9,c13,<n>	<n> is 3 - 7. Reserved
MRC|MCR p15,0,Rd,c9,c14,0	*User Enable Register (USEREN)* on page 10-91
MRC|MCR p15,0,Rd,c9,c14,1	*Interrupt Enable Set Register (INTENS)* on page 10-83
MRC|MCR p15,0,Rd,c9,c14,2	*Interrupt Enable Clear Register (INTENC)* on page 10-84
MRC|MCR p15,0,Rd,c9,c14,<n>	<n> is 3 - 7. Reserved

a. All registers can be accessed by both MRC and MCR commands. Only the MRC commands are listed. The MCR commands have the same syntax, for example for the PMC Register the command is MCR p15,0,Rd,c9,c12,0

 Copyright © 2006 ARM Limited. All rights reserved.

9.8.1 Power domains and performance counters registers reset

ARMv7 recommends that performance counters are implemented as part of the processor logic power domain, not as part of a separate debug logic power domain. There is no interface to access the performance counter registers when the processor logic is powered down.

The performance counter registers must be set to their reset values on a core reset (**nSYSPORESET**, **nCOREPORESET** or **nRESET**). Performance counter registers are not changed by a debug logic reset (**PRESETDBGn**).

See *Recommended reset scheme* on page 5-12 for more information on the reset scheme recommended by ARMv7.

Copyright © 2006 ARM Limited. All rights reserved.

9.9 Access permissions

Normally the Performance Counter Registers are accessible from privileged modes only. To allow access from User Mode code, for example for instrumentation and profiling purposes, a control flag is provided in the USEREN Register. However, USEREN does not provide access to the registers that control interrupt generation.

Table 9-2 Performance Counter Access Permissions

Register	Operation	Access from a privileged mode	Access from User Mode[a]	
			USEREN = 0	USEREN = 1
PMNC	MRC or MCR	Proceed	UNDEFINED	Proceed
CNTENS	MRC or MCR	Proceed	UNDEFINED	Proceed
CNTENC	MRC or MCR	Proceed	UNDEFINED	Proceed
FLAG	MRC or MCR	Proceed	UNDEFINED	Proceed
SWINCR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMNXSEL	MRC or MCR	Proceed	UNDEFINED	Proceed
CCNT	MRC or MCR	Proceed	UNDEFINED	Proceed
EVTSELX	MRC or MCR	Proceed	UNDEFINED	Proceed
PMNX	MRC or MCR	Proceed	UNDEFINED	Proceed
USEREN[a]	MRC	Proceed	Proceed	Proceed
	MCR	Proceed	UNDEFINED	UNDEFINED
INTENS	MRC or MCR	Proceed	UNDEFINED	UNDEFINED
INTENC	MRC or MCR	Proceed	UNDEFINED	UNDEFINED
Reserved [b]	MRC or MCR	UNPREDICTABLE	UNDEFINED	UNDEFINED

a. See *User Enable Register (USEREN)* on page 10-91 for details.

b. All the registers marked as Reserved in Table 9-1 on page 9-10.

Copyright © 2006 ARM Limited. All rights reserved.

9.10 Event numbers

The event numbers are described in the following subsections:

- *Common feature numbers*
- *implementation defined feature numbers* on page 9-16.

9.10.1 Common feature numbers

For the common features, normally the counters must increment only once for each event. Exceptions to this rule are stated in the individual definitions.

In these definitions, the term *architecturally executed* means that the instruction flow is such that the counted instruction would have been executed in a simple sequential execution model.

Note

An instruction is architecturally executed if the behavior of the program on the processor is consistent with the instruction having been executed on a simple execution model of the architecture. Therefore an instruction that has been executed and retired is defined to be *architecturally executed*. In processors that perform speculative execution, an instruction is not architecturally executed if the results of the speculative execution are discarded. Where an instruction has no visible effect (for example, a `NOP`), the point where the instruction is retired is IMPLEMENTATION DEFINED.

The common feature numbers are assigned to the following events:

`0x00` Software increment. The register is incremented only on writes to the Software Increment Register. See *Software Increment Register (SWINCR)* on page 10-89 for details.

`0x01` Instruction fetch that causes a refill at (at least) the lowest level(s) of instruction or unified cache. Each instruction fetch from normal cacheable memory that causes a refill from outside of the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss, are not counted. Where instruction fetches consist of multiple instructions, these accesses count as single events. CP15 cache maintenance operations do not count as events. This counter increments on speculative instruction fetches as well as on fetches of instructions that reach execution.

`0x02` Instruction fetch that causes a TLB refill at (at least) the lowest level of TLB. Each instruction fetch that causes an access to a level of memory system due to a page table walk or an access to another level of TLB caching is counted CP15 TLB maintenance operations do not count as events. This counter increments on speculative instruction fetches as well as on fetches of instructions that reach execution.

`0x03` Data Read or Write operation that causes a refill at (at least) the lowest level of data or unified cache. Each data read from or write to normal cacheable memory that causes a refill from outside of the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss are not counted. Each access to a cache line to normal cacheable memory that causes a new linefill is counted, including the multiple transactions of load or store multiples, including `PUSH` and `POP`. Write-through writes that hit

 Copyright © 2006 ARM Limited. All rights reserved.

in the cache do not cause a linefill and so are not counted. CP15 cache maintenance operations do not count as events. This counter increments on speculative data accesses as well as for data accesses that are explicitly made by instructions.

`0x04` Data Read or Write operation that causes a cache access at (at least) the lowest level(s) of data or unified cache. Each access to a cache line to normal cacheable memory is counted including the multiple transactions of instructions such as `LDM` or `STM`. CP15 cache maintenance operations do not count as events. This counter increments on speculative data accesses as well as for data accesses that are explicitly made by instructions.

`0x05` Data Read or Write operation that causes a TLB refill at (at least) the lowest level of TLB. Each data read or write operation that causes a page table walk or an access to another level of TLB caching is counted. CP15 TLB maintenance operations do not count as events. This counter increments on speculative data accesses as well as for data accesses that are explicitly made by instructions.

`0x06` Data Read architecturally executed. This counter increments for every instruction that explicitly read data (including `SWP`). This counter only increments for instructions that are unconditional or that pass their condition codes.

`0x07` Data Write architecturally executed. The counter increments for every instruction that explicitly wrote data (including `SWP`). This counter only increments for instructions that are unconditional or that pass their condition codes.

`0x08` Instruction architecturally executed. This counter counts for all instructions, including conditional instructions that fail their condition code.

`0x09` Exception taken. This counts for each exception taken.

`0x0A` Exception return architecturally executed. This counts every exception return:

- `RFE <addressing_mode> <Rn>{!}`
- `MOVS PC` (and other similar data processing instructions)
- `LDM <addressing_mode> Rn{!}, <registers_and_pc>`

These instructions, with the exception of `RFE`, copy the SPSR to the CPSR. `RFE` copies a value in memory to the CPSR. This counter only increments for instructions that are unconditional or that pass their condition codes.

`0x0B` Instruction that writes to the Context ID register architecturally executed. This counter only increments for instructions that are unconditional or that pass their condition codes.

`0x0C` Software change of PC (except by an exception) architecturally executed. This counter only increments for instructions that are unconditional or that pass their condition codes.

`0x0D` Immediate branch architecturally executed:

- `B{L} <target_address>`
- `BLX <target_address>`
- `CB{N}Z <target_address>`
- `HB{L} #<handler_id>` (ThumbEE state only)

Copyright © 2006 ARM Limited. All rights reserved.

- `HB{L}P #<immed>, #<handler_id>` (ThumbEE state only).

This counter counts for all immediate branch instructions that are architecturally executed, including conditional instructions that fail their condition code.

`0x0E` Procedure return (other than exception returns) architecturally executed:

- `BX R14`
- `MOV PC, LR`
- `POP {...,PC}`
- `LDR PC,[R13],#offset`
- `LDMIA R9!,{...,PC}` (ThumbEE state only)
- `LDR PC,[R9],#offset` (ThumbEE state only).

This counter only increments for instructions that are unconditional or that pass their condition codes.

Note

Only these instructions are counted as procedure returns. For example, the following are not counted as procedure return instructions:

- `BX R0` (Rm != R14)
- `MOV PC,R0` (Rm != R14)
- `LDM R13,{...,PC}` (writeback not specified)
- `LDR PC,[R13,#offset]` (wrong addressing mode).

`0x0F` Unaligned access architecturally executed. This counts each instruction that was an access to an unaligned address. That is, either triggered an unaligned fault, or would have done so if the A-bit in the CPSR had been 1. This counter only increments for instructions that are unconditional or that pass their condition codes.

`0x10` Branch mispredicted/not predicted: this counts for every pipeline flush due to a misprediction from the program flow prediction resources that could have predicted correctly within the core.

`0x11` Reserved.

`0x12` Branches or other change in program flow that could have predicted by the branch prediction resources of the processor.

`0x13-0x3F` Reserved.

Copyright © 2006 ARM Limited. All rights reserved.

9.10.2 IMPLEMENTATION DEFINED feature numbers

For IMPLEMENTATION DEFINED feature numbers, the counters are defined to increment only once for each event, or they can be used to count the duration for which an event occurs as defined for each feature.

Implementers are encouraged to establish house styles for these events, with common definitions, and common count numbers, applied to all cores implemented by a particular implementer. In general, the approach is for standardization across implementations with common features. However, it is recognized that the approach of standardizing micro-architecturally specific features across too wide a range is not productive.

It is strongly recommended that at least the following classes of event are identified within this section:

- Cumulative duration of stalls due to the holes in the instruction availability, separating out counts for key buffering points that might exist.
- Cumulative duration of stalls due to data dependent stalling, separating out counts for key dependency classes that might exist.
- Cumulative duration of stalls due to unavailability of execution resources (including write buffers, for example), separating out counts for key resources that might exist.
- Missed superscalar issue opportunities (if relevant), separating out counts for key classes of issue that might exist.
- Miss rates for different levels of caches and TLB.
- Transaction counts on external buses.
- External events passed into the core via an IMPLEMENTATION DEFINED mechanism. Typically this involves counting the number of cycles while the signal is asserted.
- Cumulative duration that I and F interrupt masks are set to 1.
- Any other micro-architectural features that are deemed valuable to count.

IMPLEMENTATION DEFINED feature numbers are `0x40` to `0xFF`.

Copyright © 2006 ARM Limited. All rights reserved.

Chapter 10
Debug Register Reference

See *Debug Register Map* on page 5-13 for a list of all the debug registers.

In this section, the register offsets refer to the offsets in the ARMv7 Memory-mapped or external debug interface. In ARMv7 there is also a canonical mapping from these offsets to coprocessor instructions in the ARMv7 Extended CP14 Interface, see *Extended CP14 interface* on page 5-25. The locations of these registers in the ARMv6 external debug interface might differ.

The register numbers and offsets for the DSCR, DTRRX and DTRTX registers apply only to the external view of that register. See *Internal and external views of DSCR and DTR* on page 5-16 for more information.

Note

The recommended ARMv7 external debug interface is described in *ARM Debug Interface v5 Architecture Specification*. Contact ARM Limited if you require details of the ARMv6 recommended external debug interface.

This chapter contains the following sections:

- *Identification registers* on page 10-3
- *Control and status registers* on page 10-8
- *Instruction and data transfer registers* on page 10-32
- *Breakpoint and watchpoint registers* on page 10-39
- *Operating-system save and restore registers* on page 10-58
- *Memory system control registers* on page 10-61

 Copyright © 2006 ARM Limited. All rights reserved.

- *Management registers* on page 10-69
- *Core-based performance counters registers* on page 10-80.

Copyright © 2006 ARM Limited. All rights reserved.

10.1 Identification registers

This section contains the following subsections:

- *Debug ID Register (DIDR)*
- *Debug ROM Address Register (DRAR)* on page 10-5
- *Debug Self Address Offset Register (DSAR)* on page 10-6.

10.1.1 Debug ID Register (DIDR)

The DIDR is register 0, at offset `0x000`.

The DIDR is required on all versions of the debug architecture from ARMv6 onwards. It specifies which version of the debug architecture is implemented. Table 10-1 shows the layout of the Debug Identification Register.

Table 10-1 Debug Identification Register bit definitions

Bits	Access	Description
[31:28]	RO	*Number of Watchpoint Register Pairs (WRPs) implemented, bits [31:28]* on page 10-5
[27:24]	RO	*Number of Breakpoint Register Pairs implemented, bits [27:24]* on page 10-5
[23:20]	RO	*Number of Breakpoint Register Pairs (BRPs) with Context ID comparison capability, bits [23:20]* on page 10-4
[19:16]	RO	*Debug Architecture Version, bits [19:16]* on page 10-4
[15:14]	RAZ	Reserved
[13]	RO	*Program Counter Sampling Register implemented, bit [13]* on page 10-4
[12]	RO	*Security Extensions implemented, bit [12]* on page 10-4
[11:8]	RAZ	Reserved
[7:4]	RO	*Variant, bits [7:4]* on page 10-4
[3:0]	RO	*Revision, bits [3:0]*

Revision, bits [3:0]

Bits [3:0]. The value of the Revision field is IMPLEMENTATION DEFINED. This number is incremented on corrections. The value must match bits [3:0] of the CP15 Main ID register.

Copyright © 2006 ARM Limited. All rights reserved.

Variant, bits [7:4]

Bits [7:4]. The value of the Variant field is IMPLEMENTATION DEFINED. This number is incremented on functional changes. The value must match bits [23:20] of the CP15 Main ID register.

Security Extensions implemented, bit [12]

The meanings of the values of bit [12] are as follows:

0	Security Extensions are not implemented
1	Security Extensions are implemented.

v6 Debug is not a permitted option for implementations that include Security Extensions. This bit always reads as zero in v6 Debug.

Program Counter Sampling Register implemented, bit [13]

In ARMv6, the Program Counter Sampling Register is an IMPLEMENTATION DEFINED feature of the Debug Access Port, and is not indicated in the DIDR.

The meanings of the values of bit [13] are as follows:

0	Program Counter Sampling Register (PCSR) is not implemented
1	Program Counter Sampling Register (PCSR) is implemented.

See also *Program Counter Sampling Register (PSCR)* on page 10-31.

Debug Architecture Version, bits [19:16]

The meanings of the values of bits [19:16] are as follows:

b0001	ARMv6 Debug Architecture
b0010	v6.1 Debug Architecture
b0011	ARMv7 Debug Architecture - Extended CP14 interface implemented
b0100	ARMv7 Debug Architecture - No Extended CP14 Interface implemented.

Number of Breakpoint Register Pairs (BRPs) with Context ID comparison capability, bits [23:20]

The meanings of the values of bits [23:20] are as follows:

b0000	1 BRP has Context ID comparison capability
b0001	2 BRPs have Context ID comparison capability
b0010	3 BRPs have Context ID comparison capability
...	...
b1111	16 BRPs have Context ID comparison capability.

The minimum number of BRPs with Context ID comparison capability is 1.

Copyright © 2006 ARM Limited. All rights reserved.

The breakpoint comparators with Context ID comparison capability *must* be the highest addressed comparators. For example, if six comparators are implemented and two have Context ID comparison capability, these must be comparators 4 and 5.

Number of Breakpoint Register Pairs implemented, bits [27:24]

The meanings of the values of bits [27:24] are as follows:

b0000 Reserved
b0001 2 BRPs implemented
b0010 3 BRPs implemented
... ...
b1111 16 BRPs implemented.

The minimum number of BRPs is 2.

Number of Watchpoint Register Pairs (WRPs) implemented, bits [31:28]

The meanings of the values of bits [31:28] are as follows:

b0000 1 WRP implemented
b0001 2 WRPs implemented
b0010 3 WRPs implemented
... ...
b1111 16 WRPs implemented.

The minimum number of WRPs is 1.

10.1.2 Debug ROM Address Register (DRAR)

v6, v6.1 This register is not defined in ARMv6.

v7 If no Memory-mapped debug components (including this processor) are implemented, this register reads as zero.

This register is only implemented through the Baseline CP14 interface. See *Coprocessor interface* on page 5-24.

The *Debug ROM Address Register* (DRAR) defines the physical address in memory of a ROM table that locates the debug components in the system. The ROM table contains a zero-terminated list of signed 32-bit offsets from the ROM table base to other Memory-mapped debug components in the system. All the debug components pointed to must contain a set of Component Identification Registers compatible with the format in *Component Identification Registers (COMPONENTID)* on page 10-79.

It is IMPLEMENTATION DEFINED how the processor determines the value to be read in debug ROM address. If the processor cannot determine the value, the register must read as zero.

Copyright © 2006 ARM Limited. All rights reserved.

One implementation scheme is to provide inputs **DBGROMADDR**[31:12] and **DBGROMADDRV** that a system designer must tie-off to the correct value. **DBGROMADDRV** must be tied HIGH only if **DBGROMADDR**[31:12] is tied off to a valid value, otherwise **DBGROMADDR**[31:12] and **DBGROMADDRV** must be tied LOW.

The DRAR is read-only. Table 10-2 shows the layout of the DRAR.

Table 10-2 Debug ROM Address Register

Bits	Access	Value	Description
[31:12]	RO	**DBGROMADDR**[31:12]	Bits [31:12] of the debug ROM physical address.
[11:2]	RAZ/WI	-	Reserved.
[1:0]	RO	(**DBGROMADDRV**), see *Description* entry	Reads b11 if **DBGROMADDRV** is tied HIGH, reads b00 otherwise.

10.1.3 Debug Self Address Offset Register (DSAR)

v6, v6.1 This register is not defined in ARMv6.

v7 If no Memory-mapped interface is provided, this register reads as zero.

This register is only implemented through the Baseline CP14 interface. See *Coprocessor interface* on page 5-24.

The *Debug Self Address Offset Register* (DSAR) gives the offset from the debug ROM address register to the physical address of the processor's own debug registers.

If the processor cannot determine the offset from the debug ROM address register, this register must read as zero, and software must scan the contents of the debug ROM (if provided) to locate the processor.

It is IMPLEMENTATION DEFINED how the processor determines the value to be read in debug self address offset. If the processor cannot determine the value, the register must read as zero.

One implementation scheme is to provide inputs **DBGSELFADDR**[31:12] and **DBGSELFADDRV** that a system designer must tie-off to the correct value. **DBGSELFADDRV** must be tied HIGH only if **DBGSELFADDR**[31:12] is tied off to a valid value, otherwise **DBGSELFADDR**[31:12] and **DBGSELFADDRV** must be tied LOW.

The DSAR is read-only. Table 10-3 on page 10-7 shows the layout of the DSAR.

 Copyright © 2006 ARM Limited. All rights reserved.

Table 10-3 Debug Self Address Offset Register

Bits	Access	Value	Description
[31:12]	RO	**DBGSELFADDR**[31:12]	Bits [31:12] of the 2's complement offset from the debug ROM physical address to the physical address where the debug registers are mapped.
[11:2]	RAZ/WI	-	Reserved
[1:0]	RO	(**DBGSELFADDRV**), see *Description* entry	Reads b11 if **DBGSELFADDRV** is tied HIGH, reads b00 otherwise.

Copyright © 2006 ARM Limited. All rights reserved.

10.2 Control and status registers

This section contains the following subsections:

- *Debug Status and Control Register (DSCR)*
- *Watchpoint Fault Address Register (WFAR)* on page 10-22
- *Debug Run Control Register (DRCR)* on page 10-23
- *Device Power-Down and Reset Control Register (PRCR)* on page 10-25
- *Device Power-Down and Reset Status Register (PRSR)* on page 10-28
- *Program Counter Sampling Register (PSCR)* on page 10-31.

10.2.1 Debug Status and Control Register (DSCR)

The DSCR external view is register 34 at offset `0x088`.

The DSCR external view is the main control register for the debug facilities in the ARM architecture.

In all debug versions there are two views of the DSCR: INT-DSCR (internal view) and EXT-DSCR (external view). See *Internal and external views of DSCR and DTR* on page 5-16 for definitions of the INT- and EXT- views.

The behavior differs only on the Access Type (for ARMv6 only) and on the behavior of DTRRXfull and DTRTXfull on reads of DSCR through the two views.

Table 10-4 Debug Status and Control Register bit definitions

Bits	Versions	Access [a]	Debug reset value	Description
[31]	-	RAZ/SBZP	-	Reserved
[30]	All	RO	0	*DTRRX register full (DTRRXfull), bit [30]* on page 10-22
[29]	All	RO	0	*DTRTX register full (DTRTXfull), bit [29]* on page 10-21
[28]	-	RAZ/SBZP	-	Reserved
[27]	v7	RO	0	Bit 27 is the Latched DTRRXfull (DTRRXfull_l) bit. Bit 26 is the Latched DTRTXfull (DTRTXfull_l) bit. See *Latched DTRTXfull (DTRTXfull_l) and Latched DTRRXfull (DTRRXfull_l), bits [27:26]* on page 10-21.
[26]	v7	RO	0	
[25]	v7	RO	UNP	*Sticky Pipeline Advance (PipeAdv), bit [25]* on page 10-21
[24]	v7	RO	UNP	*Latched Instruction Complete (InstrCompl_l), bit [24]* on page 10-20
[23:22]	-	RAZ/SBZP	-	Reserved
[21:20]	v7	RW_{EXT}	00	*EXT-DTR Access Mode, bits [21:20]* on page 10-16
[19]	v6.1, v7	RO/RW[b]	0	*Imprecise Data Aborts discarded, bit [19]* on page 10-16

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-4 Debug Status and Control Register bit definitions (continued)

Bits	Versions	Access [a]	Debug reset value	Description
[18]	v6.1, v7	RO	[e]	*Nonsecure world status (NS), bit [18]* on page 10-16
[17]	v6.1, v7	RO	[e]	*Secure Privileged Non-invasive Debug Disabled (SPNIDDIS), bit [17]* on page 10-15
[16]	v6.1, v7	RO	[e]	*Secure Privileged Invasive Debug Disabled (SPIDDIS), bit [16]* on page 10-15
[15]	All	RW_{INT}	0	*Monitor Debug-mode enable, bit [15]* on page 10-15
[14]	All	RW_{EXT}	0	*Halting Debug-mode enable, bit [14]* on page 10-15
[13]	All	RW_{EXT}	0	*Execute ARM instruction enable, bit [13]* on page 10-14
[12]	All	RW_{INT}	0	*User Mode access to Comms Channel disable, bit [12]* on page 10-14
[11]	All	RW_{EXT}	0	*Interrupts Disable (IntDis), bit [11]* on page 10-14
[10]	All	RW_{EXT}	0	*Force Debug Acknowledge (DbgAck), bit [10]* on page 10-13
[9]	v6, v6.1[c]	RW_{EXT}	0	*No Power-down (DBGNOPWRDWN), bit [9]* on page 10-13
[8]	v6.1, v7	RO [d]	0	*Sticky Undefined, bit [8]* on page 10-13
[7]	All	RO [d]	0	*Sticky Imprecise Abort, bit [7]* on page 10-12
[6]	All	RO [d]	0	*Sticky Precise Abort, bit [6]* on page 10-11
[5:2]	All	RW	0	*Method of Debug Entry (MOE), bits [5:2]* on page 10-10
[1]	All	RO	[e]	*Core Restarted, bit [1]* on page 10-10
[0]	All	RO	[e]	*Core Halted, bit [0]* on page 10-10

a. In ARMv6 RW bits can only be written through either the coprocessor instruction (INT) or through the Debug Access Port (EXT), as indicated by the subscript. For example, the Disable Interrupts bit (bit [11]) is read-write through the *Debug Access Port* (DAP) but read-only through the coprocessor interface. In ARMv7 INT-DSCR is read-only.

b. DSCR[19] can be RW. This is IMPLEMENTATION DEFINED, see *Imprecise Data Aborts and entry to Debug state* on page 4-4.

c. This bit is Reserved in ARMv6, but some implementations have used it for the same purpose as the No Power-down bit in the Device Power-down and Reset Control Register. In ARMv7 the Device Power-down and Reset Control Register is implemented, and this bit is Reserved and RAZ/SBZP.

d. In ARMv6 these bits are cleared to 0 on read through the Debug Access Port. In ARMv7 these registers have to be cleared to 0 using a write to the Debug Run Control Register, see *Debug Run Control Register (DRCR)* on page 10-23.

e. These are read-only status bits that reflect the current state of the processor.

Copyright © 2006 ARM Limited. All rights reserved.

Core Halted, bit [0]

After programming a Debug event, the external debugger can poll this bit until it is 1, so that it knows that the processor has entered Debug state. See Chapter 4 *Debug State* for a definition of Debug state.

The meanings of the values of bit [0] are as follows:

0 The processor is in normal state

1 The processor is in Debug state.

Core Restarted, bit [1]

After forcing the processor to leave Debug state, the external debugger polls this bit until it is set to 1 so that it knows that the exit command has taken effect and the processor has exited Debug state. Polling DSCR[0] until it is set to 0 is not safe, because the processor could re-enter Debug state due to another Debug event before the external debugger samples the DSCR. See Chapter 4 *Debug State* for a definition of Debug state.

The meanings of the values of bit [1] are as follows:

0 The processor is exiting Debug state. This bit only reads as 0 between receiving a request to exit Debug state, and restarting normal state

1 The processor has exited Debug state. This bit remains 1 if the processor re-enters Debug state.

Method of Debug Entry (MOE), bits [5:2]

Table 10-5 shows the meanings of Method of Debug Entry values.

Table 10-5 Meaning of Method of Debug Entry values

Value	Versions	Description	Section
b0000	All	Halt Request Debug event occurred.	*Halting Debug events* on page 2-18
b0001	All	Breakpoint Debug event occurred.	*Breakpoint Debug events* on page 2-10
b0010	All	Imprecise Watchpoint Debug event occurred.	*Watchpoint Debug events* on page 2-6
b0011	All	BKPT Instruction Debug event occurred.	*BKPT Instruction Debug events* on page 2-12
b0100	All	External Debug Request Debug event occurred.	*Halting Debug events* on page 2-18
b0101	All	Vector catch Debug event occurred.	*Vector Catch Debug events* on page 2-13
b0110	v6 only	D-side abort occurred. This value is Reserved in v6.1 and v7.	-

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-5 Meaning of Method of Debug Entry values (continued)

Value	Versions	Description	Section
b0111	v6 only	I-side abort occurred. This value is Reserved in v6.1 and v7.	-
b1000	v7	OS Unlock Catch Debug event occurred. This value is Reserved in v6 and v6.1.	*Halting Debug events* on page 2-18
b1001	All	Reserved.	-
b1010	v7	Precise Watchpoint Debug event occurred. This value is Reserved in v6 and v6.1.	*Watchpoint Debug events* on page 2-6
b1011-b1111	All	Reserved.	-

A Prefetch abort or Data Abort handler can determine whether a Debug event occurred by checking the value of the relevant Fault Status Register (IFSR or DFSR), and use these bits to determine the specific Debug event.

In v6 Debug, the DSCR can be checked first to determine whether an abort had occurred, and hence whether the abort handler jumps to the debug monitor or not. In v6.1 and ARMv7 the *D-side abort occurred* and *I-side abort occurred* encodings are Reserved. An abort handler must therefore always check the IFSR or DFSR first.

When debug is disabled or not permitted, the `BKPT` instruction generates a Debug exception rather than being ignored. The DSCR, IFSR, and IFAR are set as if a BKPT Instruction Debug Exception occurred. See *Effects of Debug Exceptions on CP15 registers and the WFAR* on page 3-4. Monitor software might also need to check that debug was not disabled for security reasons before communicating with an external debugger.

In ARMv7 support for precise watchpoint events is added, see *Precise and Imprecise Watchpoint Debug events* on page 2-8.

Sticky Precise Abort, bit [6]

v6 If the DSCR[13] (Execute ARM instruction enable) bit is 0, or the core is not in Debug state, the value of this flag is UNPREDICTABLE.

v6, v6.1 This flag is cleared to 0 on reads of the DSCR by the external debugger.

v6.1 If the core is not in Debug state this flag reads as zero.

v6.1, v7 This flag does not change when the processor is not in Debug state.

 Copyright © 2006 ARM Limited. All rights reserved.

v7 This flag can be cleared to 0 only by writing to DRCR[2], see *Debug Run Control Register (DRCR)* on page 10-23.

Leaving Debug state with this flag set to 1 leads to UNPREDICTABLE behavior.

This flag is used to detect precise Data Aborts generated by instructions issued to the processor while in Debug state.

The meanings of the values of bit [6] are as follows:

0 no precise Data Abort exception occurred since the last time this bit was cleared to 0

1 a precise Data Abort exception has occurred since the last time this bit was cleared to 0.

If DSCR[6] is set to 1 then instructions are not issued by the Instruction Transfer Register. See *Instruction Transfer Register (ITR)* on page 10-37, and also *EXT-DTR Access Mode, bits [21:20]* on page 10-16.

See also *Exceptions in Debug state* on page 4-19.

Sticky Imprecise Abort, bit [7]

v6 If the DSCR[13] (Execute ARM instruction enable) bit is 0, or the core is not in Debug state, the value of this flag is UNPREDICTABLE.

v6, v6.1 This flag is cleared to 0 on reads of the DSCR by the external debugger.

v6.1 This flag is set to 1 on all imprecise aborts occurring when in Debug state.

If the core is not in Debug state, this flag reads as zero.

v6.1, v7 This flag does not change when the processor is not in Debug state.

v7 This flag is only set to 1 on Imprecise Data Aborts that are discarded in Debug state due to DSCR[19] being set to 1, and not other Imprecise Data Aborts occurring in Debug state.

This flag can be cleared to 0 only by writing to DRCR[2], see *Debug Run Control Register (DRCR)* on page 10-23.

Leaving Debug state with this flag set to 1 leads to UNPREDICTABLE behavior.

This flag is used to detect imprecise aborts generated by, or taken on, instructions issued to the processor whilst in Debug state, but that have been discarded because DSCR[19] is set to 1 (see *Imprecise Data Aborts discarded, bit [19]* on page 10-16). If DSCR[19] is 0, the imprecise Data Abort is acted upon on exit from Debug state.

The meanings of the values of bit [7] are as follows:

0 no imprecise Data Abort exception has been discarded since the last time this bit was cleared to 0

1 an imprecise Data Abort exception has been discarded since the last time this bit was cleared to 0.

See also *Imprecise Data Aborts and entry to Debug state* on page 4-4 and *Exceptions in Debug state* on page 4-19.

Copyright © 2006 ARM Limited. All rights reserved.

Sticky Undefined, bit [8]

v6 This flag is not defined in v6 Debug. DSCR[8] is Reserved and RAZ/SBZP.

v6.1 If the core is not in Debug state, this flag reads as zero.

This flag is cleared to 0 on reads to the DSCR by the external debugger.

v6.1, v7 This flag does not change when the processor is not in Debug state.

v7 This flag can be cleared to 0 only by writing to DRCR[2], see *Debug Run Control Register (DRCR)* on page 10-23.

Leaving Debug state with this flag set to 1 leads to UNPREDICTABLE behavior.

This flag is used to detect Undefined Instruction exceptions generated by instructions issued to the processor while in Debug state.

The meanings of the values of bit [0] are as follows:

0 no Undefined Instruction exception occurred since the last time this bit was cleared to 0

1 an Undefined Instruction exception occurred since the last time this bit was cleared to 0.

See also *Exceptions in Debug state* on page 4-19.

No Power-down (DBGNOPWRDWN), bit [9]

v6, v6.1 This bit is not defined in ARMv6, although many implementations define DSCR[9] as the No Power-down bit. If not implemented, DSCR[9] is Reserved and RAZ/SBZP.

v7 This bit is implemented in the Device Power-down and Reset Control Register (PRCR), see *No Power-down (DBGNOPWRDWN), bit [0]* on page 10-26. DSCR[9] is Reserved and RAZ/SBZP in ARMv7.

This bit has the same behavior as the No Power-down bit in the ARMv7 Device Power-down and Reset Control Register (PRCR) see *No Power-down (DBGNOPWRDWN), bit [0]* on page 10-26.

Force Debug Acknowledge (DbgAck), bit [10]

If this bit (DbgAck) is set to 1, the **DBGACK**, **DBGTRIGGER**, and **DBGCPUDONE** output signals are forced HIGH, regardless of the processor state. See *EDBGRQ, DBGTRIGGER, DBGCPUDONE and DBGACK* on page 6-4 for details of these signals.

If the external debugger needs to execute pieces of code in normal state as part of the debugging process, but needs the rest of the system to behave as if the processor is in Debug state, the external debugger must set this bit to 1.

Copyright © 2006 ARM Limited. All rights reserved.

Interrupts Disable (IntDis), bit [11]

If this bit (IntDis) is set to 1, the **IRQ** and **FIQ** input signals are inhibited:

0 interrupts enabled

1 interrupts disabled.

If the external debugger needs to execute pieces of code in normal state as part of the debugging process, but that code must not be interrupted, the external debugger must set this bit to 1.

For example, when single stepping code in a system with a periodic timer interrupt, the period of the interrupt is likely to be more frequent than the stepping frequency of the debugger. In this situation, if the debugger steps the target without setting DSCR[11] for the duration of the step, the interrupt is pending. This means that, if interrupts are enabled in the CPSR, the interrupt is taken as soon as the processor leaves Debug state.

DSCR[11] is ignored if DSCR[15:14] is set to b00 or **DBGEN** is LOW (that is, if DSCR[15:14] reads as b00), see *Halting Debug-mode enable, bit [14]* on page 10-15, *Monitor Debug-mode enable, bit [15]* on page 10-15, and *Authentication signals* on page 6-3.

User Mode access to Comms Channel disable, bit [12]

If this bit is 1 and a User Mode process tries to access the DIDR, INT-DSCR, INT-DTRRX, or INT-DTRTX through CP14 operations, the Undefined Instruction exception is taken.

Setting bit [12] to 1 prevents User Mode process access to any CP14 debug register.

The meanings of the values of bit [12] are:

0 User Mode access to Comms Channel enabled

1 User Mode access to Comms Channel disabled.

Execute ARM instruction enable, bit [13]

v6, v6.1 If the external debug interface does not have a mechanism for forcing the core to execute instructions in Debug state via the external debug interface, this bit always reads as zero and ignores writes.

v7 This bit, and the Instruction Transfer Register (the mechanism for forcing the core to execute instructions in Debug state) are required.

The meanings of the values of this bit are:

0 disabled

1 the mechanism for forcing the core to execute instructions in Debug state via the external debug interface is enabled.

Setting this bit to 1 when the core is not in Debug state leads to UNPREDICTABLE behavior.

 Copyright © 2006 ARM Limited. All rights reserved.

Halting Debug-mode enable, bit [14]

The meanings of the values of this bit are:

0 Halting Debug-mode disabled

1 Halting Debug-mode enabled.

Note

If the external interface input **DBGEN** is LOW, DSCR[14] reads as 0. The programmed value is masked until **DBGEN** is taken HIGH. When **DBGEN** goes HIGH, the value read and the behavior of the core correspond to the programmed value.

It is the programmed value of DSCR[14], not the value returned by reads of the DSCR, that is saved by the OS Save and Restore Register in a power-down sequence.

Monitor Debug-mode enable, bit [15]

The meanings of the values of this bit are as follows:

0 Monitor Debug-mode disabled

1 Monitor Debug-mode enabled.

Note

If Halting Debug-mode is enabled (bit [14] is set) then the monitor Debug-mode setting is disabled regardless of the setting of the Monitor Debug-mode enable bit.

If the external interface input **DBGEN** is LOW, DSCR[15] reads as 0. The programmed value is masked until **DBGEN** is taken HIGH. When **DBGEN** goes HIGH, the value read and the behavior of the core correspond to the programmed value.

It is the programmed value of DSCR[15], not the value returned by reads of the DSCR, that is saved by the OS Save and Restore Register in a power-down sequence.

Secure Privileged Invasive Debug Disabled (SPIDDIS), bit [16]

v6 This bit is not defined in v6 Debug. DSCR[16] is Reserved and RAZ/SBZP.

v6.1 If the processor implements Security Extensions, bit [16] takes the value of the inverse of the **SPIDEN** input. Otherwise it reads as zero.

v7 This bit is the inverse of bit [4] of the Authentication Status Register, see *Authentication Status Register (AUTHSTATUS)* on page 10-74.

Secure Privileged Non-invasive Debug Disabled (SPNIDDIS), bit [17]

v6 this bit is not defined in v6 Debug. DSCR[17] is Reserved and RAZ/SBZP

 Copyright © 2006 ARM Limited. All rights reserved.

v6.1 if the processor implements Security Extensions, this bit takes the value of the inverse of the **SPNIDEN** input. Otherwise it reads as zero.

v7 This bit is the inverse of bit [6] of the Authentication Status Register, see *Authentication Status Register (AUTHSTATUS)* on page 10-74.

Nonsecure world status (NS), bit [18]

If the processor implements Security Extensions, this bit indicates whether the processor is in the Secure world. The meanings of the values of the NS bit are as follows:

0 the processor is in the Secure world (SCR[0]=0 or the processor is in Monitor Mode)

1 the processor is not in the Secure world (SCR[0]=1 and the processor is not in Monitor Mode).

If the processor does not implement Security Extensions, this bit reads as zero.

Imprecise Data Aborts discarded, bit [19]

In v6 Debug, this flag is not defined. DSCR[19] is Reserved and RAZ/SBZP.

It is IMPLEMENTATION DEFINED whether this bit is automatically set to 1 on entry to Debug state:

- If the processor logic sets this bit to 1 on entry to Debug state then this bit is read-only.
- If this bit is not automatically set to 1 on entry to Debug state then it is set to 1 after execution of an IMPLEMENTATION DEFINED sequence of operations in Debug state. In this case it is IMPLEMENTATION DEFINED whether this bit is read-only or read/write:
  - this bit can only be read/write if the sequence of operations performed on entry to Debug state includes an explicit write of 1 to this bit
  - if the processor detects the sequence of operations performed on entry to Debug state and automatically sets DSCR[19] to 1b1 following that sequence then this bit must be read-only.

The bit is cleared to 0 on exit from Debug state.

If an imprecise Data Abort is signaled while this bit is set to 1, the core sets the Sticky Imprecise Abort bit to 1, but otherwise discards the abort (see *Sticky Imprecise Abort, bit [7]* on page 10-12).

See *Imprecise Data Aborts and entry to Debug state* on page 4-4 for details.

EXT-DTR Access Mode, bits [21:20]

In ARMv6, these flags are not defined. DSCR[21:20] are Reserved and RAZ/SBZP.

The EXT-DTR Access Mode can be used to optimize access to EXT-DTR from an External Debugger. The aim of the various modes is to cut down the number of interactions required through the external debug interface, which reduces the bandwidth required between the device and the debugger.

 Copyright © 2006 ARM Limited. All rights reserved.

Note

The EXT-DTR Access Mode controls all accesses made to EXT-DTRRX and EXT-DTRTX, and in some cases accesses to the Instruction Transfer Register (ITR). Those accesses might not necessarily be made by an External Debugger. See *Internal and external views of DSCR and DTR* on page 5-16.

Three modes of operation are provided, as follows:

Table 10-6 Meaning of Method of Debug Entry values

Value	Description	Section
b00	Non-blocking mode	*Non-blocking mode*
b01	Stall mode	*Stall mode* on page 10-18
b10	Fast mode	*Fast mode* on page 10-19
b11	Reserved	-

Note

Non-blocking mode is the default setting because improper use of the other modes can result in the external debug interface becoming deadlocked.

See *Instruction and data transfer registers* on page 10-32. The DTR External Access Mode flags have no effect on INT-DTRRX or INT-DTRTX accesses.

Non-blocking mode

When Non-blocking mode is selected, reads from EXT-DTRTX and writes to EXT-DTRRX and ITR are ignored if the appropriate latched *ready* flag was not in the *ready* state. These latched flags are updated on DSCR reads:

- if DTRRXfull_l is set to 1, writes to EXT-DTRRX are ignored
- if InstrCompl_l is set to 0, writes to ITR are ignored
- if DTRTXfull_l is set to 0, reads from EXT-DTRTX are ignored and return an UNPREDICTABLE value.

Following a successful write to EXT-DTRRX, DTRRXfull and DTRRXfull_l are set to 1.

Following a successful read from EXT-DTRTX DTRTXfull and DTRTXfull_l are cleared to 0.

Following a successful write to ITR, InstrCompl and InstrCompl_l are cleared to 0.

Copyright © 2006 ARM Limited. All rights reserved.

Note

If the access is made through the Memory-mapped interface and the Software Lock is set the registers are read-only and the flags remain unchanged. If the access is made through the external debug interface and the OS Lock or Sticky Power Down flag is set to 1, accesses to these registers generate a error response and the flags remain unchanged. See *Access permissions for External Debug and Memory-mapped interfaces* on page 5-37 for more information.

Debuggers accessing these registers must first read EXT-DSCR. This has the side-effect of copying DTRRXfull and DTRTXfull to DTRRXfull_l and DTRTXfull_l, and setting InstrCompl_l. The debugger can then:

- write to the EXT-DTRRX if the DTRRXfull flag was 0 (DTRRXfull_l is 0)
- read from the EXT-DTRTX if the DTRTXfull flag was 1 (DTRTXfull_l is 1)
- write to the ITR if the InstrCompl_l flag was 1.

However, debuggers can issue both actions together and later determine from the read EXT-DSCR value whether the read/write went ahead.

Stall mode

When Stall mode is selected, accesses to EXT-DTRRX, EXT-DTRTX, and ITR are modified such that each access stalls under the following conditions:

- writes to EXT-DTRRX are not completed until DTRRXfull is 0
- writes to ITR are not completed until InstrCompl is 1
- reads from EXT-DTRTX are not completed until DTRTXfull is 1.

If an access is stalled in this way you cannot access any of the debug registers until the stalled EXT-DTRRX, EXT-DTRTX, or ITR access completes.

Note

The mechanism by which an access is stalled by the external debug interface must be defined by the external debug interface. For details of how accesses are stalled by the recommended ARM Debug Interface v5, see the *ARM Debug Interface v5 Architecture Specification*.

Following a write to EXT-DTRRX or ITR, or a read from EXT-DTRTX, the flags InstrCompl, InstrCompl_l, DTRRXfull, DTRXfull_l, DTRTXfull, and DTRTXfull_l are set as in *Non-blocking mode* on page 10-17.

Note

The rules used in Non-blocking mode for ignoring accesses based on the values of the latched flags InstrCompl_l, DTRRXfull_l and DTRTXfull_l do not apply in Stall mode.

Copyright © 2006 ARM Limited. All rights reserved.

Stall mode can be selected when the processor is not in Debug state. However, because Stall mode blocks the interface to the DTR until the core issues the correct `MCR` or `MRC` instruction to unblock the access, it is not recommended to use Stall mode in cases where the External Debugger does not have complete control over the instructions executing on the processor.

Accesses to EXT-DTRRX and EXT-DTRTX through the Extended CP14 Interface are UNPREDICTABLE if Stall mode is selected.

Fast mode

If Fast mode is selected and the Execute ARM Instruction Enable bit is 0, or the processor is not in Debug state, the results are UNPREDICTABLE.

When Fast mode is selected, writes to the ITR register do not trigger an instruction for execution. Instead, the instruction is latched. Accesses to EXT-DTRRX and EXT-DTRTX through the Extended CP14 Interface are UNPREDICTABLE. Other reads of EXT-DTRTX and writes to EXT-DTRRX cause the latched instruction to be executed by the processor.

This allows a single instruction to be executed repeatedly without reloading the ITR.

In summary:

- Writes to ITR do not trigger an instruction to be executed. If a previously issued instruction is executing, it must not be affected by the write to the ITR. Implementations can choose to stall the write until InstrCompl is set to 1 to achieve this requirement.
- External access writes to EXT-DTRRX:
  - are not completed until InstrCompl is set to 1
  - write the data to the DTRRX register
  - issue the instruction last written to ITR in time for it to read the data written to the EXT-DTRRX, if it is an instruction that reads the INT-DTRRX.

  If DTRRXfull is set to 1 before the write, after the write the values of DTRRX and the DTRRXfull and DTRRXfull_l flags in the DSCR are UNPREDICTABLE.
- Reads from EXT-DTRTX:
  - Are not completed until InstrCompl is set to 1.
  - Return the data from the DTRTX.
  - Issue the instruction last written to the ITR in time for it to write a new value to the DTRTX, if it is an instruction that writes to the INT-DTRTX, without affecting the data returned from this read of the EXT-DTRTX. (That is, this instruction can write the next DTRTX value to be read.)

  If DTRTXfull is set to 0 before the read, after the read the values of DTRTX and the DTRTXfull and DTRTXfull_l flags in the DSCR are UNPREDICTABLE.

If a Fast mode access is stalled you cannot access any of the debug registers until the stalled EXT-DTRRX, EXT-DTRTX, or ITR access completes.

Copyright © 2006 ARM Limited. All rights reserved.

Note

The rules used in Non-blocking mode for ignoring accesses based on the values of the latched flags InstrCompl_l, DTRRXfull_l and DTRTXfull_l do not apply in Fast mode.

If DSCR[6] (Sticky Precise Abort) is set to 1, reads of EXT-DTRTX and writes to EXT-DTRTX do not cause the latched instruction to be executed by the processor, and the access completes. In these cases:

- a read of EXT-DTRTX returns an UNPREDICTABLE value, and the values of DTRTX and the DTRTXfull and DTRTXfull_l flags become UNPREDICTABLE
- if you write to EXT-DTRRX, the values of DTRRX and the DTRRXfull and DTRRXfull_l flags in the DSCR become UNPREDICTABLE.

Otherwise, following a write to EXT-DTRRX or ITR, or a read from EXT-DTRTX, the flags InstrCompl, InstrComp_l, DTRRXfull, DTRRXfull_l, DTRTXfull, and DTRTXfull_l are set as in *Non-blocking mode* on page 10-17.

Latched Instruction Complete (InstrCompl_l), bit [24]

In v6 and v6.1 Debug, InstrCompl and InstrCompl_l are not defined. DSCR[24] is Reserved and RAZ/SBZP.

InstrCompl_l is a copy of the internal Instruction Complete flag (InstrCompl), taken on reads of EXT-DSCR. InstrCompl signals whether the processor has completed execution of an instruction issued through the Instruction Transfer Register (ITR).

Normally, InstrCompl is cleared to 0 following issue of an instruction through ITR, and InstrCompl becomes 1 once the instruction completes. InstrCompl is set to 1 on entry to Debug state. For more information about the behavior of InstrCompl, InstrCompl_l and the ITR register, see:

- *Instruction Transfer Register (ITR)* on page 10-37
- *Host to Target Data Transfer Register (DTRRX)* on page 10-32
- *Target to Host Data Transfer Register (DTRTX)* on page 10-35.

On reads of EXT-DSCR, InstrCompl_l always returns the current value of InstrCompl.

The meanings of the values of InstrCompl are as follows:

0 an instruction previously issued through the ITR has not completed its changes to the architectural state of the processor

1 all instructions previously issued through the ITR have completed their changes to the architectural state of the processor.

If InstrCompl_l reads as 0, a subsequent write to the ITR register is ignored unless DSCR[20] != b00 (Stall mode or Fast mode selected).

If the processor is not in Debug state, the value read for this flag is UNPREDICTABLE. The value for this flag when read through the Baseline CP14 Interface is always UNPREDICTABLE.

 Copyright © 2006 ARM Limited. All rights reserved.

Sticky Pipeline Advance (PipeAdv), bit [25]

This flag is not defined in v6 and v6.1 Debug. DSCR[25] is Reserved and RAZ/SBZP.

This bit is set to 1 every time the processor pipeline retires one instruction. It is cleared to 0 by a write to DRCR[3], see *Debug Run Control Register (DRCR)* on page 10-23.

The purpose is to allow the debugger to detect that the processor is idle. In some situations this might mean that the processor is deadlocked.

Latched DTRTXfull (DTRTXfull_l) and Latched DTRRXfull (DTRRXfull_l), bits [27:26]

These flags are not defined in ARMv6. DSCR[27:26] is Reserved and RAZ/SBZP.

Bit 27 is the Latched DTRRXfull (DTRRXfull_l) bit.

Bit 26 is the Latched DTRTXfull (DTRTXfull_l) bit.

The DTRTXfull_l and DTRRXfull_l bits are, respectively, copies of the DTRTXfull and DTRRXfull bits, taken on read of EXT-DSCR. That is, they represent the last values of DTRTXfull and DTRRXfull read through EXT-DSCR.

On reads of EXT-DSCR, DTRRXfull_l and DTRTXfull_l always read the same values as DTRRXfull and DTRTXfull respectively.

Normally:

- DTRTXfull_l is cleared to 0 on reads of EXT-DTRTX
- DTRRXfull_l is set to 1 on writes of EXT-DTRRX.

On reads of INT-DSCR, the values read for these bits are UNPREDICTABLE.

The latched versions of the flags control the processor behavior on reads of EXT-DTRTX and writes to EXT-DTRRX. For more information about the behavior of the DTRRX and DTRTX registers see *Host to Target Data Transfer Register (DTRRX)* on page 10-32 and *Target to Host Data Transfer Register (DTRTX)* on page 10-35.

DTRTX register full (DTRTXfull), bit [29]

The meanings of the values of DTRTXfull are as follows:

0 DTRTX register empty

1 DTRTX register full.

Normally, DTRTXfull is:

- cleared to 0 on reads of EXT-DTRTX
- set to 1 on writes to INT-DTRTX.

For more information about the behavior of DTRTXfull and the DTRTX register see *Target to Host Data Transfer Register (DTRTX)* on page 10-35.

Copyright © 2006 ARM Limited. All rights reserved.

DTRRX register full (DTRRXfull), bit [30]

The meanings of the values of DTRRXfull are as follows:

0 DTRRX register empty

1 DTRRX register full.

Normally, DTRRXfull is:

- set to 1 on writes to EXT-DTRRX
- cleared to 0 on reads of INT-DTRRX.

For more information about the behavior of DTRRXfull and the DTRRX register see *Host to Target Data Transfer Register (DTRRX)* on page 10-32.

10.2.2 Watchpoint Fault Address Register (WFAR)

The WFAR is register 6, at offset `0x018`.

v6 Debug In v6 Debug, the WFAR can only be accessed through CP15.

v6.1 Debug In v6.1 Debug, the WFAR can be accessed through CP14, and the CP15 access is deprecated.

ARM v7 In ARMv7, the WFAR encoding in CP15 is UNDEFINED in User Mode and UNPREDICTABLE in privileged modes.

On every Watchpoint Debug event the WFAR is updated with the *Instruction Virtual Address* (IVA) of the instruction that accessed the watchpointed address plus an offset that depends on the processor state:

- 8 if the processor was in ARM state
- 4 if the processor was in Thumb or ThumbEE state
- an IMPLEMENTATION DEFINED offset if the processor was in Jazelle state.

See *Memory addresses* on page 2-13 for a definition of the IVA used to update the WFAR.

Table 10-7 Watchpoint Fault Address Register bit definition

Bits	Access	Debug reset value	Description
[31:0]	RW	UNPREDICTABLE	Address of the watchpointed instruction plus an offset that depends on the processor state

Copyright © 2006 ARM Limited. All rights reserved.

10.2.3 Debug Run Control Register (DRCR)

The DRCR is register 36, at offset `0x090`.

Table 10-8 shows the layout of the Debug Run Control Register.

This register is not defined in ARMv6.

The main purpose of the run-control register is to request the processor to enter or leave Debug state. It is also used to clear to 0 the sticky exception bits in the DSCR.

Table 10-8 Debug Run Control Register bit definition

Bits	Access	Description
[31:5]	RAZ/SBZP	Reserved
[4]	RAZ/W	*Cancel BIU Requests, bit [4]* on page 10-24
[3]	RAZ/W	*Clear Sticky Pipeline Advance, bit [3]* on page 10-24
[2]	RAZ/W	*Clear Sticky Exceptions, bit [2]* on page 10-24
[1]	RAZ/W	*Restart Request, bit [1]*
[0]	RAZ/W	*Halt Request, bit [0]*

Halt Request, bit [0]

The actions on writing Halt Request values are as follows:

0 no action

1 request entry to Debug state.

Writing 1 to this bit requests that the processor enters Debug state. This request is held until the Debug state entry occurs, see *Halting Debug events* on page 2-18.

Writing 0 has no effect.

Once the request has been made, the debugger polls DSCR[0] until it reads 1.

This bit always reads as 0. Writes are ignored if the processor is already in Debug state.

Restart Request, bit [1]

The actions on writing Restart Request values are as follows:

0 no action

1 request exit from Debug state.

Writing 1 to this bit requests that the processor leaves Debug state. This request is held until the processor exits Debug state.

 Copyright © 2006 ARM Limited. All rights reserved.

Writing 0 has no effect.

Once the request has been made, the debugger polls DSCR[1] until it reads 1.

This bit always reads as 0. Writes are ignored if the processor is not in Debug state.

Clear Sticky Exceptions, bit [2]

The actions on writing Clear Sticky Exceptions values are as follows:

0 no action

1 clear DSCR[8:6] to b000.

When the processor is not in Debug state, it is UNPREDICTABLE whether a write of 1 to DRCR[2] clears DSCR[8:6] to b000.

When the processor is in Debug state, a request to clear DSCR[8:6] combined with a restart request, DRCR[1], in a single write of DRCR with DRCR[2:1] = b11, clears DSCR[8:6] to b000 before leaving Debug state.

Clear Sticky Pipeline Advance, bit [3]

The actions on writing Clear Sticky Pipeline Advance values are as follows:

0 no action

1 clear DSCR[25] to 0.

When the processor is powered down, it is UNPREDICTABLE whether a write of 1 to DRCR[3] clears DSCR[25] to 0.

Cancel BIU Requests, bit [4]

The actions on writing Cancel BIU Requests are as follows:

0 no action

1 cancel pending transactions.

It is IMPLEMENTATION DEFINED whether this feature is supported. If this feature is not implemented, writes to this bit are ignored.

When support for Cancel BIU Requests is implemented, if 1 is written to this bit, the processor cancels any pending transactions (Bus Interface Unit Requests) on the system bus until Debug state is entered (that is, Debug state entry is the acknowledge event that clears this request). An implementation must abandon all data load and store transactions; it is IMPLEMENTATION DEFINED whether other transactions, including instruction fetches and cache operations, are also abandoned.

Abandoned transactions have the following behavior:

- abandoned data stores write an UNPREDICTABLE value to the target address
- abandoned data loads return an UNPREDICTABLE value to the register bank
- abandoned instruction fetches return an UNPREDICTABLE instruction for execution
- abandoned cache operations leave the memory system in an UNPREDICTABLE state.

Copyright © 2006 ARM Limited. All rights reserved.

However, an abandoned transaction does not cause any exception. Additional BIU requests, after Debug state has been entered, have an UNPREDICTABLE behavior.

The number of ports on the processor and their protocols are implementation specific and, therefore, the detailed behavior of this bit is IMPLEMENTATION DEFINED. It is also IMPLEMENTATION DEFINED whether this behavior is supported on all ports of a processor. For example, an implementation can choose not to implement this behavior on instruction fetches.

The purpose of this control bit is to allow the debugger to release a deadlock on the system bus so Debug state can be entered. This Debug state entry does not need to be recoverable, because the debugger only wants to know what the state of the processor was at the time the deadlock occurred. A Halt Request (DRCR[0]) or External Debug Request (**EDBGRQ**) must be pending at the time the deadlock is released.

If a Debug state entry occurs, the PC reads as if the cancelled transactions completed successfully.

Note

It might not be easy to infer the cause of the deadlock from the PC value if, for example, the processor has a non-blocking cache design or a write buffer, or if the deadlocked transaction corresponded to a load to the PC.

If the processor implements Security Extensions, a write to this bit only takes effect if **DBGEN** and **SPIDEN** are HIGH, meaning that invasive debug is allowed in all processor states and modes.

If the processor does not implement Security Extensions, a write to this bit is ignored unless **DBGEN** is HIGH.

See *Authentication signals* on page 6-3 for details of **DBGEN** and **SPIDEN**.

It is UNPREDICTABLE whether a write of 1 to DRCR[4] has any effect when the processor is powered-down.

10.2.4 Device Power-Down and Reset Control Register (PRCR)

The PRCR is register 196, at offset `0x310`, and is not defined in ARMv6.

Table 10-9 shows the layout of the PRCR in ARMv7.

The PRCR controls reset and power-down related functionality.

Table 10-9 Device Power-down & Reset Control Register bit definition

Bits	Access	Debug reset value	Description
[31:3]	RAZ/SBZP	-	Reserved
[2]	RW	0	*Hold Internal Reset, bit [2]* on page 10-27
[1]	RAZ/W	0	*Force Internal Reset, bit [1]* on page 10-26
[0]	RW	0	*No Power-down (DBGNOPWRDWN), bit [0]* on page 10-26

Copyright © 2006 ARM Limited. All rights reserved.

No Power-down (DBGNOPWRDWN), bit [0]

The actions on writing No Power-down values are as follows:

0 **DBGNOPWRDWN** LOW

1 **DBGNOPWRDWN** HIGH.

DBGNOPWRDWN is an IMPLEMENTATION DEFINED feature. If it is implemented, setting this flag requests the power controller to work in an emulation mode where the core is not actually powered down when requested.

See *DBGNOPWRDWN* on page 6-7 for details.

Force Internal Reset, bit [1]

The actions on writing Force Internal Reset values are as follows:

0 no action

1 force internal reset.

Note

- Force Internal Reset is an IMPLEMENTATION DEFINED feature. If an implementation does not support Force Internal Reset then bit [1] ignores writes.
- This bit always reads as zero. The sticky reset status bit in the Device Power-Down and Reset Status Register must be used to read the current reset status of the processor, see *Sticky Reset Status, bit [3]* on page 10-30.

The external debugger can use this bit to force the processor into reset if it does not have access to the **nRESET** input. The reset behavior is the same as soft reset (**nRESET**). It does not cause power-down.

If the processor implements Security Extensions, a write to this bit is ignored unless both the external debug interface signals **DBGEN** and **SPIDEN** are HIGH, meaning that invasive debug is allowed in all processor states and modes.

If the processor does not implement Security Extensions, a write is ignored unless **DBGEN** is HIGH.

See *Authentication signals* on page 6-3 for details of **DBGEN** and **SPIDEN**.

Unless Hold Internal Reset (bit [2]) is set to 1, the internal reset is only held for long enough to reset the core. The bit then self-clears to 0 and the core leaves the reset state.

Note

If an implementation supports both features, the Force Internal Reset and Hold Internal Reset bits can be set to 1 in a single write to the PRCR. In this case the core must enter reset and be held there.

Copyright © 2006 ARM Limited. All rights reserved.

Hold Internal Reset, bit [2]

The actions on writing Hold Internal Reset values are as follows:

0 do not hold internal reset on power-up or warm reset

1 hold processor's non-debug logic in reset on power-up or warm reset until this flag is cleared to 0.

Note

- Hold Internal Reset is an IMPLEMENTATION DEFINED feature. If an implementation does not support Hold Internal Reset then bit [2] is Read-as-zero and ignores writes.
- When Hold Internal Reset is implemented this bit cannot work on system power-up, because it resets to 0.

In ARMv7 the primary purpose of this bit is to avoid the processor running again before the debugger has had the chance to detect a power-down occurrence and restore the state of the debug registers inside the core power domain.

This bit also causes the internal **nRESET** signal to be held on warm **nRESET** or a Force Internal Reset request, that is, on a regular reset of the processor's non-debug logic, not related to core power-down. It can therefore be used in conjunction with an external reset controller to take the processor into reset and hold it there while letting the rest of the system come out of reset.

The bit can be written at the same time as a Force Internal Reset request to force the core into reset and hold it there, for example while programming other debug registers such as programming a Halt Debug Request to take the processor into Debug state on leaving Reset. See *Halt Request, bit [0]* on page 10-23 for details.

Note

The core is not held in Debug state, and cannot enter Debug state until released from reset. The processor must not accept instructions issued via the Instruction Transfer Register (ITR) while held in reset.

The effect of this bit depends on the state of the external debug interface signals:

- If the processor implements Security Extensions, this bit only takes effect if both of the external debug interface signals **DBGEN** and **SPIDEN** are HIGH, meaning that invasive debug is allowed in all processor states and modes.
- If the processor does not implement Security Extensions, this bit only takes effect if the external debug interface signal **DBGEN** is HIGH.

See *Authentication signals* on page 6-3 for details of **DBGEN** and **SPIDEN**.

The debugger can distinguish between a held power-down occurrence and a held warm **nRESET** by examining the Device Power-Down and Reset Status Register (PRSR).

Copyright © 2006 ARM Limited. All rights reserved.

10.2.5 Device Power-Down and Reset Status Register (PRSR)

The PRSR is register 197, at offset `0x314`, and is not defined in ARMv6.

SinglePower: in ARMv7, if only a single power-domain is implemented, bits [1:0] of this register read as b01.

This register gives the information about the reset and power-down state of the processor.

Table 10-10 shows the layout of the PRSR.

Table 10-10 Device Power-down & Reset Control Register bit definition

Bits	Access	Debug reset value	Description
[31:4]	RAZ/SBZP	-	Reserved
[3]	RC	0 [a]	*Sticky Reset Status, bit [3]* on page 10-30
[2]	RO	[b]	*Reset Status, bit [2]* on page 10-29
[1]	RC	1	*Sticky Power-down Status, bit [1]* on page 10-29
[0]	RO	[b]	*Power-up Status, bit [0]*

a. If both the debug logic and core are reset at the same time (**PRESETDBGn** and **nRESET**), the value of the Sticky Reset Status flag after reset is UNPREDICTABLE.

b. Bits [2,0] are status bits. On read they report the current status of the processor.

Power-up Status, bit [0]

The meaning of the Power-up Status values are as follows:

0 the processor is powered-down

1 the processor is powered-up.

The Power-up Status bit reads the value of the **DBGPWRDUP** input on the external debug interface. See *DBGPWRDUP* on page 6-7 for details of the **DBGPWRDUP** input.

The processor is in the powered-up state when **DBGPWRDUP** is HIGH, and is in the powered-down state when **DBGPWRDUP** is LOW.

Power-up Status is not affected by the reset state of the processor, whether that reset is:

- a power-up reset, **nCOREPORESET**
- an internal reset, **nRESET**
- a reset occurring because the Hold Internal Reset bit in the Device Power-Down and Reset Control Register (PRCR) is set to 1.

 Copyright © 2006 ARM Limited. All rights reserved.

Reads of PRSR made when the processor is in the powered up state return 1 for Power-up Status.

Reads of PRSR made when the processor is in the powered down state return 0 for Power-up Status.

For more information see *Power domains and debug* on page 5-6.

Sticky Power-down Status, bit [1]

The meaning of Sticky Power-down Status values are as follows:

0 the processor has not been powered-down since the last time this register was read

1 the processor has powered-down since the last time this register was read. Cleared to 0 on reading this register.

Note

Powered-down is defined in *Power-up Status, bit [0]* on page 10-28.

When the processor is in the powered-down state, the Sticky Power-down Status bit is set to 1.

Reads of PRSR made when the processor is in the powered down state return 1 for Sticky Power-down Status and do not change the value of Sticky Power-down Status.

Note

Bits [1:0] of the PRSR never read as b00.

Reads of PRSR made when the processor is in the powered up state return the current value of Sticky Power-down Status, and then clear Sticky Power-down Status to 0.

Note

The bit is not cleared to 0 if the read of the PRSR was made through the Memory-mapped interface when the Software Lock is set. See *Permissions in relation to locks* on page 5-21 for more information on the Software Lock.

If this bit is set to 1, accesses to certain registers return an error response. See *Permissions in relation to power-down* on page 5-22 for more information.

Reset Status, bit [2]

The meaning of Reset Status values are as follows:

0 the processor is not currently held in reset state

1 the processor is currently held in reset state.

The processor enters reset state following the assertion of one or both of:

- the internal or warm reset input, **nRESET**, asserted LOW
- the power-up reset input, **nCOREPORESET**, asserted LOW

Copyright © 2006 ARM Limited. All rights reserved.

The processor stops executing instructions before it enters reset state.

The processor remains in reset state until:

- both **nRESET** and **nCOREPORESET** are deasserted HIGH
- the Hold Internal Reset request bit in the Device Power-Down and Reset Control Register (PRCR) is 0.

The processor then resumes execution of instructions with the Reset exception.

Reads of the PRSR made when the processor is in reset state return 1 for the Reset Status.

Reads of the PRSR made when the processor is not in reset state return 0 for the Reset Status.

Sticky Reset Status, bit [3]

The meaning of Sticky Reset Status values are as follows:

0 the processor has not been reset since the last time this register was read

1 the processor has been reset since the last time this register was read.

Note

Reset state is defined in *Reset Status, bit [2]* on page 10-29.

When the processor is in reset state, the Sticky Reset Status bit is set to 1.

Reads of PRSR made when the processor is in reset state return 1 for Sticky Reset Status.

Reads of PRSR made when the processor is not in reset state return the current value of Sticky Reset Status, and then clear Sticky Reset Status to 0.

Note

- The Sticky Reset Status bit is not cleared to 0 if the read of the PRSR is made through the Memory-mapped interface when the Software Lock is set. See *Permissions in relation to locks* on page 5-21 for more information on the Software Lock.
- Bits [3:2] of PRSR never read as b01.

Copyright © 2006 ARM Limited. All rights reserved.

10.2.6 Program Counter Sampling Register (PSCR)

The PSCR is register 33, at offset `0x084`.

v6, v6.1 This register is not defined in ARMv6. However, it might form part of the external debug interface.

v7 It is IMPLEMENTATION DEFINED whether the PCSR is implemented. If the PCSR is not implemented, reads of the PCSR return an UNPREDICTABLE value.

Reads through the Extended CP14 Interface of the CP14 register that maps to the PCSR return an UNPREDICTABLE value.

In ARMv7, writes to the PCSR write to the Instruction Transfer Register. See *Instruction Transfer Register (ITR)* on page 10-37. In ARMv6 the PCSR, if implemented, is read-only.

For more information about the PCSR, see *Program counter sampling register* on page 8-3. See also *Program Counter Sampling Register implemented, bit [13]* on page 10-4.

Table 10-11 shows the layout of the PCSR.

Table 10-11 Program Counter Sampler Register bit definition

Bits	Access	Debug reset value	Description
[31:2]	R	-	Program Counter Sample value
[1:0]	R	-	*Meaning of PC Sample Value, bits [1:0]*

Meaning of PC Sample Value, bits [1:0]

The value sampled through this register is the *Instruction Virtual Address* (IVA) of the instruction plus an offset that depends on the processor state. The bottom two bits of the sampled value encode the processor state so the profiling tool can work out the IVA by subtracting the offset. See *Memory addresses* on page 2-13 for a definition of the IVA read through the PCSR.

The meaning of Meaning of PC Sample Value values are as follows:

b00 ((PCSR[31:2] << 2) - 8) references an ARM state instruction

bx1 ((PCSR[31:1] << 1) - 4) references a Thumb or ThumbEE state instruction

b10 IMPLEMENTATION DEFINED.

Copyright © 2006 ARM Limited. All rights reserved.

10.3 Instruction and data transfer registers

This section contains the following subsections:

- *Host to Target Data Transfer Register (DTRRX)*
- *Target to Host Data Transfer Register (DTRTX)* on page 10-35
- *Instruction Transfer Register (ITR)* on page 10-37.

The following registers and flags form the Debug Communications Channel:

- the DTRRX register, see *Host to Target Data Transfer Register (DTRRX)*
- the DTRTX register, see *Target to Host Data Transfer Register (DTRTX)* on page 10-35
- the DTRRXfull flag, see *DTRRX register full (DTRRXfull), bit [30]* on page 10-22
- the DTRTXfull_l and DTRRXfull_l flags, see *Latched DTRTXfull (DTRTXfull_l) and Latched DTRRXfull (DTRRXfull_l), bits [27:26]* on page 10-21.

10.3.1 Host to Target Data Transfer Register (DTRRX)

The DTRRX is register 32, at offset `0x080`.

v6, v6.1 DTRRX was previously named rDTR. EXT-DTRRX is not defined in ARMv6. However, the functionality must be implemented as part of the external debug interface.

v7 The Extended CP14 Interface instructions that access EXT-DTRRX, if implemented, are UNPREDICTABLE in Debug state. See *Internal and external views of DSCR and DTR* on page 5-16 and *Extended CP14 interface* on page 5-25 for more details.

Table 10-12 shows the layout of the DTRRX.

Table 10-12 Host to Target Data Transfer Register bit definition

Bits	Debug reset value	Description
[31:0]	UNPREDICTABLE	Host to target data

In all Debug versions there are two views of DTRRX:

- INT-DTRRX, the internal view
- EXT-DTRRX, the external view.

See *Internal and external views of DSCR and DTR* on page 5-16 for definitions of the internal and external views.

 Copyright © 2006 ARM Limited. All rights reserved.

The behavior on various accesses to the DTRRX is described in the following tables:

- Table 10-13 shows the behavior of accesses to INT-DTRRX
- Table 10-14 shows the behavior of read accesses to EXT-DTRRX
- Table 10-15 on page 10-34 shows the behavior of write accesses to EXT-DTRRX.

Table 10-13 Behavior of accesses to INT-DTRRX

Access	DTRRXfull	Action	New DTRRXfull
Read	0	Returns an UNPREDICTABLE value.	Unchanged
	1	Returns DTRRX contents	0
Write	X	Not possible. There is no operation that writes to INT-DTRRX	-

Note

- If the `STC` instruction that reads INT-DTRRX aborts, the value in DTRRX and the value of the DTRRXfull flag are UNPREDICTABLE.
- The behavior on accesses to INT-DTRRX does not depend on the value of DTRRXfull_l, and accesses to INT-DTRTXfull do not update the value of DTRRXfull_l.

Table 10-14 Behavior of read accesses to EXT-DTRRX

Access mode[a]	Flag[b]	Flag value	Action	New DTRRXfull	New DTRRXfull_l
X	DTRRXfull	0	Returns an UNPREDICTABLE value.	Unchanged	Unchanged
		1	Returns DTRRX contents	Unchanged	Unchanged

a. For more information see *EXT-DTR Access Mode, bits [21:20]* on page 10-16.

b. This column indicates which of the DTRTXfull, DTRTXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-15 Behavior of write accesses to EXT-DTRRX

Access mode[a]	Flag[b]	Flag value	Action	New DTRRXfull	New DTRRXfull_l
Non-blocking	DTRRXfull_l	0	Writes to DTRRX[c].	1[c]	1[c]
		1	Write is ignored.	Unchanged	Unchanged
Stall	DTRRXfull	0	Writes to DTRRX[c].	1[c]	1[c]
		1	Stall until DTRRXfull = 0	-	-
Fast	InstrCompl	0	Stall until (InstrCompl = 1)	-	-
		1	Writes to DTRRX[c, d] and issue the instruction from the ITR[c, e]	1[c]	1[c]

a. For more information see *EXT-DTR Access Mode, bits [21:20]* on page 10-16.
b. This column indicates which of the DTRTXfull, DTRTXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.
c. If the write is made through the Memory-mapped interface and the Software Lock is set, the registers are read-only and DTRRX, DTRRXfull, DTRRXfull_l, InstrCompl and InstrCompl_l are unchanged. No instruction is issued in Fast Mode. For more information see *Permission summaries for Memory-mapped and external debug interface* on page 5-36.
d. If DTRRXfull is 1, the values of DTRRX, DTRRXfull, and DTRRXfull_l become UNPREDICTABLE.
e. If DSCR[6], the Sticky Precise Data Abort bit, is set to 1, the instruction is not issued. InstrCompl and InstrCompl_l are unchanged, and the values of DTRRX, DTRRXfull and DTRRXfull_l become UNPREDICTABLE. For more information see *Sticky Precise Abort, bit [6]* on page 10-11.
Otherwise, the instruction is issued and InstrCompl and InstrCompl_l are cleared to 0.

Copyright © 2006 ARM Limited. All rights reserved.

10.3.2 Target to Host Data Transfer Register (DTRTX)

The DTRTX is register 35, at offset `0x08C`.

v6, v6.1 DTRTX was previously named wDTR. EXT-DTRTX is not defined in ARMv6. However, the functionality must be implemented as part of the external debug interface.

v7 The Extended CP14 Interface instructions that access EXT-DTRTX, if implemented, are UNPREDICTABLE in Debug state. See *Internal and external views of DSCR and DTR* on page 5-16 and *Extended CP14 interface* on page 5-25 for more details.

Table 10-16 shows the layout of the DTRTX.

Table 10-16 Target to Host Data Transfer Register bit definition

Bits	Debug reset value	Description
[31:0]	UNPREDICTABLE	Target to host data

In all Debug versions there are two views of DTRTX:

- INT-DTRTX, the internal view
- EXT-DTRTX, the external view.

See *Internal and external views of DSCR and DTR* on page 5-16 for definitions of the internal and external views.

The behavior on various accesses to the DTRTX is described in the following tables:

- Table 10-17 shows the behavior of accesses to INT-DTRTX
- Table 10-18 on page 10-36 shows the behavior of read accesses to EXT-DTRTX
- Table 10-19 on page 10-37 shows the behavior of write accesses to EXT-DTRTX.

Table 10-17 Behavior of accesses to INT-DTRTX

Access	DTRTXfull	Action	New DTRTXfull
Read	X	Not possible. There is no operation that reads from INT-DTRTX.	-
Write	0	Writes value to DTRTX.	1
	1	UNPREDICTABLE.	Unchanged

Copyright © 2006 ARM Limited. All rights reserved.

Note

- If the LDC instruction that writes to INT-DTRTX aborts, the value in DTRTX and the value of the DTRTXfull flag are UNPREDICTABLE.
- The behavior on accesses to INT-DTRTX does not depend on the value of DTRTXfull_l, and accesses to INT-DTRTXfull do not update the value of DTRTXfull_l.

Table 10-18 Behavior of read accesses to EXT-DTRTX

Access mode[a]	Flag[b]	Flag value	Action	New DTRTXfull	New DTRTXfull_l
Non-blocking	DTRTXfull_l	0	Returns UNPREDICTABLE value.	Unchanged	Unchanged
		1	Returns DTRTX contents.	0[c]	0[c]
Stall	DTRTXfull	0	Stalls until DTRTXfull = 1.	-	-
		1	Returns DTRTX contents.	0[c]	0[c]
Fast	InstrCompl	0	Stalls until (InstrCompl = 1).	-	-
		1	Returns DTRTX contents[d]. Issue the instruction in the ITR[c, e].	0[c]	0[c]

a. For more information see *EXT-DTR Access Mode, bits [21:20]* on page 10-16.
b. This column indicates which of the DTRTXfull, DTRTXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.
c. If the read is made through the Memory-mapped interface and the Software Lock is set, the registers are read-only and DTRRXfull, DTRRXfull_l, InstrCompl and InstrCompl_l remain unchanged. No instruction is issued in Fast Mode. For more information see *Permission summaries for Memory-mapped and external debug interface* on page 5-36.
d. If DTRTXfull is 0, this returns an UNPREDICTABLE value and the values of DTRTX, DTRTXfull and DTRTXfull_l become UNPREDICTABLE.
e. The value returned is the value of DTRTX before the instruction issued modifies the state of the processor.
If DSCR[6], the Sticky Precise Data Abort bit, is set to 1, the instruction is not issued, InstrCompl and InstrCompl_l remain unchanged, and the values of DTRTXfull and DTRTXfull_l become unpredictable. For more information see *Sticky Precise Abort, bit [6]* on page 10-11.
Otherwise, the instruction is issued and InstrCompl and InstrCompl_l are cleared to 0.

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-19 Behavior of write accesses to EXT-DTRTX

Access mode[a]	Flag[b]	Flag value	Action	New DTRTXfull and DTRTXfull_l
X	X	X	Updates DTRTX value[c].	Unchanged

a. For more information see *EXT-DTR Access Mode, bits [21:20]* on page 10-16.
b. This column indicates which of the DTRTXfull, DTRTXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.
c. In the event of a race condition with writes to both INT-DTRTX and EXT-DTRTX occurring, the result is UNPREDICTABLE. Writes to EXT-DTRTX must only be performed under controlled circumstances, for example when the core is in Debug state.

10.3.3 Instruction Transfer Register (ITR)

The ITR is register 33, at offset `0x084`.

v6, v6.1 ITR is not defined in ARMv6. However, it might have formed part of the external debug interface.

v7 Writes through the Extended CP14 Interface of the CP14 register that maps to the ITR are always UNPREDICTABLE.

The Instruction Transfer Register (ITR) allows the external debugger to feed ARM instructions into the core for execution while in Debug state. The ITR is write-only.

Table 10-20 shows the layout of the ITR.

Table 10-20 Instruction Transfer Register bit definition

Bits	Access	Debug reset value	Description
[31:0]	W	UNPREDICTABLE	ARM instruction to execute on the core

Writes to the ITR are UNPREDICTABLE when:

- not in Debug state
- DSCR[13], Execute ARM instructions enable, is set to 0.

 Copyright © 2006 ARM Limited. All rights reserved.

Table 10-21 shows the behavior of writes to the ITR when in Debug state with DSCR[13] set to 1.

Table 10-21 Behavior of write accesses to ITR

Access mode[a]	Flag[b]	Flag value	Action	New InstrCompl	New InstrCompl_l
Non-blocking	InstrCompl_l	0	Write is ignored.	Unchanged	Unchanged
		1	Issue instruction[c].	0[c]	0[c]
Stall	InstrCompl	0	Stall until (InstrCompl = 0).	-	-
		1	Issue instruction[c].	0[c]	0[c]
Fast	Not applicable	-	Save instruction in ITR[d].	-	-

a. For more information see *EXT-DTR Access Mode, bits [21:20]* on page 10-16.

b. This column indicates which of the DTRTXfull, DTRTXfull_l and InstrCompl flags are used to control the access. The access does not depend on the value of any other flags.

c. If DSCR[6], the Sticky Precise Data Abort bit, is set to 1, the instruction is not issued and InstrCompl remains unchanged. For more information see *Sticky Precise Abort, bit [6]* on page 10-11.

d. The instruction is saved in the ITR and is issued on a read of EXT-DTRTX or a write of EXT-DTRTX. For more information see *EXT-DTR Access Mode, bits [21:20]* on page 10-16.

If the write is made through the Memory-mapped interface and the Software Lock is set to 1, writes to the ITR are ignored, and the ITR, InstrCompl and InstrCompl_l remain unchanged. No instruction is issued. For more information see *Permission summaries for Memory-mapped and external debug interface* on page 5-36.

 Copyright © 2006 ARM Limited. All rights reserved.

10.4 Breakpoint and watchpoint registers

This section contains the following subsections:

- *Breakpoint Value Registers (BVRn)*
- *Breakpoint Control Registers (BCRn)* on page 10-40
- *Watchpoint Value Registers (WVRn)* on page 10-48
- *Watchpoint Control Registers (WCRn)* on page 10-49
- *Vector Catch Register (VCR)* on page 10-54
- *Event Catch Register (ECR)* on page 10-57

10.4.1 Breakpoint Value Registers (BVRn)

The BVRs are registers 64-79, at offsets `0x100-0x13C`.

Each BVR is associated with a BCR register:

- BVR0 with BCR0
- BVR1 with BCR1

This pattern continues up to:

- BVR15 with BCR15.

A pair of breakpoint registers, BVRn and BCRn, is called a Breakpoint Register Pair (BRPn).

The breakpoint value contained in this register corresponds to either an *Instruction Virtual Address* (IVA) or a Context ID. Breakpoints can be set either on an IVA, a Context ID or an IVA/Context ID pair. For the third case, two BRPs have to be linked (see *Breakpoint Control Registers (BCRn)* on page 10-40). A Debug event is generated when both the IVA and the Context ID pair match at the same time.

See *Memory addresses* on page 2-13 for a definition of the IVA used to program a BVR.

Note

Not all BVR registers support Context ID comparison.

Table 10-22 on page 10-40 shows the layout of the Breakpoint Value Registers.

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-22 Breakpoint Value Register bit definition

Bits	Access	Debug reset value	Description	
			IVA comparison	Context ID comparison
[31:2]	RW	UNPREDICTABLE	Breakpoint address [31:2]	Context ID [31:2]
[1:0]	RW[a]	UNPREDICTABLE	Must be written as b00[b]	Context ID [1:0]

a. If this BRP cannot be used for Context ID comparison, BVR[1:0] is RAZ/SBZP.
b. If the BRP can be used for Context ID comparison and is configured as for IVA comparison, then if BVR[1:0] is not programmed as b00, breakpoint hit generation is UNPREDICTABLE.

10.4.2 Breakpoint Control Registers (BCRn)

The Breakpoint Control Registers are registers 80-95, at offsets `0x140-0x71C`.

Table 10-23 shows the layout of the Breakpoint Control Registers.

Table 10-23 Breakpoint Control Register bit definition

Bits	Access	Versions	Debug reset value	Description
[31:29]	RAZ/SBZP	All	-	Reserved
[28:24]	RW	v7	UNPREDICTABLE	*Breakpoint address mask, bits [28:24]* on page 10-47
[23]	RAZ/SBZP	All	-	Reserved
[22:20]	RW	All	UNPREDICTABLE	*Meaning of BVR, bits [22:20]* on page 10-43
[19:16]	RW	All	UNPREDICTABLE	*Linked BRP number, bits [19:16]* on page 10-43
[15:14]	RW	v6.1, v7	UNPREDICTABLE	*Secure world control, bits [15:14]* on page 10-43
[13:9]	RAZ/SBZP	All	-	Reserved
[8:5]	RW	All	UNPREDICTABLE	*Byte address select, bits [8:5]* on page 10-41
[4:3]	RAZ/SBZP	All	-	Reserved
[2:1]	RW	All	UNPREDICTABLE	*Privileged mode control, bits [2:1]* on page 10-41
[0]	RW	All	UNPREDICTABLE[a]	*Breakpoint enable, bit [0]* on page 10-41

a. In ARMv6, the enable bit resets as 0. On an ARMv7 processor, before programming DSCR[15:14] to enable debug, a debugger must ensure that BCR[0] has a defined state.

 Copyright © 2006 ARM Limited. All rights reserved.

Breakpoint enable, bit [0]

The meaning of the Breakpoint enable bit is as follows:

0 Breakpoint disabled

1 Breakpoint enabled.

Privileged mode control, bits [2:1]

The breakpoint can be conditional on the mode of the processor. The meanings of the privileged mode control values are as follows:

b00 USR/SYS/SVC. Match any of User, System and Supervisor Modes.
This value is supported in ARMv7 only.

b01 Privileged. Match in any privileged mode.

b10 USR. Match in User Mode only.

b11 Any. Match in any mode.

For more information see *Generation of Debug events* on page 2-20.

Byte address select, bits [8:5]

The BVR is programmed with a word address. You can use this field to program the breakpoint so that it hits only if certain byte addresses are accessed. The exact interpretation depends on the setting of the Meaning of BVR bits, see *Meaning of BVR, bits [22:20]* on page 10-43.

Table 10-24 Breakpoint hit and miss generation and Byte address select values, when BRP programmed for IVA match or IVA mismatch

CPSR					This BRP programmed for:	
J	**T**	**State**	**Instruction PC value** [a]	**BCR[8:5]**	**IVA match**	**IVA mismatch**
X	X	-	Any address	b0000	Miss	Hit
0	0	ARM	BVR AND `0xFFFFFFFC`	b1111	Hit	Miss
				b0000	Miss	Hit
				Any other value	UNPREDICTABLE	
			Any other address	bxxxx	Miss	Hit

 Copyright © 2006 ARM Limited. All rights reserved.

Table 10-24 Breakpoint hit and miss generation and Byte address select values, when BRP programmed for IVA match or IVA mismatch (continued)

CPSR					This BRP programmed for:	
J	T	State	Instruction PC value [a]	BCR[8:5]	IVA match	IVA mismatch
X	1	Thumb or ThumbEE	BVR AND `0xFFFFFFFC`	bxx11	Hit	Miss
				bxx10	UNPREDICTABLE	
				bxx01	UNPREDICTABLE	
				bxx00	Miss	Hit
			(BVR AND `0xFFFFFFFC`) + 2	b11xx	Hit	Miss
				b10xx	UNPREDICTABLE	
				b01xx	UNPREDICTABLE	
				b00xx	Miss	Hit
			Any other address	bxxxx	Miss	Hit
1	0	Jazelle	BVR AND `0xFFFFFFFC`	bxxx1	Hit	Miss
				bxxx0	Miss	Hit
			(BVR AND `0xFFFFFFFC`) + 1	bxx1x	Hit	Miss
				bxx0x	Miss	Hit
			(BVR AND `0xFFFFFFFC`) + 2	bx1xx	Hit	Miss
				bx0xx	Miss	Hit
			(BVR AND `0xFFFFFFFC`) + 3	b1xxx	Hit	Miss
				b0xxx	Miss	Hit
			Any other address	bxxxx	Miss	Hit

a. The instruction PC value is the address of the first unit of the instruction as described in *Variable length instruction sets* on page 2-11. If this address is the address of a unit of the instruction other than the first unit, the behavior is as described in *Variable length instruction sets* on page 2-11.

If the BRP is programmed for Context ID comparison (linked or unlinked), or an address mask field is used, the byte address select comparison field must be programmed as follows:

b1111 the Byte address select field must be programmed to b1111.

bxxxx UNPREDICTABLE. Breakpoints and watchpoints might not be generated as expected.

In a processor with a trivial implementation of the Jazelle extension that does not provide hardware acceleration of opcodes, writing a value to BCR[8:5] such that BCR[8] != BCR[7], or BCR[6] != BCR[5], has UNPREDICTABLE results.

 Copyright © 2006 ARM Limited. All rights reserved.

These are little-endian byte addresses. This ensures that a breakpoint is triggered regardless of the endianness of the instruction fetch. For example, if a breakpoint is set on a Thumb instruction by setting BCR[8:5] = b0011, it is triggered if fetched little-endian with IVA[1:0] = b00, or if fetched big-endian with IVA[1:0] = b10.

See also *Variable length instruction sets* on page 2-11.

Secure world control, bits [15:14]

v6 Bits [15:14] of BCRn are Reserved and RAZ/SBZP.

v6.1, v7 If the processor does not implement Security Extensions, bits [15:14] of BCRn are Reserved and RAZ/SBZP.

If this processor implements Security Extensions, these bits allow the breakpoint to be conditional on the world of the processor:

b00 breakpoint matches in both Nonsecure and Secure worlds

b01 breakpoint matches only in Nonsecure world

b10 breakpoint matches only in Secure world

b11 Reserved.

See also *Generation of Debug events* on page 2-20.

Linked BRP number, bits [19:16]

If this BRP is configured for linked or unlinked Context ID match, or for unlinked Instruction Virtual Address match or mismatch, BCR[19:16] must be programmed as b0000.

Otherwise, this BRP is configured for linked Instruction Virtual Address match or mismatch, and the binary number encoded in BCR[19:16] indicates another BRP to link this one with.

If a BRP is linked with itself, it is UNPREDICTABLE whether a Breakpoint Debug event is generated or not.

If the BRP that this BRP is linked to is not configured for linked Context ID match, it is UNPREDICTABLE when Breakpoint Debug events might be generated.

See also *Generation of Debug events* on page 2-20.

Meaning of BVR, bits [22:20]

v6 bit [22] is RAZ on v6 Debug implementations.

v6.1, v7 bit [22] is implemented for all BRPs. See also *Variable length instruction sets* on page 2-11 regarding limitations on the use of the mismatch capability on v6.1 Debug.

Copyright © 2006 ARM Limited. All rights reserved.

The meanings of the Meaning of BVR values are as follows:

b000 Unlinked Instruction Virtual Address match

Compare this BVR[31:2] and BCR[8:5] against the IVA bus, and the state of the processor against this BCR[15:14,2:1]. Generate a Breakpoint Debug event on a joint IVA match and state match.

BCR[19:16] must be b0000.

b001 Linked Instruction Virtual Address match

Compare this BVR[31:2] and BCR[8:5] against the IVA bus, and the state of the processor against this BCR[15:14,2:1]. This BRP is linked with the one indicated by BCR[19:16]. Generate a Breakpoint Debug event on a joint IVA match, Context ID match and state match.

b010 Unlinked Context ID match

Compare this BVR[31:0] against the CP15 Context ID (register 13), and the state of the processor against this BCR[15:14,2:1]. This BRP is not linked with any other one. Generate a Breakpoint Debug event on a joint Context ID match and state match.

BCR[8:5] for this BRP must be programmed to b1111, otherwise the generation of breakpoint events is UNPREDICTABLE.

BCR[19:16] must be b0000.

Note

In ARMv6, breakpoint hit generation is disabled in this case if Monitor Debug-mode is selected and the processor is in a privileged mode.

b011 Linked Context ID match

Compare this BVR[31:0] against the CP15 Context ID (register 13). Another BRP (of the BCR[21:20] = b01 type) or WRP (with WCR[20] = 1) is linked with this BRP. Generate a Breakpoint/Watchpoint Debug event on a joint IVA/DVA match and Context ID match.

For this BRP, BCR[8:5] must be programmed to b1111, BCR[15:14] must be programmed to b00, and BCR[2:1] must be programmed to b11, otherwise the generation of breakpoint events is UNPREDICTABLE.

BCR[19:16] must be b0000.

If the state of the processor is to be tested for this breakpoint, the required values for BCR[15:14,2:1] must be programmed in the linked BCR/WCR registers.

Copyright © 2006 ARM Limited. All rights reserved.

b100 Unlinked Instruction Virtual Address mismatch

Compare this BVR[31:2] and BCR[8:5] against the IVA bus, and the state of the processor against this BCR[15:14,2:1]. Generate a Breakpoint Debug event on a joint IVA mismatch (not equal) and state match.

BCR[19:16] must be b0000.

Note

This feature is not supported in v6 Debug.

In v6.1 Debug, breakpoint hit generation is disabled in this case if Monitor Debug-mode is selected and the processor is in a privileged mode.

b101 Linked Instruction Virtual Address mismatch

Compare this BVR[31:2] and BCR[8:5] against the IVA bus, and the state of the processor against this BCR[15:14,2:1]. This BRP is linked with the one indicated by BCR[19:16]. Generate a Breakpoint Debug event on a joint IVA mismatch (not equal), state match and Context ID match.

Note

This feature is not supported in v6 Debug.

In v6.1 Debug, breakpoint hit generation is disabled in this case if Monitor Debug-mode is selected and the processor is in a privileged mode.

b11x Reserved

Behavior is UNPREDICTABLE.

If this BRP does not have Context ID comparison capability, bit [21] reads as zero.

Table 10-25 on page 10-46 shows which values are compared and which are not for each type of BRP. Table entries in **`bold typewriter`** indicate an element of the comparison that is made. Reading across the *Comparison* columns for a row of the table gives the comparison to be made. For example, for the Linked IVA mismatch (b001), the comparison is:

Not (**`Equals`**[IVA] && **`Set`**[Byte lanes]) && **`Match`**[State] && **`Link`**[Linked Breakpoint]

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-25 Meaning of BVR bits summary

Value	Description	Comparison					Notes
		IVA[a, b]	Byte lanes[c]	Context ID[d]	State[e]	Linked	
b000	IVA[a] match	`Equals`	&& `Set`		&& `Match`		-
b001	Linked IVA match	`Equals`	&& `Set`		&& `Match`	&& `Link`	f
b010	Context ID			`Equals`	&& `Match`		g, h
b011	Linked Context ID			`Equals`		&& `Link`	g, i
b100	IVA[a] mismatch	Not (`Equals`	&& `Set`)		&& `Match`		h
b101	Linked IVA[a] mismatch	Not (`Equals`	&& `Set`)		&& `Match`	&& `Link`	f, h
b11x	Reserved	-	-	-	-	-	-

a. IVA = Instruction Virtual Address.
b. Matching IVA[31:2] against BVR[31:2]. If the breakpoint address mask bits [28:24] are set to a value other than b00000, a masked comparison is used. See *Breakpoint address mask, bits [28:24]* on page 10-47.
c. IVA byte lanes. BCR[8:5] indicate the byte lanes to be compared, see *Byte address select, bits [8:5]* on page 10-41.
d. Matching CID[31:0] against BVR[31:0].
e. Processor state comparison made, according to value of BCR[15:14, 2:1], see *Secure world control, bits [15:14]* on page 10-43 and *Privileged mode control, bits [2:1]* on page 10-41.
f. The Context ID is compared against the value of the linked breakpoint and a breakpoint event is only generated when both conditions match. If the linked breakpoint is not capable of Context ID comparison, or is not configured for linked Context ID match, the generation of breakpoint events is UNPREDICTABLE.
g. BCR[8:5] for this BRP must be programmed to b1111; otherwise the generation of Breakpoint Debug events is UNPREDICTABLE.
h. In ARMv6, these events are ignored in privileged modes when Monitor Debug-mode is selected. In ARMv7, these events are generated. Care must therefore be taken when programming BCR[2:1] (supervisor access control) to prevent the processor entering an unrecoverable state. See *unpredictable behavior on Software Debug events* on page 2-14.
i. See *Linked Context ID matching* on page 10-47.

Copyright © 2006 ARM Limited. All rights reserved.

Linked Context ID matching

For linked Context ID matching, the IVA of the instruction or DVA of the data access is compared with the linked BVR or WVR. A linked breakpoint is configured for either match or mismatch. Only if both conditions are true will the Breakpoint or Watchpoint Debug event be generated.

If no breakpoints or watchpoints are linked to this breakpoint, no Debug events will be generated. If a breakpoint that is linked to this breakpoint is not configured for linked IVA match or IVA mismatch the generation of Breakpoint Debug events is UNPREDICTABLE.

For the BRP that is configured for linked Context ID matching, BCR[15:14] must be programmed to b00, and BCR[2:1] must be programmed to b11. If these bits are not programmed correctly, the generation of linked breakpoint or watchpoint events is UNPREDICTABLE. If the state of the processor is to be tested for this breakpoint, the required values for BCR/WCR[15:14,2:1] must be programmed in the linked BCR or WCR registers.

Breakpoint address mask, bits [28:24]

v6, v6.1 Bits [28:24] of BCRn are Reserved and RAZ/SBZP.

v7 Breakpoint address masks are optional. If not implemented, bits [28:24] are RAZ/SBZP.

This field can be used to break on a range of addresses by masking lower order address bits out of the breakpoint comparison. The meaning of Breakpoint address mask values is as follows:

b00000 no mask

b00001 Reserved

b00010 Reserved

b00011 `0x00000007` mask for instruction address

b00100 `0x0000000F` mask for instruction address

b00101 `0x0000001F` mask for instruction address

. .

. .

. .

b11111 `0x7FFFFFFF` mask for instruction address.

The IVA mismatch function can be combined with a masked address. If BCR[28:24] != b00000 and BCR[22] = 1, the address comparison portion of breakpoint generation succeeds for all addresses outside of the masked address region.

If BCR[28:24] != b00000, BCR[8:5] must be set to b1111, otherwise the behavior is UNPREDICTABLE.

If BCR[28:24] != b00000, the corresponding BVR bits that are not being included in the comparison must be zero, otherwise the behavior is UNPREDICTABLE.

If this BRP does not support breakpoint address masking, these bits read as zero.

There is no encoding for a full 32-bit mask. The same effect of a break anywhere breakpoint can be achieved by setting BCR[22] to 1 (selecting an IVA mismatch) and BCR[8:5] to b0000.

If this BRP is programmed for Context ID comparison, this field must be programmed to b00000.

Copyright © 2006 ARM Limited. All rights reserved.

10.4.3 Watchpoint Value Registers (WVRn)

The WVRs are registers 96-111, at offsets `0x180-0x1BC`.

Each WVR is associated with a WCR register:

- WVR0 with WCR0
- WVR1 with WCR1
- ...
- WVR15 with WCR15.

A pair of watchpoint registers, WVRn and WCRn, is called a Watchpoint Register Pair (WRPn).

The watchpoint value contained in this register always corresponds to a *Data Virtual Address* (DVA). See *Memory addresses* on page 2-13 for a definition of the DVA.

Watchpoints can be set either on a DVA or on a DVA/Context ID pair. For the second case a WRP and a BRP with Context ID comparison capability have to be linked, see *Watchpoint Control Registers (WCRn)* on page 10-49. A Debug event is generated when both the DVA and the Context ID pair match at the same time.

Table 10-26 shows the layout of the Watchpoint Value Registers.

Table 10-26 Watchpoint Value Register bit definition

Bits	Access	Debug reset value	Description
[31:2]	RW	UNPREDICTABLE	Watchpoint value bits [31:2]
[1:0]	RAZ/SBZP	-	Reserved

 Copyright © 2006 ARM Limited. All rights reserved.

10.4.4 Watchpoint Control Registers (WCRn)

The WCRs are registers 112-127, at offsets `0x1C0-0x1FC`.

Each of these registers contains all the necessary control bits for setting appropriately either a simple watchpoint or a linked watchpoint.

Table 10-27 shows the layout of the Watchpoint Control Registers.

Table 10-27 Watchpoint Control Register bit definition

Bits	Access	Versions	Debug reset value	Description
[31:29]	RAZ/SBZP	All	0	Reserved
[28:24]	RW	v7	UNPREDICTABLE	*Watchpoint address mask, bits [28:24]* on page 10-53
[23:21]	RAZ/SBZP	All	0	Reserved
[20]	RW	All	UNPREDICTABLE	*Enable linking, bit [20]* on page 10-53
[19:16]	RW	All	UNPREDICTABLE	*Linked BRP number, bits [19:16]* on page 10-52
[15:14]	RW	v6.1, v7	UNPREDICTABLE	*Secure world control, bits [15:14]* on page 10-52
[13]	RAZ/SBZP	All	0	Reserved
[12:5]	RW	All	UNPREDICTABLE	*Byte address select, bits [12:5]* on page 10-50
[4:3]	RW	All	UNPREDICTABLE	*Load/Store access control, bits [4:3]* on page 10-50
[2:1]	RW	All	UNPREDICTABLE	*Privileged access control, bits [2:1]* on page 10-50
[0]	RW	All	UNPREDICTABLE[a]	*Watchpoint enable, bit [0]*

a. In ARMv6, the enable bit resets as 0. On an ARMv7 processor, before programming DSCR[15:14] to enable debug, a debugger must ensure that BCR[0] has a defined state.

Watchpoint enable, bit [0]

The meaning of the Watchpoint enable values is as follows:

0 Watchpoint disabled

1 Watchpoint enabled.

Copyright © 2006 ARM Limited. All rights reserved.

Privileged access control, bits [2:1]

The watchpoint can be conditional on the privilege of the access. The meaning of the privileged access control values is as follows:

b00 Reserved.

b01 Privileged. Match only privileged accesses.

b10 USR. Match only non-privileged accesses.

b11 Any. Match all accesses.

Note

For all cases the match refers to the privilege of the access, not the mode of the processor. For example, if the watchpoint is configured to match privileged accesses only (b01), and the processor executes an `LDRT` or `STRT` instruction in a privileged mode, the watchpoint does not match.

Load/Store access control, bits [4:3]

The watchpoint can be conditional on the type of access being done. The meaning of the Load/Store access control values is as follows:

b00 Reserved.

b00 Load, load exclusive, or swap.

b00 Store, store exclusive or swap. A store exclusive matches whether or not it succeeds.

b00 Any type of access.

Byte address select, bits [12:5]

v6, v6.1 Bits [12:9] of WCRn are Reserved and RAZ/SBZP.

v7 It is IMPLEMENTATION DEFINED whether bits [12:9] of WCRn can be programmed.

WVRs are programmed with word-aligned addresses. You can use this field to program the watchpoint so that it hits only if certain byte addresses are accessed.

Table 10-28 on page 10-51 and Table 10-29 on page 10-51 show the meaning of the byte address select values.

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-28 Meaning of Byte address select values (word-aligned address)

Value	Description
b00000000	watchpoint never hits
bxxxxxxx1	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFFC) + 0 is accessed
bxxxxxx1x	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFFC) + 1 is accessed
bxxxxx1xx	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFFC) + 2 is accessed
bxxxx1xxx	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFFC) + 3 is accessed

In ARMv7, it is IMPLEMENTATION DEFINED whether bits [12:9] of WCRn can also be programmed. If they can be programmed, then WVRn can be programmed with a 64-bit-aligned address (WVRn[2] = 0) and WCRn programmed to match any of the 8-bytes within that 64-bit value.

If WVRn[2] = 1 (indicating a word aligned, but not a 64-bit-byte aligned address), then bits [12:9] of WCRn must be programmed with zero. If WVRn[2] = 1 and WCRn[9:5] != b0000, Watchpoint Debug event generation is UNPREDICTABLE.

If they cannot be programmed, only four byte address select bits are implemented, and WCRn[12:9] are ignored and read-as-zero.

This allows the same programming model on both implementation options.

Table 10-29 Meaning of Byte address select values (doubleword-aligned address)

Value	Description
bxxx1xxxx	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFF8) + 4 is accessed
bxx1xxxxx	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFF8) + 5 is accessed
bx1xxxxxx	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFF8) + 6 is accessed
b1xxxxxxx	watchpoint hits if byte at address (WVR[31:0] & 0xFFFFFFF8) + 7 is accessed

Copyright © 2006 ARM Limited. All rights reserved.

Note

These are little-endian byte addresses. This ensures that, on ARMv6 processors that implement word-invariant memory models, a watchpoint is triggered regardless of the way it is accessed. For example, if a watchpoint is set on a certain byte in memory by setting WCR[8:5] = b0001, and Rn contains a 4-byte aligned address, then:

- `LDRB Rd,[Rn,0]` triggers the watchpoint in the little-endian and BE-8 memory models
- `LDRB Rd,[Rn,3]` triggers the watchpoint in the BE-32 memory model.

BE-32 is not supported in the ARMv7 Architecture.

Secure world control, bits [15:14]

v6 Bits [15:14] of WCRn are Reserved and RAZ/SBZP.

v6.1, v7 If the processor does not implement Security Extensions, bits [15:14] of WCRn are Reserved and RAZ/SBZP.

If this processor implements Security Extensions then these bits allow the watchpoint to be conditional on the security of the world making the access:

b00 watchpoint matches accesses made in both Nonsecure and Secure world

b01 watchpoint only matches accesses made in Nonsecure world

b10 watchpoint only matches accesses made in Secure world

b11 Reserved.

If this processor does not implement Security Extensions these bits are Reserved and read-as-zero.

Note

For all cases the match refers to the security world of the processor, not the security of the access. For example, on a processor that implements the *Virtual Memory System Architecture* (VMSA), the watchpoint does not match when all of the following apply:

- the watchpoint is configured to match in Nonsecure world only (b01)
- the processor is executing code in the Secure world (SCR[0] = 0 or in Monitor Mode)
- the address accessed is in a page marked as nonsecure in the page tables.

Linked BRP number, bits [19:16]

The binary number encoded here indicates a BRP to link this one with.

If the BRP this WRP is linked to is not configured for linked Context ID match, it is UNPREDICTABLE when Watchpoint Debug events might be generated.

If linking is not enabled, bits [19:16] must be zero.

 Copyright © 2006 ARM Limited. All rights reserved.

Enable linking, bit [20]

When this bit is set to 1, this watchpoint is linked with the BRP selected by the linked BRP number field:

0 linking disabled

1 linking enabled.

Watchpoint address mask, bits [28:24]

v6, v6.1 Bits [28:24] of WCRn are Reserved and RAZ/SBZP.

v7 Watchpoint address masks are optional. If not implemented, bits [28:24] are RAZ/SBZP.

This field can be used to watch a range of addresses by masking lower order address bits out of the watchpoint comparison:

b00000 no mask

b00001 Reserved

b00010 Reserved

b00011 `0x00000007` mask for data address

b00100 `0x0000000F` mask for data address

b00101 `0x0000001F` mask for data address

... ...

b11111 `0x7FFFFFFF` mask for data address.

If WCR[28:24] != b00000 then WCR[12:5] must be b11111111 (or WCR[8:5] must be b1111, if bits [12:9] are not implemented), otherwise the behavior is UNPREDICTABLE.

If WCR[28:24] != b00000 then the corresponding WVR bits that are not being included in the comparison must be zero, otherwise the behavior is UNPREDICTABLE.

If this WRP does not support watchpoint address masking, these bits read as zero.

To watch for a write to any byte in an 8-byte aligned object of size 8 bytes, debuggers are recommended to set WCR[28:24] = `0x7`, and WCR[12:5] = b11111111, as this is compatible with both implementations with 8 byte address select bits and implementations with 4 byte address select bits, because in the latter case writes to WCR[12:9] are ignored.

Copyright © 2006 ARM Limited. All rights reserved.

10.4.5 Vector Catch Register (VCR)

The VCR is register 7, at offset `0x01C`.

v6 Bits [31,30,28:25,15,14,12:10] are not implemented. These bits are Reserved and RAZ/SBZP.

v6.1 Bits [31,30,28:25,15,14,12:10] are not implemented on processors that do not implement Security Extensions. If the processor does implement Security Extensions, these bits are optional. If these bits are not implemented, they are Reserved and RAZ/SBZP.

For forwards compatibility with ARMv7, ARM Limited recommends that these bits are implemented on processors that implement Security Extensions.

v7 Bits [31,30,28:25,15,14,12:10] are not implemented on processors that do not implement Security Extensions. If the processor does implement Security Extensions, these bits are required. If these bits are not implemented, they are Reserved and RAZ/SBZP.

If a bit in the VCR is 1, when an instruction is prefetched from the corresponding vector address and committed for execution, either a Debug exception or a Debug State entry can be generated.

For more information, see *Vector Catch Debug events* on page 2-13.

The vector catch behaves exactly like a BRP set with BVR[31:2] set to the top 30-bits of the exception vector address, and BCR[8:5] set to b1111 and programmed for IVA match.

Note

Under this model, any kind of prefetch from an exception vector can trigger a vector catch, not just those due to exception entries.

Instruction fetches from non word-aligned addresses within the 4-bytes of the exception vector address also trigger vector catches, for example, Thumb instruction fetches from the second half-word of the address.

However, unlike breakpoints, if the vector catch address matches a unit of an instruction in a variable length instruction set that is not the first unit of the instruction, vector catch generation is UNPREDICTABLE. See *Variable length instruction sets* on page 2-11 for more information on breakpoint generation and variable length instruction sets.

Bits [7:6,4:0] are required for all implementations.

Table 10-30 on page 10-55 shows the bit definition for processors that implement bits [31:30,28:25,15:14,12:10]. All such processors implement Security Extensions. Therefore each vector address is relative to a vector base address register. The table uses the following abbreviations:

VBAR Vector Base Address Register (Secure)

$VBAR_{NS}$ Vector Base Address Register (Nonsecure)

MVBAR Monitor Vector Base Address Register.

If bits [31:8] are implemented:

- catches due to bits [15:0] are only triggered in Secure world
- catches due to bits [31:25] are only triggered in Nonsecure world.

 Copyright © 2006 ARM Limited. All rights reserved.

If bits [31:8] are not implemented and the processor implements Security Extensions (v6.1 only), then catches due to bits [7:0] are triggered in either world, with the current world vector base address used as the base-address for the vector catch match.

Table 10-30 Vector Catch Register bit definition, for processors that implement Security Extensions

Bits	Access	Vector Catch enable		World	Normal address	High vector address
[31]	RW	FIQ	VE = 0	Nonsecure	$VBAR_{NS}$+0x0000001C	0xFFFF001C
			VE = 1	Nonsecure	Most recent nonsecure FIQ address	
[30]	RW	IRQ	VE = 0	Nonsecure	$VBAR_{NS}$+0x00000018	0xFFFF0018
			VE = 1	Nonsecure	Most recent nonsecure IRQ address	
[29]	RAZ/SBZP	Reserved		-	-	-
[28]	RW	Data Abort		Nonsecure	$VBAR_{NS}$+0x00000010	0xFFFF0010
[27]	RW	Prefetch abort		Nonsecure	$VBAR_{NS}$+0x0000000C	0xFFFF000C
[26]	RW	SVC		Nonsecure	$VBAR_{NS}$+0x00000008	0xFFFF0008
[25]	RW	Undefined		Nonsecure	$VBAR_{NS}$+0x00000004	0xFFFF0004
[24:16]	RAZ/SBZP	Reserved		-	-	-
[15]	RW	FIQ		Secure	MVBAR+0x0000001C	MVBAR+0x0000001C
[14]	RW	IRQ		Secure	MVBAR+0x00000018	MVBAR+0x00000018
[13]	RAZ/SBZP	Reserved		-	-	-
[12]	RW	Data Abort		Secure	MVBAR+0x00000010	MVBAR+0x00000010
[11]	RW	Prefetch abort		Secure	MVBAR+0x0000000C	MVBAR+0x0000000C
[10]	RW	SMC		Secure	MVBAR+0x00000008	MVBAR+0x00000008
[9:8]	RAZ/SBZP	Reserved		-	-	-
[7]	RW	FIQ	VE = 0	Secure	VBAR+0x0000001C	0xFFFF001C
			VE = 1	Secure	Most recent Secure FIQ address	
[6]	RW	IRQ	VE = 0	Secure	VBAR+0x00000018	0xFFFF0018
			VE = 1	Secure	Most recent Secure IRQ address	
[5]	RAZ/SBZP	Reserved		-	-	-
[4]	RW	Data Abort		Secure	VBAR+0x00000010	0xFFFF0010
[3]	RW	Prefetch abort		Secure	VBAR+0x0000000C	0xFFFF000C
[2]	RW	SVC		Secure	VBAR+0x00000008	0xFFFF0008
[1]	RW	Undefined		Secure	VBAR+0x00000004	0xFFFF0004
[0]	RW	Reset		-	0x00000000	0xFFFF0000

Copyright © 2006 ARM Limited. All rights reserved.

Note

Debug Reset Values

In ARMv6, all defined bits of the VCR reset as 0. On an ARMv7 processor, the reset values are UNPREDICTABLE, and before programming DSCR[15:14] to enable debug, a debugger must ensure that the VCR has a defined state.

If the processor does not implement Security Extensions, bits [31:8] are not implemented and read as zero. Table 10-31 shows the VCR definition for processors that do not implement Security Extensions. Because the vector base address registers are part of the Security Extensions, the vector addresses are absolute.

Table 10-31 Vector Catch Register bit definition, for processors without Security Extensions

Bits	Access	Vector Catch enable		Normal address	High vectors address
[31:8]	RAZ/SBZP	Reserved		-	-
[7]	RW	FIQ	VE = 0	`0x0000001C`	`0xFFFF001C`
			VE = 1	Most recent FIQ address	
[6]	RW	IRQ	VE = 0	`0x00000018`	`0xFFFF0018`
			VE = 1	Most recent IRQ address	
[5]	RAZ/SBZP	Reserved		-	-
[4]	RW	Data Abort		`0x00000010`	`0xFFFF0010`
[3]	RW	Prefetch abort		`0x0000000C`	`0xFFFF000C`
[2]	RW	SVC		`0x00000008`	`0xFFFF0008`
[1]	RW	Undefined		`0x00000004`	`0xFFFF0004`
[0]	RW	Reset		`0x00000000`	`0xFFFF0000`

Note

Debug Reset Values

In ARMv6, all defined bits of the VCR reset as 0. On an ARMv7 processor, the reset values are UNPREDICTABLE, and before programming DSCR[15:14] to enable debug, a debugger must ensure that the VCR has a defined state.

Bits [28:27,12] (where implemented) and bits [4:3] must be programmed as zero if Monitor Debug-mode is configured and enabled. See *unpredictable behavior on Software Debug events* on page 2-14.

 Copyright © 2006 ARM Limited. All rights reserved.

10.4.6 Event Catch Register (ECR)

The ECR is register 9,at offset `0x024`, and is not defined in ARMv6.

The ECR allows the external debugger or debug monitor to configure the debug logic so that it triggers a Debug state entry on certain conditions.

Table 10-32 shows the layout of the Event Catch Register.

Table 10-32 Event Catch Register bit definition

Bits	Access	Debug reset value	Description
[31:1]	RAZ/SBZP	-	Reserved
[0]	RW	0	*OS Unlock Catch, bit [0]*

OS Unlock Catch, bit [0]

The meanings of OS Unlock Catch values are as follows:

0 Catch disabled

1 Catch enabled.

If this bit is 1, an OS Unlock Catch Debug event is generated on unlocking of the OS Lock, see *OS Lock Access Register (OSLAR)* on page 10-58. The OS Unlock Catch Debug event is a Halting Debug event, see *Halting Debug events* on page 2-18.

OS Unlock Catch Debug events are only generated on clearing of the OS Lock. That is, its transition from locked to unlocked.

Note

In the scenario where a debugger is monitoring an application running on top of an OS with save/restore capability, this event indicates the right time when the debug session can continue.

If an implementation supports debug over power-down, then it must support the OS Unlock Catch Debug event. If the implementation does not support debug over power-down, the OS Unlock Catch Debug event is not supported and this bit is read-as-zero.

 Copyright © 2006 ARM Limited. All rights reserved.

10.5 Operating-system save and restore registers

v6, v6.1 These registers are not defined in ARMv6.

v7 If an implementation supports debug over power-down, then it must support the OS save and restore registers. If the implementation does not support debug over power-down, these registers are implemented as read-as-zero/write-ignored.

In *SinglePower* systems, it is IMPLEMENTATION DEFINED whether the OS Save and Restore Registers are implemented.

This section contains the following subsections:

- *OS Lock Access Register (OSLAR)*
- *OS Lock Status Register (OSLSR)* on page 10-59
- *OS Save and Restore Register (OSSRR)* on page 10-60.

10.5.1 OS Lock Access Register (OSLAR)

The OSLAR is register 192, at offset `0x300`.

The layout of the OSLAR is shown in Table 10-33.

Table 10-33 OS Lock Access Register

Bits	Access	Description
[31:0]	RAZ/W	OS Lock Access

Writing the key value `0xC5ACCE55` to the OS Lock Access Register (OSLAR) locks the debug registers, so that accesses to locked registers return a slave-generated error response. Writing any other value to OSLAR unlocks them if they are currently locked.

Writing the key also has the side effect of resetting the internal save/restore counter.

For details of error responses when accessing the debug registers, see *Access permissions* on page 5-20.

Note

A write to OSLAR is not guaranteed to lock/unlock the OS Lock until the end of the next *Data Synchronization Barrier* (DSB) operation. See *Synchronization of debug register updates* on page 5-18 for details.

If bit [0] of the Event Catch Register is set to 1 at the point when the OS Lock is unlocked, an OS Unlock Catch Debug event is generated. See *OS Unlock Catch, bit [0]* on page 10-57 for details.

 Copyright © 2006 ARM Limited. All rights reserved.

10.5.2 OS Lock Status Register (OSLSR)

The OSLSR is register 193, at offset `0x304`.

Table 10-34 shows the layout of the OSLSR.

Table 10-34 OS Lock Status Register

Bits	Access	Debug reset value	Description
[31:3]	RAZ	-	Reserved
[2]	RO	0	*32-bit access, bit [2]*
[1]	RO	**DBGOSLOCKINIT**	*Locked, bit [1]*
[0]	RO	1	*Lock implemented, bit [0]*

Lock implemented, bit [0]

This bit reads 1 if it is possible to set the OS Lock for this processor. If this bit reads 0, OS Lock and OS Save/Restore registers are not implemented and the entire register reads as zero.

Locked, bit [1]

The meanings of the Locked bit are as follows:

0 Lock not set.

1 Lock set. Writes ignored.

On debug logic reset:

- if **DBGOSLOCKINIT** is LOW this lock is not set
- if **DBGOSLOCKINIT** is HIGH this lock is set.

32-bit access, bit [2]

This bit always reads as 0. It indicates that a 32-bit access is needed to write the key to the OS Lock Access Register.

Copyright © 2006 ARM Limited. All rights reserved.

10.5.3 OS Save and Restore Register (OSSRR)

The OSSRR is register 194, at offset `0x308`.

Table 10-35 shows the layout of the OSSRR.

Table 10-35 OS Save and Restore Register

Bits	Access	Debug reset value	Description
[31:0]	RW	UNPREDICTABLE	OS Save/Restore

This register works in conjunction with an internal sequence counter, so that a series of reads or writes of this register return or restore the complete debug state of the processor that would be lost when the core is powered down.

The OS Lock must be set before accessing the OSSRR, see *OS Lock Access Register (OSLAR)* on page 10-58.

See *Operating System Save and Restore support* on page 5-9 for a description of using the OS Save and Restore Registers.

Copyright © 2006 ARM Limited. All rights reserved.

10.6 Memory system control registers

v6, v6.1 In some ARMv6 implementations a Cache Behavior Override Register (CBOR) and a TLB Debug Control Register (TDCR) are provided in the IMPLEMENTATION DEFINED region of the CP15 register space. In addition, primarily in v6.1 Debug implementations, the Debug state MMU Control Register (DSMCR) and Debug state Cache Control Register (DSCCR) are implemented as IMPLEMENTATION DEFINED extensions to CP14 as described below.

ARMv6 does not architecturally require such registers. However, these features are recommended to assist debuggers in maintaining memory coherency avoiding costly explicit coherency operations.

v7 In ARMv7, the DSMCR and DSCCR registers are mandatory; however, there may be IMPLEMENTATION DEFINED limits on their behavior. The CP15 registers CBOR and TDCR remain IMPLEMENTATION DEFINED.

The Debug State Cache Control Register (DSCCR) and Debug State MMU Control Register (DSMCR) control cache and TLB behavior for memory operations issued by a debugger when the processor is in Debug state.

They enable a debugger to request the minimum amount of intrusion to the processor caches, as permitted by the implementation. It is IMPLEMENTATION DEFINED what levels of cache and TLB are controlled by these requests, and it is IMPLEMENTATION DEFINED to what extent the intrusion will be limited.

The DSCCR also provides a mechanism for a debugger to force writes to memory through to the point of coherency without the overhead of issuing additional operations.

The DSCCR and DSMCR controls must apply for all memory operations issued in Debug state when the DSCR[19] Imprecise Aborts discarded bit is set to 1. It is IMPLEMENTATION DEFINED whether memory operations issued in Debug state whilst this bit is not set to 1 are affected by the DSCCR and DSMCR.

Copyright © 2006 ARM Limited. All rights reserved.

10.6.1 Debug State Cache Control Register (DSCCR)

The DSCCR is register 10, at offset `0x028`.

The DSCCR controls cache behavior in Debug state.

Table 10-36 shows the layout of the DSCCR.

Table 10-36 Debug State Cache Control Register bit definition

Bits	Access	Debug reset value[a]	Mnemonic	Description
[31:3]	RAZ/SBZP	-	-	Reserved.
[2]	RW	UNPREDICTABLE	nWT	*Force Write-through, bit [2]* on page 10-63.
[1]	RW/RAZ	UNPREDICTABLE	nIL/-	*Cache Line-Fill and Eviction, bits [1:0]* / Reserved.
[0]	RW/RAZ	UNPREDICTABLE	nDL/-	*Cache Line-Fill and Eviction, bits [1:0]* / Reserved.

a. In ARMv6, all defined bits of the DSCCR reset as 0. On an ARMv7 processor, the reset values are UNPREDICTABLE, and before issuing operations through the ITR with the processor in Debug state, a debugger must ensure that the DSCCR has a defined state.

Cache Line-Fill and Eviction, bits [1:0]

The meaning of the Cache Line-Fill values is as follows:

0 Request disabling of cache line-fills and evictions for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation of cache line-fills and evictions for memory operations issued by a debugger when the processor is in Debug state.

When cache line-fill and eviction is disabled, all memory accesses that would be checked against a cache are checked against the cache. If a match is found, the cached result is used. If no match is found the next level of memory is used, but the result is not cached, and no cache entries are evicted.

The *next level of memory* can refer to looking in the next level of cache, or to accessing external memory, depending on the numbers of levels of cache implemented.

When the processor is in Debug state, cache maintenance operations are not affected by the nDL and nIL control bits, and have their normal architected behavior.

The behavior of memory hint instructions, `PLD` and `PLI`, is UNPREDICTABLE in Debug state when the corresponding nDL or nIL control bit is set to 1.

Copyright © 2006 ARM Limited. All rights reserved.

Force Write-through, bit [2]

The meaning of the Force Write-through values is as follows:

0 Force write-through behavior for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation for memory operations issued by a debugger when the processor is in Debug state.

In Debug state, if the nWT bit is set to 1, on completion a write to memory the effect of the write must be visible at all levels of memory to the Point of Coherency (PoC). The nWT control *must* be implemented at all levels of memory to the PoC. This allows an External Debugger to write through to the PoC without having to perform costly and intrusive cache clean operations.

Note

nWT does not force the ordering of writes, and does not force writes to complete immediately. A debuggers might have to insert a barrier operations to ensure ordering.

Permitted IMPLEMENTATION DEFINED limits

The DSCCR register is required. However, there can be IMPLEMENTATION DEFINED limits on its behavior. Table 10-37 lists five permitted options for implementations (some of these options are orthogonal). However, nWT must always be implemented to allow a debugger to ensure that writes are made to the point of coherency.

Table 10-37 shows the limits on DSCCR behavior.

Table 10-37 Permitted IMPLEMENTATION DEFINED limits on DSCCR behavior

Limit	Description	Notes
Full DSCCR	Bits [2:0] implemented	-
No I-cache control	bit [1] RAZ	Instruction cache line-fill and eviction disable features not implemented. Instruction fetches are disabled in Debug state. For most implementations no instruction cache accesses take place in Debug state, and nIL is not required.
Unified cache	bit [1] RAZ	-
Cache evictions always enabled	-	nIL / nDL disables cache line-fills in Debug state. However cache evictions might still take place even when these control bits are set to 0.
No line-fill control	Bits [1:0] RAZ	No cache line-fill and eviction disable features are implemented.

Copyright © 2006 ARM Limited. All rights reserved.

Interaction with Cache Behavior Override Register

On ARMv6 processors, a Cache Behavior Override Register (CBOR) might also be implemented in CP15.

Table 10-38 shows the relative precedence of the CBOR and the DSCCR according to the state of the processor where a core implements both the Debug state Cache Control Register (DSCCR) and the CBOR.

Table 10-38 Interaction of CP15 Cache Behavior Override Register (CBOR) and DSCCR

DSCCR setting	CBOR setting	Debug state	Behavior
nWT = 1	WT = 0	X	areas marked WB are write-back
nWT = X	WT = 0	No	areas marked WB are write-back
nWT = X	WT = 1	X	areas marked WB are write-through
nWT = 0	WT = X	Yes	areas marked WB are write-through
nDL = 1	DL = 0	X	Data/Unified cache line-fills are enabled
nDL = X	DL = 0	No	Data/Unified cache line-fills are enabled
nDL = X	DL = 1	X	Data/Unified cache line-fills are disabled
nDL = 0	DL = X	Yes	Data/Unified cache line-fills are disabled
nIL = 1	IL = 0	X	Instruction cache line-fills are enabled
nIL = X	IL = 0	No	Instruction cache line-fills are enabled
nIL = X	IL = 1	X	Instruction cache line-fills are disabled
nIL = 0	IL = X	Yes	Instruction cache line-fills are disabled

A processor that does not implement Security Extensions has only WT, IL and DL settings in the CP15 Cache Behavior Override Register. Processors that implement Security Extensions can have separate settings for, for example, NS_WT and S_WT in the CP15 Cache Behavior Override Register. For brevity the full matrix of possibilities in this case is omitted from Table 10-38. For the behavior on such a processor, duplicate Table 10-38, once for the Nonsecure case (for example NS_WT), once for the Secure case (for example S_WT).

Copyright © 2006 ARM Limited. All rights reserved.

10.6.2 Debug State MMU Control Register (DSMCR)

The DSMCR is register 11, at offset `0x02C`.

The DSMCR controls TLB behavior in Debug state. Table 10-39 shows the layout of the DSMCR.

Table 10-39 Debug State MMU Control Register bit definition

Bits	Access	Debug reset value[a]	Mnemonic	Description
[31:4]	RAZ / SBZP	-	-	Reserved
[3]	RW / RAZ	UNPREDICTABLE	nIUM/-	Instruction *TLB Matching, bit [3:2]* on page 10-66 / Reserved
[2]	RW / RAZ	UNPREDICTABLE	nDUM/-	Data/Unified *TLB Matching, bit [3:2]* on page 10-66 / Reserved
[1]	RW / RAZ	UNPREDICTABLE	nIUL/-	Instruction *TLB Loading, bits [1:0]* / Reserved
[0]	RW / RAZ	UNPREDICTABLE	nDUL/-	Data/Unified *TLB Loading, bits [1:0]* / Reserved

a. In ARMv6, all defined bits of the DSMCR reset as 0. On an ARMv7 processor, the reset values are UNPREDICTABLE, and before issuing operations through the ITR with the processor in Debug state, a debugger must ensure that the DSMCR has a defined state.

TLB Loading, bits [1:0]

The meanings of the TLB loading values are as follows:

0 Request disabling of TLB load and flush for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation of TLB loading and flushing for memory operations issued by a debugger when the processor is in Debug state.

When TLB load and flush is disabled, all memory accesses normally checked against a TLB are checked against the TLB. If a match is found, the cached result is used. If no match is found the next level of translation is performed, but the result is not cached in the TLB, and no TLB entries are evicted.

The *next level of translation* might mean looking in the next level TLB, or doing a page table walk, depending on the numbers of levels of TLB implemented.

In Debug state, TLB maintenance operations are not affected by the nDUL/nIUL control bits, and have their normal architected behavior.

 Copyright © 2006 ARM Limited. All rights reserved.

TLB Matching, bit [3:2]

The meanings of the TLB matching values are as follows:

0 Request disabling of TLB matching for memory operations issued by a debugger when the processor is in Debug state

1 Normal operation of TLB matching for memory operations issued by a debugger when the processor is in Debug state.

When TLB matching is disabled, all memory accesses normally checked against a TLB are not checked against the TLB. For every access the next level of translation is performed. The results are not cached in the TLB, and no TLB entries are evicted. The next level of translation is used for every access.

The *next level of translation* might mean looking in the next level TLB, or doing a page table walk, depending on the numbers of levels of TLB implemented.

Note

If TLB matching is disabled, and TLB management functions have not been correctly performed by the system being debugged (for example, if the TLB has not been flushed following a change to the translation tables), memory accesses made by the debugger might not undergo the same virtual to physical memory mapping results as the application being debugged.

A debugger can create temporary alternative memory mappings by altering the contents of the external page tables and disabling all levels of TLB matching. However, for normal debugging operations, it is recommended that this bit is set to 1.

Permitted IMPLEMENTATION DEFINED limits

The DSMCR register is required. However, there can be IMPLEMENTATION DEFINED limits on its behavior. Table 10-40 lists six permitted options for implementations (some of these options are orthogonal).

Table 10-40 Permitted IMPLEMENTATION DEFINED limits on DSCCR behavior

Limit	Description	Notes
Full DSMCR	Bits [2:0] implemented	-
No I-TLB controls	Bits [3,1] RAZ	Instruction cache line-fill and eviction disable features not implemented. Instruction fetches disabled in Debug state. For most implementations no TLB accesses take place in Debug state, and nIUL and nIUM are not required.
Unified TLB	Bits [3,1] RAZ	-

 Copyright © 2006 ARM Limited. All rights reserved.

Table 10-40 Permitted IMPLEMENTATION DEFINED limits on DSCCR behavior (continued)

Limit	Description	Notes
No matching control	Bits [3:2] RAZ	The TLB matching controls are not used to reduce the impact of debugging, only for advanced debugging features. If not implemented, these bits read as zero, although the processor behaves as if they were set to 1.
TLB evictions always enabled	-	nIUL / nDUL disable TLB loading in Debug state. However TLB evictions can still take place even when these control bits are set to 0.
No loading control	Bits [1:0] RAZ	

Interaction with TLB Debug Control Register

On ARMv6 processors, a TLB Debug Control Register (TDCR) can also be implemented in CP15.

Where a core implements both the Debug state MMU Control Register (DSMCR) and the TDCR, Table 10-41 shows the relative precedence of the TDCR and the DSMCR according to the state of the processor.

Table 10-41 Interaction of CP15 TLB Debug Control Register (TDCR) and DSMCR

DSMCR setting	TDCR setting	Debug state	Behavior
nDUM = 1	DUM = 0	X	Data/Unified TLB matching enabled
nDUM = X	DUM = 0	No	Data/Unified TLB matching enabled
nDUM = X	DUM = 1	X	Data/Unified TLB matching disabled
nDUM = 0	DUM = X	Yes	Data/Unified TLB matching disabled
nDUL = 1	DUL = 0	X	Data/Unified TLB loading enabled
nDUL = X	DUL = 0	No	Data/Unified TLB loading enabled
nDUL = X	DUL = 1	X	Data/Unified TLB loading disabled
nDUL = 0	DUL = X	Yes	Data/Unified TLB loading disabled

The same interaction applies for the IUM and IUL control bits in the TDCR, and the corresponding nIUM and nIUL control bits in the DSMCR. If the TDCR provides controls for more than one level of TLB, these controls interact with the DSMCR controls in the same way.

Copyright © 2006 ARM Limited. All rights reserved.

A processor that does not implement Security Extensions has only IUM, DUM and related settings in the TDCR. Processors that implement Security Extensions can have separate NS_IUM, S_IUM and related settings in the TDCR. For brevity the full matrix of possibilities in this case is omitted from Table 10-41 on page 10-67. For the behavior on such an implementation, duplicate Table 10-41 on page 10-67, once for the Nonsecure case (NS_DUM and NS_DUL), once for the Secure case (S_DUM and S_DUL).

 Copyright © 2006 ARM Limited. All rights reserved.

10.7 Management registers

These registers are not defined in ARMv6.

The layout of the Management Registers (registers 832-1023) is in line with the *CoreSight Architecture Specification*.

10.7.1 Processor Identification Registers

These are registers 832-895, at offsets `0xD00-0xDFC`.

Note

The Extended CP14 Interface `MRC` instructions that map to these registers return UNPREDICTABLE values, and the `MCR` instructions are ignored. These registers can be read through the CP15 interface.

These registers return the values stored in the main ID and feature registers of the processor. Writes to these registers are ignored.

Table 10-42 Processor Identifier Registers

Register number	Access	Mnemonic	Description
832	RO	CPUID	Main ID Register [a]
833	RO (RAZ)[b]	CACHETYPE (-)[b]	Cache Type Register[a] (Reserved)[b]
834	RO (RAZ)[b]	TCMTYPE (-)[b]	TCM Type Register [a] (Reserved)[b]
835	RO (RAZ)[b]	TLBTYPE (-)[b]	TLB Type Register [a] (Reserved)[b]
836	RO (RAZ)[b]	MPUTYPE (-)[b]	MPU Type Register [a] (Reserved)[b]
837	RO (RAZ)[b]	MPAFF (-)[b]	MP Affinity Register[a] (Reserved)[b]
838	RAZ	-	Reserved
839	RO (RAZ)[b]	FEATID (-)[b]	Feature ID Register[a] (Reserved)[b]
840	RO	ID_PFR0	Processor Feature Register 0
841	RO	ID_PFR1	Processor Feature Register 1
842	RO	ID_DFR0	Debug Feature Register 0
843	RO	ID_AFR0	Auxiliary Feature Register 0
844	RO	ID_MMFR0	Processor Feature Register 0
845	RO	ID_MMFR1	Processor Feature Register 1

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-42 Processor Identifier Registers (continued)

Register number	Access	Mnemonic	Description
846	RO	ID_MMFR2	Processor Feature Register 2
847	RO	ID_MMFR3	Processor Feature Register 3
848	RO	ID_ISAR0	ISA Feature Register 0
849	RO	ID_ISAR1	ISA Feature Register 1
850	RO	ID_ISAR2	ISA Feature Register 2
851	RO	ID_ISAR3	ISA Feature Register 3
852	RO	ID_ISAR4	ISA Feature Register 4
853	RO	ID_ISAR5	ISA Feature Register 5
854-895	RAZ	-	Reserved

a. Implemented ID code registers, registers 832-839, return the same value as returned by the instruction `MRC p15,0,Rd,c0,c0,<opcode_2>`, where `<opcode_2>` is the register number minus 832. If the register is not implemented or Reserved, the register reads as zero. For details of these registers, refer to the *ARM Architecture Reference Manual*.

b. These registers are optional. The information in brackets applies when the register is not implemented.

10.7.2 Integration Mode Control Register (ITCTRL)

The ITCTRL register is register 960, at offset `0xF00`.

The ITCTRL register allows the device to switch from its default functional mode into *Integration mode*, where the inputs and outputs of the device can be directly controlled for the purpose of integration testing or topology detection. Once the processor is in this mode, the IMPLEMENTATION DEFINED Integration Registers can be used to drive output values and to read inputs.

Table 10-43 shows the layout of the Integration Mode Control Register.

Table 10-43 Integration Mode Control Register bit definition

Bits	Access	Debug reset value	Description
[31:1]	RAZ/SBZP	-	Reserved
[0]	RW	0	*Integration Mode Enable* on page 10-71

Copyright © 2006 ARM Limited. All rights reserved.

Integration Mode Enable

The meanings of the Integration Mode Enable bit are as follows:

0 Normal operation

1 Integration Mode enabled.

When this bit is set to 1, the device reverts to an Integration Mode to enable integration testing or topology detection. The Integration Mode behavior is IMPLEMENTATION DEFINED.

10.7.3 Claim Tag Set Register (CLAIMSET)

The Claim Tag Set Register is register 1000, at offset `0xFA0`.

CLAIMSET bits do not have any specific functionality. The expected usage model is for an external debugger and a debug monitor to set specific bits to 1 to claim the corresponding debug resources. ARMv7 processors implement eight claim tag bits.

Table 10-44 shows the layout of the Claim Tag Set Register.

Table 10-44 Claim Tag Set Register bit definition

Bits	Access	Debug reset value	Description
[31:8]	RAZ/SBZP	-	Reserved
[7:0]	RAO/W	`0xFF`	Claim tags

Writing 1 to a specific claim tag set bit sets that claim tag. Writing 0 to a specific claim tag bit has no effect. This register always reads `0xFF`, indicating eight claim tags are implemented.

10.7.4 Claim Tag Clear Register (CLAIMCLR)

The Claim Tag Clear Register is register 1001, at offset `0xFB0`.

Table 10-44 shows the layout of the Claim Tag Clear Register.

Table 10-45 Claim Tag Clear Register bit definition

Bits	Access	Debug reset value	Description
[31:8]	RAZ/SBZP	-	Reserved
[7:0]	RW	`0x00`	Claim tags

Writing 1 to a specific claim tag set bit clears that claim tag. Writing 0 to a specific claim tag bit has no effect. Reading this register returns the current claim tag values.

 Copyright © 2006 ARM Limited. All rights reserved.

10.7.5 Lock Access Register (LAR)

The LAR is register 1004, at offset `0xFB0`.

Note

This register reads as zero and ignores writes when accessed through the external debug interface or Extended CP14 Interface.

Where the external debug interface and Memory-mapped interface use the same port, external debug interface accesses are distinguished by, for example, **PADDRDBG**[31] being set to 1, see *PADDRDBG* on page 6-12.

Writes to the debug registers through the Memory-mapped interface are controlled via the Lock Access Register. The purpose of this lock mechanism is to reduce the risk of accidental damage to the contents of the debug registers. It does not, and cannot, prevent all accidental or malicious damage.

The Software Lock (as opposed to the OS Lock described in *OS Lock Access Register (OSLAR)* on page 10-58) is cleared by writing the `0xC5ACCE55` key to the Lock Access Register (LAR). It is set by writing any value other than `0xC5ACCE55` to the LAR.

Note

The state of this lock is on debug power. It is unaffected by the core powering down.

This lock is set on reset of the debug power domain (that is, on **PRESETDBGn** or **nSYSPORESET**).

Table 10-46 shows the layout of the Lock Access Register.

Table 10-46 Lock Access Register bit definition

Bits	Access	Description
[31:0]	RAZ/W	Lock access control. Write a `0xC5ACCE55` key to unlock the debug registers. Write any other value to lock them.

Accesses through the Memory-mapped interface to locked debug registers are ignored, see *Permissions in relation to locks* on page 5-21 for more information.

Copyright © 2006 ARM Limited. All rights reserved.

10.7.6 Lock Status Register (LSR)

The Lock Status Register is register 1005, at offset `0xFB4`.

Note

This register reads as zero when read through the external debug interface or Extended CP14 Interface.

Where the external debug interface and Memory-mapped interface use the same port, external debug interface accesses are distinguished by, for example, **PADDRDBG**[31] being set to 1, see *PADDRDBG* on page 6-12.

The Lock Status Register returns the current lock status. Table 10-46 on page 10-72 shows the layout of the Lock Status Register.

Table 10-47 Lock Status Register bit definition

Bits	Access	Debug reset value	Description
[31:3]	RAZ/SBZP	-	Reserved
[2]	RO	0	*32-bit Access, bit [2]* on page 10-74
[1]	RO	See *Locked, bit [1]*	*Locked, bit [1]*
[0]	RO	See *Lock implemented, bit [0]*	*Lock implemented, bit [0]*

Lock implemented, bit [0]

The meanings of the Lock Implemented bit are as follows:

0 This access is from an interface that ignores the lock registers, that is, the external debug interface or the Extended CP14 Interface

1 This access is from an interface that requires the registers to be unlocked, that is, the Memory-mapped interface.

Locked, bit [1]

The meanings of the Locked bit are as follows:

0 writes are permitted

1 writes are ignored.

For interfaces that require the registers to be unlocked, the registers are locked from reset, and this bit has the reset value 1. For interfaces that ignore the lock registers, this bit always reads as zero.

Copyright © 2006 ARM Limited. All rights reserved.

32-bit Access, bit [2]

This bit always reads 0. It indicates that a 32-bit access is needed to write the key to the Lock Access Register.

10.7.7 Authentication Status Register (AUTHSTATUS)

AUTHSTATUS is register 1006, at offset `0xFB8`.

AUTHSTATUS reads the current values of the configuration inputs that determine the debug permission level. The value read depends on whether the processor implements Security Extensions.

Table 10-48 shows the layout of the Authentication Status Register for a processor that implements Security Extensions.

Table 10-49 on page 10-75 shows the layout of the Authentication Status Register for a processor that does not implement Security Extensions. If a processor does not implement Security Extensions, it does not implement any Nonsecure debug features.

Table 10-48 Authentication Status Register bit definition, when Security Extensions are implemented

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7]	RO	1	Secure non-invasive debug features implemented
[6]	RO	Logical result of (**DBGEN** OR **NIDEN**) AND (**SPIDEN** OR **SPNIDEN**)	Secure non-invasive debug enabled
[5]	RO	1	Secure invasive debug features implemented
[4]	RO	Logical result of (**DBGEN** AND **SPIDEN**)	Secure invasive debug enabled
[3]	RO	1	Nonsecure non-invasive debug features implemented
[2]	RO	Logical result of (**DBGEN** OR **NIDEN**)	Nonsecure non-invasive debug enabled
[1]	RO	1	Nonsecure invasive debug features implemented
[0]	RO	State of **DBGEN** signal	Nonsecure invasive debug enabled

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-49 Authentication Status Register bit definition, when Security Extensions are not implemented

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7]	RO	1	Secure non-invasive debug features implemented
[6]	RO	Logical result of (**DBGEN** OR **NIDEN**)	Secure non-invasive debug enabled
[5]	RO	1	Secure invasive debug features implemented
[4]	RO	State of **DBGEN** signal	Secure invasive debug enabled
[3]	RO	0	Nonsecure non-invasive debug features not implemented
[2]	RO	0	Nonsecure non-invasive debug disabled
[1]	RO	0	Nonsecure invasive debug features not implemented
[0]	RO	0	Nonsecure invasive debug disabled

10.7.8 Device Type Register (DEVTYPE)

The Device Type Register is register 1011, at offset `0xFCC`.

DEVTYPE is a read-only register present in all CoreSight architecture compatible components and indicates the type of debug component. Table 10-50 shows the layout of the Device Type Register.

Table 10-50 Device Type Register bit definition

Bits	Access	Value	Description
[31:8]	RO	-	Reserved
[7:4]	RO	`0x1`	Sub Type: Processor/Core.
[3:0]	RO	`0x5`	Main class: Debug Logic.

Copyright © 2006 ARM Limited. All rights reserved.

10.7.9 Peripheral Identification Registers (PERIPHERALID)

The Peripheral Identification Registers are registers 1012-1019, at offsets `0xFD0-0xFEC`.

These registers provide standard information required by all ARM Debug Interface v5 components. Only bits [7:0] of each register are used.

There are eight Peripheral Identification Registers, as shown in Table 10-51.

Table 10-51 Peripheral Identification Registers (except v7C Debug)

Register number	Access	Description	Reference
1012	RO	Peripheral ID4	Table 10-57 on page 10-78
1013	RO	Reserved for Peripheral ID5	-
1014	RO	Reserved for Peripheral ID6	-
1015	RO	Reserved for Peripheral ID7	-
1016	RO	Peripheral ID0	Table 10-53 on page 10-77
1017	RO	Peripheral ID1	Table 10-54 on page 10-77
1018	RO	Peripheral ID2	Table 10-55 on page 10-78
1019	RO	Peripheral ID3	Table 10-56 on page 10-78

Copyright © 2006 ARM Limited. All rights reserved.

These registers contain the fields shown in Table 10-52.

Table 10-52 Fields in the Peripheral Identification Registers

Name	Size	Description
4KB Count	4 bits	Log_2 of the number of 4KB blocks occupied by the device. ARMv7 debug registers occupy a single 4KB block, so this field is always `0x0`.
JEP106 code	4+7 bits	Identifies the designer of the device. This consists of a 4-bit continuation code (bank number) and a 7-bit identity code. For implementations designed by ARM Limited, the continuation code is `0x4` (bank 5) and the identity code is `0x3B`. For details, see *JEP106M, Standard Manufacture's Identification Code*.
Part Number	12 bits	Part number for the device. Each designer keeps its own part number list.
Revision	4 bits	Starts at `0x0` and increments by 1 at both major (functionality change) and minor (bug fix) revisions.
RevAnd	4 bits	Manufacturer Revision Number. Indicates a late modification to the device, usually as a result of an Engineering Change Order. Starts at `0x0` and increments by the integrated circuit manufacturer on metal fixes.
Customer modified	4 bits	Indicates an endorsed modification to the device. If the system designer cannot modify the RTL of the processor designer, these bits read as zero.

For more information about these fields, see the *ARM Debug Interface v5 Architecture Specification*.

Table 10-53 Peripheral ID0 bit definition

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7:0]	RO	IMPLEMENTATION DEFINED	Part Number[7:0]

Table 10-54 Peripheral ID1 bit definition

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7:4]	RO	IMPLEMENTATION DEFINED	JEP106 Identity Code [3:0]
[3:0]	RO	IMPLEMENTATION DEFINED	Part Number[11:8]

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-55 Peripheral ID2 bit definition

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7:4]	RO	IMPLEMENTATION DEFINED	Revision
[3]	RO	1	Always 1
[2:0]	RO	IMPLEMENTATION DEFINED	JEP106 Identity Code [6:4]

Table 10-56 Peripheral ID3 bit definition

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7:4]	RO	IMPLEMENTATION DEFINED	RevAnd
[3:0]	RO	IMPLEMENTATION DEFINED	Customer Modified

Table 10-57 Peripheral ID4 bit definition

Bits	Access	Value	Description
[31:8]	RAZ	-	Reserved
[7:4]	RO	`0x0`	4KB count, always `0x0`
[3:0]	RO	IMPLEMENTATION DEFINED	JEP106 Continuation Code

Copyright © 2006 ARM Limited. All rights reserved.

10.7.10 Component Identification Registers (COMPONENTID)

The Component Identification Registers are registers 1020-1023, at offsets `0xFF0-0xFFC`.

These registers identify the processor as an ARM Debug Interface v5 Component. For more information, see the *ARM Debug Interface v5 Architecture Specification*.

Only bits [7:0] of each register are used. Bits [31:0] read as zero. The values in the registers are fixed.

There are four Component Identification Registers, as shown in Table 10-58.

Table 10-58 Component Identification Registers bit definitions

Register number	Bits	Access	Value	Description
1020	[7:0]	RO	`0x0D`	Preamble
1021	[3:0]	RO	`0x0`	Preamble
	[7:4]	RO	`0x9`	Component class, ARM Debug Interface component
1022	[7:0]	RO	`0x05`	Preamble
1023	[7:0]	RO	`0xB1`	Preamble

 Copyright © 2006 ARM Limited. All rights reserved.

10.8 Core-based performance counters registers

v6, v6.1 These registers are not defined in ARMv6.

v7 The Core-based Performance Counters are an optional feature in ARMv7.

This section contains the following subsections:

- *Performance Monitor Control Register (PMNC)*
- *Interrupt Enable Set Register (INTENS)* on page 10-83
- *Interrupt Enable Clear Register (INTENC)* on page 10-84
- *Count Enable Set Register (CNTENS)* on page 10-85
- *Count Enable Clear Register (CNTENC)* on page 10-86
- *Overflow Flag Status Register (FLAG)* on page 10-87
- *Software Increment Register (SWINCR)* on page 10-89
- *Cycle Count Register (CCNT)* on page 10-89
- *Performance Counter Selection Register (PMNXSEL)* on page 10-90
- *Event Select Register (EVTSELX)* on page 10-90
- *Performance Count Registers (PMNX)* on page 10-91
- *User Enable Register (USEREN)* on page 10-91.

10.8.1 Performance Monitor Control Register (PMNC)

Table 10-59 shows the layout of the Performance Monitor Control Register.

Table 10-59 Performance Monitor Control Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description
[31:24]	RO	IMP. DEF. [a]	IMP	*Implementer code, bits [31:24]* on page 10-83
[23:16]	RO	IMP. DEF. [a]	IDCODE	*Identification code, bits [23:16]* on page 10-82
[15:11]	RO	IMP. DEF. [a]	N	*Number of counters, bits [15:11]* on page 10-82
[10:6]	RAZ/SBZP	-	-	Reserved
[5]	RW	0	DP	*Disable CCNT when prohibited, bit [5]* on page 10-82
[4]	RW	0	X	*Export Enable, bit [4]* on page 10-82
[3]	RW	0	D	*Clock Divider, bit [3]* on page 10-81
[2]	RAZ/W	0	C	*Clock Counter reset, bit [2]* on page 10-81
[1]	RAZ/W	0	P	*Performance Counter reset, bit [1]* on page 10-81
[0]	RW	0	E	*Enable, bit [0]* on page 10-81

a. IMPLEMENTATION DEFINED.

Copyright © 2006 ARM Limited. All rights reserved.

Enable, bit [0]

The meanings of the Enable bit values are as follows:

0 all counters, including the *Clock Counter* (CCNT), are disabled

1 all counters are enabled.

Performance counter overflow IRQs are only signaled when the enable bit is 1.

Performance Counter reset, bit [1]

This bit is write-only. It always reads as zero.

The effects of writing to the Performance Counter reset bit are as follows:

0 no action

1 reset all performance counters, not including CCNT, to zero.

Note

Resetting the performance counters does not clear any overflow flags to 0. See *Overflow Flag Status Register (FLAG)* on page 10-87 for details.

Clock Counter reset, bit [2]

This bit is write-only. It always reads as zero.

The effects of writing to the Clock Counter reset bit are as follows:

0 no action

1 reset CCNT to zero.

Note

Resetting CCNT does not clear the CCNT overflow flag to 0. See *Overflow Flag Status Register (FLAG)* on page 10-87 for details.

Clock Divider, bit [3]

The meanings of the Clock Divider bit values are as follows:

0 CCNT counts every clock cycle when enabled

1 CCNT counts once every 64 clock cycles when enabled.

Copyright © 2006 ARM Limited. All rights reserved.

Export Enable, bit [4]

The meanings of the Export Enable bit values are as follows:

0 export of events is disabled

1 export of events is enabled.

This bit exists to allow events to be exported to another debug device, such as the Embedded Trace Macrocell (ETM), over an event bus. If the implementation does not implement such an event bus, this bit reads as zero and ignores writes.

This bit does not affect the generation of performance counter interrupts, which can be implemented as a signal exported from the core to an interrupt controller.

Disable CCNT when prohibited, bit [5]

The meanings of the Disable CCNT when prohibited bit values are as follows:

0 count is enabled in prohibited regions

1 count is disabled in prohibited regions.

Prohibited regions are defined as regions where event counting would be prohibited. For example, if the **SPNIDEN** input to the processor is LOW, the Secure world is a prohibited region. See *Authentication signals* on page 6-3 for a description of **SPNIDEN**.

Note

This bit exists only to allow a nonsecure process to discard cycle counts from being accumulated during periods that the other counts are prohibited due to security prohibitions. It is not a control to enhance security. The function of this bit is to avoid corruption of the count. See also *Interaction with Security Extensions* on page 9-7.

Number of counters, bits [15:11]

The meanings of the Number of counters values are as follows:

b00000 no counters implemented

b00001 1 counter implemented

b00010 2 counters implemented

... ...

b11111 31 counters implemented.

It is permissible to implement only the CCNT register.

Identification code, bits [23:16]

The identification code is an IMPLEMENTATION DEFINED value. Each implementer must maintain a list of identification codes. The list of identification codes is unique to each implementer. A unique implementation can be determined from the combination of the implementer code and the identification code.

Copyright © 2006 ARM Limited. All rights reserved.

Implementer code, bits [31:24]

The implementer code is a value allocated by ARM Limited. Values have the same interpretation as bits [31:24] of the CP15 Main ID register. See the *ARM Architecture Reference Manual* for a list of valid values.

10.8.2 Interrupt Enable Set Register (INTENS)

Each performance counter register (PMNX) and the CCNT can generate an interrupt on overflow. The *i*th bit of the interrupt enable register controls whether PMNi register generates an interrupt on overflow. On reads this register returns the current setting. On writes, interrupts can be enabled. Interrupts can be disabled through the Interrupt Enable Clear Register, see *Interrupt Enable Clear Register (INTENC)* on page 10-84.

The instructions that access the INTENS register are always UNDEFINED in User Mode, even if the USEREN flag is set to 1, see *User Enable Register (USEREN)* on page 10-91.

Table 10-60 shows the layout of the INTENS. In this table, N is the value of PMNC[15:11].

Table 10-60 Interrupt Enable Set Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description	Reference
[31]	RW	UNPREDICTABLE	C	Interrupt on CCNT Overflow Enabled / Enable Interrupt	Table 10-62 on page 10-84
[30:N]	UNP/SBZP	-	-	Reserved	-
[N-1]	RW	UNPREDICTABLE	P_{N-1}	Interrupt on PMN_{N-1} Overflow Enabled / Enable Interrupt	Table 10-61 on page 10-84
...	...	...	...	...	
[1]	RW	UNPREDICTABLE	P_1	Interrupt on PMN_1 Overflow Enabled / Enable Interrupt	
[0]	RW	UNPREDICTABLE	P_0	Interrupt on PMN_0 Overflow Enabled / Enable Interrupt	

Note

The interrupt that can be asserted in this case is expected to be exported from the core, to allow it to be factored into a system interrupt controller if applicable. As a result, more levels of control of the interrupt generated are expected to exist in the system.

The contents of this register are UNPREDICTABLE on core reset. To avoid spurious interrupts being generated, the interrupt enable values must be set before enabling any of the counters. Interrupts are not signaled if the Enable bit in the PMNC is clear (0).

Once an interrupt is signaled, it can be removed by clearing the overflow flag for the counter in the Overflow Flag Status Register. See *Overflow Flag Status Register (FLAG)* on page 10-87.

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-61 Meaning of the Interrupt on PMN_X Overflow Enable values

Value	Meaning on read	Action on write
0	Interrupt disabled	No action
1	Interrupt enabled	Enable Interrupt

Table 10-62 Meaning of the Interrupt on CCNT Overflow Enable values

Value	Meaning on read	Action on write
0	Interrupt disabled	No action
1	Interrupt enabled	Enable Interrupt

10.8.3 Interrupt Enable Clear Register (INTENC)

Table 10-63 shows the layout of the INTENC. In this table, N is the value of PMNC[15:11].

Table 10-63 Interrupt Enable Clear Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description	Reference
[31]	RW	UNPREDICTABLE	C	Interrupt on CCNT Overflow Enabled or Disable Interrupt	Table 10-65 on page 10-85
[30:N]	UNP/SBZP	-	-	Reserved	-
[N-1]	RW	UNPREDICTABLE	P_{N-1}	Interrupt on PMN_{N-1} Overflow Enabled or Disable Interrupt	Table 10-64 on page 10-85
...	...	...	...	...	
[1]	RW	UNPREDICTABLE	P_1	Interrupt on PMN_1 Overflow Enabled or Disable Interrupt	
[0]	RW	UNPREDICTABLE	P_0	Interrupt on PMN_0 Overflow Enabled or Disable Interrupt	

 Copyright © 2006 ARM Limited. All rights reserved.

The instructions that access the INTENC register are always UNDEFINED in User Mode, even if the USEREN flag is set to 1, see *User Enable Register (USEREN)* on page 10-91.

Table 10-64 Meaning of the Interrupt on PMN_X Overflow Enable values

Value	Meaning on read	Action on write
0	Interrupt disabled	No action
1	Interrupt enabled	Disable Interrupt

Table 10-65 Meaning of the Interrupt on CCNT Overflow Enable values

Value	Meaning on read	Action on write
0	Interrupt disabled	No action
1	Interrupt enabled	Disable Interrupt

10.8.4 Count Enable Set Register (CNTENS)

Table 10-66 shows the layout of the CNTENS. In this table, N is the value of PMNC[15:11].

Table 10-66 Interrupt Enable Set Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description	Reference
[31]	RW	UNPREDICTABLE	C	CCNT Enabled or Enable CCNT	Table 10-68 on page 10-86
[30:N]	UNP/SBZP	-	-	Reserved	-
[N-1]	RW	UNPREDICTABLE	P_{N-1}	PMN_{N-1} Enabled or Enable counter PMN_{N-1}	Table 10-67 on page 10-86
...	...	...	...	...	
[1]	RW	UNPREDICTABLE	P_1	PMN_1 Enabled or Enable counter PMN_1	
[0]	RW	UNPREDICTABLE	P_0	PMN_0 Enabled or Enable counter PMN_0	

Copyright © 2006 ARM Limited. All rights reserved.

The instructions that access the CNTENS register are always UNDEFINED in User Mode, even if the USEREN flag is set to 1, see *User Enable Register (USEREN)* on page 10-91.

Table 10-67 Meaning of the PMN_X Enable values

Value	Meaning on read	Action on write
0	PMN_X counter disabled	No action
1	PMN_X counter enabled	Enable counter

Table 10-68 Meaning of the CCNT Enable values

Value	Meaning on read	Action on write
0	CCNT disabled	No action
1	CCNT enabled	Enable CCNT

10.8.5 Count Enable Clear Register (CNTENC)

Table 10-69 shows the layout of the CNTENC. In this table, N is the value of PMNC[15:11].

Table 10-69 Interrupt Enable Set Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description	Reference
[31]	RW	UNPREDICTABLE	C	CCNT Enabled or Disable CCNT	Table 10-71 on page 10-87
[30:N]	UNP/SBZP	-	-	Reserved	-
[N-1]	RW	UNPREDICTABLE	P_{N-1}	PMN_{N-1} Enabled or Disable counter PMN_{N-1}	Table 10-70 on page 10-87
...	...	...	...	...	
[1]	RW	UNPREDICTABLE	P_1	PMN_1 Enabled or Disable counter PMN_1	
[0]	RW	UNPREDICTABLE	P_0	PMN_0 Enabled or Disable counter PMN_0	

The instructions that access the CNTENC register are always UNDEFINED in User Mode, even if the USEREN flag is set to 1, see *User Enable Register (USEREN)* on page 10-91.

 Copyright © 2006 ARM Limited. All rights reserved.

Note

The Enable bit in the PMNC can be used to override the settings in this register and disable all performance counters including CCNT, see *Enable, bit [0]* on page 10-81. The counter enable register retains its value when the Enable bit is 0, even though its settings are ignored.

Table 10-70 Meaning of the PMN_X Disable values

Value	Meaning on read	Action on write
0	PMN_X counter disabled	No action
1	PMN_X counter enabled	Disable PMN_X counter

Table 10-71 Meaning of the CCNT Disable values

Value	Meaning on read	Action on write
0	CCNT disabled	No action
1	CCNT enabled	Disable CCNT

10.8.6 Overflow Flag Status Register (FLAG)

Table 10-72 shows the layout of the FLAG Register. In this table, N is the value of PMNC[15:11].

Table 10-72 Overflow Flag Status Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description	Reference
[31]	RW	UNPREDICTABLE	C	CCNT overflowed or Clear CCNT overflow to 0	Table 10-74 on page 10-88
[30:N]	UNP/SBZP	-	-	Reserved	-

Copyright © 2006 ARM Limited. All rights reserved.

Table 10-72 Overflow Flag Status Register bit definitions (continued)

Bits	Access	Core reset value	Mnemonic	Description	Reference
[N-1]	RW	UNPREDICTABLE	P_{N-1}	PMN_{N-1} counter overflowed or Clear PMN_{N-1} counter overflow to 0	Table 10-73
...	...	...	...	...	
[1]	RW	UNPREDICTABLE	P_1	PMN_1 counter overflowed or Clear PMN_1 counter overflow to 0	
[0]	RW	UNPREDICTABLE	P_0	PMN_0 counter overflowed or Clear PMN_0 counter overflow to 0	

Note

The overflow flag values for individual counters are retained until cleared to 0 by a write to the Overflow Flag Status Register or processor reset, even if the counter is later disabled by writing to the Counter Enable Set Register or through the Enable bit in the PMNC. The overflow flags are also not cleared to 0 if the counters are reset through the Performance Counter reset / Clock Counter reset bits in the PMNC.

Table 10-73 Meaning of the PMN_X Overflowed values

Value	Meaning on read	Action on write
0	PMN_X counter has not overflowed	No action
1	PMN_X counter has overflowed	Clear PMN_X counter overflow status to 0

Table 10-74 Meaning of the CCNT Overflowed values

Value	Meaning on read	Action on write
0	CCNT has not overflowed	No action
1	CCNT has overflowed	Clear CCNT overflow status to 0

Copyright © 2006 ARM Limited. All rights reserved.

10.8.7 Software Increment Register (SWINCR)

Table 10-75 shows the layout of the SWINCR. In this table, N is the value of PMNC[15:11].

Table 10-75 Software Increment Register bit definitions

Bits	Access	Mnemonic	Description
[31:N]	UNPREDICTABLE/SBZ	-	Reserved
[N-1]	UNPREDICTABLE/WO	PMN_{N-1}	*Increment PMNX, bits [N-1:0]*
...	UNPREDICTABLE/WO	...	
[1]	UNPREDICTABLE/WO	PMN_1	
[0]	UNPREDICTABLE/WO	PMN_0	

Increment PMN_X, bits [N-1:0]

These bits are write-only. They return UNPREDICTABLE values on reads.

The effect of writing to each of the PMN_X bits of the register is shown in Table 10-76.

Table 10-76 Effects of writes to PMN_X

Value	Event selected for PMN_X	Effect on write
0	X	No action
1	`0x00`, Software count	Increment PMN_X
	Not `0x00`	UNPREDICTABLE

10.8.8 Cycle Count Register (CCNT)

CCNT counts the number of clock cycles since the register was reset, see *Clock Counter reset, bit [2]* on page 10-81.

Table 10-77 shows the layout of the CCNT.

Table 10-77 Cycle Count Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description
[31:0]	RW	UNPREDICTABLE	CCNT	Cycle Count Register

Copyright © 2006 ARM Limited. All rights reserved.

10.8.9 Performance Counter Selection Register (PMNXSEL)

Table 10-78 shows the layout of the PMNXSEL.

Table 10-78 Cycle Count Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description
[31:5]	RAZ/SBZP	-	-	Reserved
[4:0]	RW	UNPREDICTABLE	SEL	*Selection value, bits [4:0]*

Selection value, bits [4:0]

The selection value determines which registers are mapped into EVTSELX and PMNX. Writing a value greater than or equal to N, the value in PMNC[15:11], gives UNPREDICTABLE results. For more information see *Number of counters, bits [15:11]* on page 10-82.

The meaning of the Number of counters values is as follows:

b00000 PMNX and EVTSELX are mapped to PMN0 and EVTSEL0

b00001 PMNX and EVTSELX are mapped to PMN1 and EVTSEL1

... ...

b11110 PMNX and EVTSELX are mapped to PMN30 and EVTSEL30

b11111 Reserved.

10.8.10 Event Select Register (EVTSELX)

Table 10-79 shows the layout of the EVTSELX.

Table 10-79 Event Selection Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description
[31:8]	RAZ/SBZP	-	-	Reserved
[7:0]	RW	UNPREDICTABLE	evtCount	*Event number, bits [7:0]*

Event number, bits [7:0]

The event numbers are used in the EVTSELX register to determine what events are used to cause counts. It is envisaged that a common set of events can be assigned that all implementers following this approach must adopt, together with a set of IMPLEMENTATION DEFINED features. A range of events associated with a linkage to debug is provided as separate from the common features, and it is IMPLEMENTATION DEFINED whether these events are supported.

Copyright © 2006 ARM Limited. All rights reserved.

The event triggers are split into two ranges:

`0x00-0x3F` common features

`0x40-0xFF` IMPLEMENTATION DEFINED features.

A more complete description of the event numbers is given in *Event numbers* on page 9-13.

10.8.11 Performance Count Registers (PMNX)

Table 10-80 shows the layout of the PMNX Register.

Table 10-80 Performance Count Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description
[31:0]	RW	UNPREDICTABLE	PMNX	Performance Count Register

The register read as PMNX depends on the value in the PMNXSEL register, see *Performance Counter Selection Register (PMNXSEL)* on page 10-90.

Note

- A read of the PMNX Register returns the current value of the register.
- You can write to PMNX even when the counter is disabled, regardless of whether the counter is disabled by CNTENS, PMNC or by non-invasive debug authentication.

10.8.12 User Enable Register (USEREN)

Table 10-81 shows the layout of the USEREN.

Table 10-81 User Enable Register bit definitions

Bits	Access	Core reset value	Mnemonic	Description
[31:1]	RAZ/SBZP	-	-	Reserved
[0]	RW	0	EN	User Mode Enable

The meanings of the User Mode Enable values are as follows:

0 User Mode access to performance counters disabled

1 User Mode access to performance counters enabled.

Certain `MCR` and `MRC` instructions used to access the performance counters are UNDEFINED in User Mode when User Mode access to the performance counters is disabled. For more information see *Access permissions* on page 9-12.

Copyright © 2006 ARM Limited. All rights reserved.

 Copyright © 2006 ARM Limited. All rights reserved.

Glossary

Abort
A mechanism that indicates to a core that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a Prefetch or Data Abort, and an internal or External Abort.

See also Data Abort, External Abort and Prefetch Abort.

Abort model
An abort model is the defined behavior of an ARM processor in response to a Data Abort exception. Different abort models behave differently with regard to load and store instructions that specify base register write-back.

Addressing modes
A mechanism, shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.

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.

Copyright © 2006 ARM Limited. All rights reserved.

AXI is targeted at high performance, high clock frequency system designs and includes a number of features that make it very suitable for high speed sub-micron interconnect.

Advanced High-performance Bus (AHB)

A bus protocol with a fixed pipeline between address/control and data phases. It only supports a subset of the functionality provided by the AMBA AXI protocol. The full AMBA AHB protocol specification includes a number of features that are not commonly required for master and slave IP developments and ARM Limited recommends only a subset of the protocol is usually used. This subset is defined as the AMBA AHB-Lite protocol.

See also Advanced Microcontroller Bus Architecture.

Advanced Microcontroller Bus Architecture (AMBA)

A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM 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.

Aligned A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

AMBA *See* Advanced Microcontroller Bus Architecture.

APB *See* Advanced Peripheral Bus.

Architecture

The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

ARM instruction

A word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.

ARM state

A processor that is executing ARM (32-bit) word-aligned instructions is operating in ARM state.

AXI *See* Advanced eXtensible Interface.

Banked registers

Those physical registers whose use is defined by the current processor mode. The banked registers are r8 to r14.

Copyright © 2006 ARM Limited. All rights reserved.

Base register

A register specified by a load or store instruction that is used to hold the base value for the instruction's address calculation. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register write-back

Updating the contents of the base register used in an instruction target address calculation so that the modified address is changed to the next higher or lower sequential address in memory. This means that it is not necessary to fetch the target address for successive instruction transfers and enables faster burst accesses to sequential memory.

BE-8 Big-endian view of memory in a byte-invariant system.

See also BE-32, LE, Byte-invariant and Word-invariant.

BE-32 Big-endian view of memory in a word-invariant system.

See also BE-8, LE, Byte-invariant and Word-invariant.

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.

Branch prediction

The process of predicting if conditional branches are to be taken or not in pipelined processors. Successfully predicting if branches are to be taken enables the processor to prefetch the instructions following a branch before the condition is fully resolved. Branch prediction can be done in software or by using custom hardware. Branch prediction techniques are categorized as static, in which the prediction decision is decided before run time, and dynamic, in which the prediction decision can change during program execution.

Breakpoint

A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.

See also Watchpoint.

 Copyright © 2006 ARM Limited. All rights reserved.

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.

Byte-invariant

In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access.
The ARM architecture supports byte-invariant systems in ARMv6 and later versions. When byte-invariant support is selected, unaligned halfword and word memory accesses are also supported. Multi-word accesses are expected to be word-aligned.

See also Word-invariant.

Cache A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions and/or data. This is done to greatly increase the average speed of memory accesses and so improve processor performance.

Cache hit

A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.

Cache line

The basic unit of storage in a cache. It is always a power of two words in size (usually four or eight words), and is required to be aligned to a suitable memory boundary.

Cache miss

A memory access that cannot be processed at high speed because the instruction/data it addresses is not in the cache and a main memory access is required.

Coherency

See Memory coherency.

Cold reset

Also known as power-up reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to enable the system to be used again. In other cases, only a warm reset is required.

See also Warm reset.

Communications channel

The hardware used for communicating between the software running on the processor, and an external host, using the debug interface. When this communication is for debug purposes, it is called the Debug Comms Channel. In an ARMv6 compliant core, the communications channel includes the Data Transfer Register, some bits of the Data Status and Control Register, and the external debug interface controller, such as the DBGTAP controller in the case of the JTAG interface.

Copyright © 2006 ARM Limited. All rights reserved.

Context The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the physical address range that it can access in memory and the associated memory access permissions.

See also Fast context switch.

Control bits
The bottom eight bits of a Program Status Register. The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Coprocessor
A processor that supplements the main processor. It carries out additional functions that the main processor cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.

Core A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

Core reset
See Warm reset.

CoreSight
The infrastructure for monitoring, tracing, and debugging a complete system on chip.

CPSR *See* Current Program Status Register

Current Program Status Register (CPSR)
The register that holds the current operating processor status.

DAP *See* Debug Access Port.

Data Abort
An indication from a memory system to the core of an attempt to access an illegal data memory location. An exception must be taken if the processor attempts to use the data that caused the abort.

See also Abort, External Abort, and Prefetch Abort.

Data cache
A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used data. This is done to greatly increase the average speed of memory accesses and so improve processor performance.

DBGTAP
See Debug Test Access Port.

Debug Access Port (DAP)
A component that provides a means for the External Debugger to make accesses to the debug registers over the External Debug Interface. for example a JTAG-based interface. In ARMv6 implementations, the Debug Access Port and External Debug Interface are always tightly integrated parts of the ARM processor macrocell. In ARMv7 this is a design choice.

Debug bus
A SoC-wide bus onto which a port for the Debug Registers Interface may be connected.

 Copyright © 2006 ARM Limited. All rights reserved.

Debug Comms Channel

One word full-duplex communication channel between software running on the core and the External Debugger. Accessible through CP14 debug instructions on the inside, and through the External Debug Interface on the outside.

Debugger

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

Debug Monitor

Debugging software that runs on the target (ARM processor).

Debug Registers Interface

An interface to the debug registers.

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 **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

Embedded Trace Buffer (ETB)

The ETB provides on-chip storage of trace data using a configurable sized RAM.

Embedded Trace Macrocell (ETM)

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

Endianness

Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

See also Little-endian and Big-endian

ETB *See* Embedded Trace Buffer.

ETM See *Embedded Trace Macrocell*.

External Debug Interface

Interface into a processor core that provides access to the Debug Registers Interface and signals to integrate the processor core with the rest of the system. The External Debugger must have means, such as a Debug Access Port, to access the External Debug Interface in order to access the debug resources.

The External Debug Interface also contains the signals and connections required for the processor to integrate with other debug components in the system, such as Cross-Triggering Interfaces and Authentication Modules.

Parts of the External Debug Interface may be tightly integrated with the Debug Access Port. External, in this context, means external to the ARM processor core, not necessarily external to the macrocell, or to the chip.

Copyright © 2006 ARM Limited. All rights reserved.

External Debugger

Any combination of hardware and software external to the ARM processor whose purpose is to drive the External Debug Interface, for example an emulator box connected to a Debug Access Port, plus a personal computer, plus debugging software such as RealView Debugger.

Event
An observable condition that can be used by an ETM to control aspects of a trace.

Exception

A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

Exception vector

See Interrupt vector.

External Abort

An indication from an external memory system to a core that the value associated with a memory access is invalid. An external abort is caused by the external memory system as a result of attempting to access invalid memory.

See also Abort, Data Abort and Prefetch Abort.

Fast context switch

In a multitasking system, the point at which the time-slice allocated to one process stops and the one for the next process starts. If processes are switched often enough, they can appear to a user to be running in parallel, in addition to being able to respond quicker to external events that might affect them.

In ARM processors, a fast context switch is caused by the selection of a non-zero PID value to switch the context to that of the next process. A fast context switch causes each Virtual Address for a memory access, generated by the ARM processor, to produce a Modified Virtual Address that is sent to the rest of the memory system to be used in place of a normal Virtual Address. For some cache control operations Virtual Addresses are passed to the memory system as data. In these cases no address modification takes place.

See also Fast Context Switch Extension.

Fast Context Switch Extension (FCSE)

An extension to the ARM architecture that enables cached processors with an MMU to present different addresses to the rest of the memory system for different software processes, even when those processes are using identical addresses.

See also Fast context switch, Modified Virtual Address.

FCSE
See Fast Context Switch Extension.

Halfword

A 16-bit data item.

Copyright © 2006 ARM Limited. All rights reserved.

Halting Debug-mode

One of two mutually exclusive debug modes. In Halting Debug-mode all processor execution halts when a breakpoint or watchpoint is encountered. All processor state, coprocessor state, memory and input/output locations can be examined and altered by the JTAG interface.

See also Monitor Debug-mode.

High vectors

Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

Host A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

IMB *See* Instruction Memory Barrier

IMPLEMENTATION DEFINED

The behavior is not architecturally defined, but is defined and documented by individual implementations.

Instruction cache

A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions. This is done to greatly increase the average speed of memory accesses and so improve processor performance.

Instruction Memory Barrier (IMB)

An operation to ensure that the prefetch buffer is flushed of all out-of-date instructions.

Interrupt handler

A program that control of the processor is passed to when an interrupt occurs.

Interrupt vector

One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.

Invalidate

To mark a cache line as being not valid by clearing the valid bit. This must be done whenever the line does not contain a valid cache entry. For example, after a cache flush all lines are invalid.

Jazelle architecture

The ARM Jazelle architecture extends the Thumb and ARM operating states by adding a Java state to the processor. Instruction set support for entering and exiting Java applications, real-time interrupt handling, and debug support for mixed Java/ARM applications is present. When in Java state, the processor fetches and decodes Java bytecodes and maintains the Java operand stack.

Joint Test Action Group (JTAG)

The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

JTAG *See* Joint Test Action Group.

Copyright © 2006 ARM Limited. All rights reserved.

LE Little endian view of memory in both byte-invariant and word-invariant systems.

See also Byte-invariant, Word-invariant.

Line *See* Cache line.

Little-endian

Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

See also Big-endian and Endianness.

Little-endian memory

Memory in which:

- a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address
- a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

See also Big-endian memory.

Macrocell

A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.

Memory coherency

A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.

Memory Management Unit (MMU)

Hardware that controls caches and access permissions to blocks of memory, and translates virtual addresses to physical addresses.

Memory Protection Unit (MPU)

Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not translate virtual addresses to physical addresses.

Microprocessor

See Processor.

Miss *See* Cache miss.

MMU *See* Memory Management Unit.

Modified Virtual Address (MVA)

A Virtual Address produced by the ARM processor can be changed by the current Process ID to provide a *Modified Virtual Address* (MVA) for the MMUs and caches.

See also Fast Context Switch Extension.

Copyright © 2006 ARM Limited. All rights reserved.

Monitor Debug-mode

One of two mutually exclusive debug modes. In Monitor Debug-mode the processor enables a software abort handler provided by the debug monitor or operating system debug task. When a breakpoint or watchpoint is encountered, this enables vital system interrupts to continue to be serviced while normal program execution is suspended.

See also Halting Debug-mode.

MPU *See* Memory Protection Unit.

MVA *See* Modified Virtual Address.

Nonsecure State

A processor that implements Security Extensions is said to be in Nonsecure state if the CP15 NS-bit is 1.

Nonsecure World

A processor that implements Security Extensions is said to be in the Nonsecure world if the CP15 NS-bit is 1 and the current processor mode is not Secure Monitor.

PA *See* Physical Address.

Physical Address (PA)

The MMU performs a translation on *Modified Virtual Addresses* (MVA) to produce the *Physical Address* (PA) that is given to the AMBA bus to perform an external access. The PA is also stored in the data cache to avoid the necessity for address translation when data is cast out of the cache.

See also Fast Context Switch Extension.

Power-on reset

See Cold reset.

Prefetching

In pipelined processors, the process of fetching instructions from memory to fill up the pipeline before the preceding instructions have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

Prefetch Abort

An indication from a memory system to the core that an instruction has been fetched from an illegal memory location. An exception must be taken if the processor attempts to execute the instruction. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data Abort, External Abort and Abort.

Processor

A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Read Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions `LDM`, `LDRD`, `LDC`, `LDR`, `LDRT`, `LDRSH`, `LDRH`, `LDRSB`, `LDRB`, `LDRBT`, `LDREX`, `RFE`, `STREX`, `SWP`, and `SWPB`, and the Thumb instructions `LDM`, `LDR`, `LDRSH`, `LDRH`, `LDRSB`, `LDRB`, and `POP`.

Copyright © 2006 ARM Limited. All rights reserved.

Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Region A partition of instruction or data memory space.

Reserved

A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces UNPREDICTABLE results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

Saved Program Status Register (SPSR)

The register that holds the CPSR of the task immediately before the exception occurred that caused the switch to the current mode.

SBZ *See* Should Be Zero.

SBZP *See* Should Be Zero or Preserved.

Secure State

A processor that implements Security Extensions is said to be in Secure state if the CP15 NS-bit is set to 0.

Secure World

A processor that implements the Security Extensions is said to be in the Secure world if the CP15 NS-bit is set to 0 or if the current processor mode is Secure Monitor.

Should Be Zero (SBZ)

Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces Unpredictable results.

Should Be Zero or Preserved (SBZP)

Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

SoC System-on-Chip.

SPSR *See* Saved Program Status Register

Synchronization primitive

The memory synchronization primitive instructions are those instructions that are used to ensure memory synchronization. That is, the `LDREX`, `STREX`, `SWP`, and `SWPB` instructions.

Tag The upper portion of a block address used to identify a cache line within a cache. The block address from the CPU is compared with each tag in a set in parallel to determine if the corresponding line is in the cache. If it is, it is said to be a cache hit and the line can be fetched from cache. If the block address does not correspond to any of the tags, it is said to be a cache miss and the line must be fetched from the next level of memory.

TCM *See* Tightly Coupled Memory.

Thumb instruction

A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

 Copyright © 2006 ARM Limited. All rights reserved.

Thumb state
A processor that is executing Thumb (16-bit) halfword aligned instructions is operating in Thumb state.

Tightly Coupled Memory (TCM)
An area of low latency memory that provides predictable instruction execution or data load timing in cases where deterministic performance is required. TCMs are suited to holding:

- critical routines such as for interrupt handling
- scratchpad data
- data types whose locality is not suited to caching
- critical data structures, such as interrupt stacks.

TLB *See* Translation Look-aside Buffer.

TPA See *Trace Port Analyzer*.

Trace port
A port on a device, such as a processor or ASIC, used to output trace information.

Trace Port Analyzer (TPA)
A hardware device that captures trace information output on a trace port. This can be a low-cost product designed specifically for trace acquisition, or a logic analyzer.

Translation Lookaside Buffer (TLB)
A cache of recently used page table entries that avoid the overhead of page table walking on every memory access. Part of the Memory Management Unit.

Translation table
A table, held in memory, that contains data that defines the properties of memory areas of various fixed sizes.

Translation table walk
The process of doing a full translation table lookup. It is performed automatically by hardware.

Unaligned
A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

Undefined
Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more details on ARM exceptions.

UNP *See* UNPREDICTABLE.

UNPREDICTABLE
For reads, the data returned when reading from this location is unpredictable, and can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. UNPREDICTABLE instructions must not halt or hang the processor, or any part of the system.

v6 Debug
Debug component of the ARMv6 Debug Architecture.

 Copyright © 2006 ARM Limited. All rights reserved.

v6.1 Debug
Debug component of the ARMv6.1 Debug Architecture.

VA *See* Virtual Address.

Virtual Address (VA)
The MMU uses its page tables to translate a Virtual Address into a Physical Address. The processor executes code at the Virtual Address, that might be located elsewhere in physical memory.

See also Fast Context Switch Extension, Modified Virtual Address, and Physical Address.

Warm reset
Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

Watchpoint
A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to enable inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested. *See also* Breakpoint.

WB *See* Write-back.

Word A 32-bit data item.

Word-invariant
In a word-invariant system, the address of each byte of memory changes when switching between little-endian and big-endian operation, in such a way that the byte with address A in one endianness has address A EOR 3 in the other endianness. As a result, each aligned word of memory always consists of the same four bytes of memory in the same order, regardless of endianness. The change of endianness occurs because of the change to the byte addresses, not because the bytes are rearranged.

The ARM architecture supports word-invariant systems in ARMv3 and later versions. When word-invariant support is selected, the behavior of load or store instructions that are given unaligned addresses is instruction-specific, and is in general not the expected behavior for an unaligned access. It is recommended that word-invariant systems use the endianness that produces the desired byte addresses at all times, apart possibly from very early in their reset handlers before they have set up the endianness, and that this early part of the reset handler must use only aligned word memory accesses.

See also Byte-invariant.

Write Writes are defined as operations that have the semantics of a store. That is, the ARM instructions `SRS`, `STM`, `STRD`, `STC`, `STRT`, `STRH`, `STRB`, `STRBT`, `STREX`, `SWP`, and `SWPB`, and the Thumb instructions `STM`, `STR`, `STRH`, `STRB`, and `PUSH`.

Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

 Copyright © 2006 ARM Limited. All rights reserved.

Write-back (WB)

In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. This is also known as copyback.

Write buffer

A block of high-speed memory, arranged as a FIFO buffer, between the data cache and main memory, whose purpose is to optimize stores to main memory.

Write-through (WT)

In a write-through cache, data is written to main memory at the same time as the cache is updated.

WT *See* Write-through.

Copyright © 2006 ARM Limited. All rights reserved.