Cortex-M3™

Revision: r0p0

Technical Reference Manual

ARM®

ARM DDI 0337B

Cortex-M3
Technical Reference Manual

Release Information

Change History

Date	Issue	Confidentiality	Change
15 December 2005	A	Confidential	First Release
13 January 2006	B	Non-Confidential	Confidentiality status amended

Proprietary Notice

Confidentiality Status

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

Product Status

The information in this document is Final (information on a developed product).

Web Address

`http://www.arm.com`

Contents
Cortex-M3 Technical Reference Manual

Preface
About this manual xviii
Feedback xxiii

Chapter 1 Introduction
1.1 About the processor 1-2
1.2 Components of the processor 1-4
1.3 Configurable options 1-12
1.4 Instruction set summary 1-13

Chapter 2 Programmer's Model
2.1 About the programmer's model 2-2
2.2 Privileged access and User access 2-3
2.3 Registers 2-4
2.4 Data types 2-10
2.5 Memory formats 2-11
2.6 Instruction set 2-13

Chapter 3 System Control
3.1 Summary of processor registers 3-2

Chapter 4 Memory Map
4.1 About the memory map 4-2
4.2 Bit-banding 4-5
4.3 ROM memory table 4-8

Chapter 5 Exceptions
5.1 About the exception model 5-2
5.2 Exception types 5-3
5.3 Exception priority 5-5
5.4 Privilege and stacks 5-8
5.5 Pre-emption 5-10
5.6 Tail-chaining 5-13
5.7 Late-arriving 5-14
5.8 Exit 5-16
5.9 Resets 5-19
5.10 Exception control transfer 5-23
5.11 Setting up multiple stacks 5-24
5.12 Abort model 5-26
5.13 Activation levels 5-31
5.14 Flowcharts 5-33

Chapter 6 Clocking and Resets
6.1 Cortex-M3 clocking 6-2
6.2 Cortex-M3 resets 6-4
6.3 Cortex-M3 reset modes 6-5

Chapter 7 Power Management
7.1 About power management 7-2
7.2 System power management 7-3

Chapter 8 Nested Vectored Interrupt Controller
8.1 About the NVIC 8-2
8.2 NVIC programmer's model 8-3
8.3 Level versus pulse interrupts 8-39

Chapter 9 Memory Protection Unit
9.1 About the MPU 9-2
9.2 MPU programmer's model 9-3
9.3 MPU access permissions 9-14
9.4 MPU aborts 9-16
9.5 Updating an MPU region 9-17
9.6 Interrupts and updating the MPU 9-20

Chapter 10 Core Debug
10.1 About core debug 10-2

10.2 Core debug registers 10-3
10.3 Core debug access example 10-12
10.4 Using application registers in core debug 10-13

Chapter 11 System Debug
11.1 About system debug 11-2
11.2 System Debug Access 11-3
11.3 System debug programmer's model 11-5
11.4 Flash Patch and Breakpoint 11-6
11.5 Data Watchpoint and Trace 11-13
11.6 Instrumentation Trace Macrocell 11-28
11.7 AHB Access Port 11-37

Chapter 12 Debug Port
12.1 About the Debug Port 12-2
12.2 JTAG-DP 12-3
12.3 SW-DP 12-20
12.4 Common Debug Port (DP) features 12-41
12.5 Debug Port Programmer's Model 12-47

Chapter 13 Trace Port Interface Unit
13.1 About the Trace Port Interface Unit 13-2
13.2 TPIU registers 13-8

Chapter 14 Bus Interface
14.1 About bus interfaces 14-2
14.2 ICode bus interface 14-3
14.3 DCode bus interface 14-5
14.4 System interface 14-6
14.5 External private peripheral interface 14-8
14.6 Access alignment 14-9
14.7 Unaligned accesses that cross regions 14-10
14.8 Bit-band accesses 14-11
14.9 Write buffer 14-12
14.10 Memory attributes 14-13

Chapter 15 Embedded Trace Macrocell
15.1 About the ETM 15-2
15.2 Data tracing 15-6
15.3 ETM Resources 15-7
15.4 Trace output 15-9
15.5 ETM architecture 15-10
15.6 ETM programmer's model 15-14

Chapter 16 Embedded Trace Macrocell Interface
16.1 About the ETM interface 16-2

16.2 CPU ETM interface port descriptions 16-3
16.3 Branch status interface 16-5

Chapter 17 Instruction Timing
17.1 About instruction timing 17-2
17.2 Processor instruction timings 17-3
17.3 Load-store timings 17-7

Appendix A Signal Descriptions
A.1 Clocks A-2
A.2 Resets A-3
A.3 Miscellaneous A-4
A.4 Interrupt interface A-5
A.5 ICode interface A-6
A.6 DCode interface A-8
A.7 System bus interface A-9
A.8 Private Peripheral Bus interface A-10
A.9 ITM interface A-11
A.10 AHB-AP interface A-12
A.11 ETM interface A-13
A.12 Test interface A-15

Glossary

List of Tables
Cortex-M3 Technical Reference Manual

	Change History	ii
Table 1-1	16-bit Cortex-M3 instruction summary	1-13
Table 1-2	32-bit Cortex-M3 instruction summary	1-16
Table 2-1	Application Program Status Register bit assignments	2-6
Table 2-2	Interrupt Program Status Register bit assignments	2-7
Table 2-3	Bit functions of the Execution PSR	2-8
Table 2-4	Nonsupported Thumb instructions	2-13
Table 2-5	Supported Thumb-2 instructions	2-13
Table 3-1	NVIC registers	3-2
Table 3-2	Core debug registers	3-5
Table 3-3	Flash patch register summary	3-6
Table 3-4	DWT register summary	3-7
Table 3-5	ITM register summary	3-9
Table 3-6	AHB-AP register summary	3-10
Table 3-7	Summary of Debug Port registers	3-11
Table 3-8	MPU registers	3-11
Table 3-9	TPIU registers	3-12
Table 3-10	ETM registers	3-13
Table 4-1	Memory interfaces	4-3
Table 4-2	Memory region permissions	4-4
Table 4-3	Cortex-M3 ROM table	4-8
Table 5-1	Exception types	5-3
Table 5-2	Priority-based actions of exceptions	5-5

Table 5-3 Priority grouping 5-7
Table 5-4 Exception entry steps 5-11
Table 5-5 Exception exit steps 5-16
Table 5-6 Exception return behavior 5-18
Table 5-7 Reset actions 5-19
Table 5-8 Reset boot-up behavior 5-20
Table 5-9 Transferring to exception processing 5-23
Table 5-10 Faults 5-27
Table 5-11 Debug faults 5-29
Table 5-12 Fault status and fault address registers 5-30
Table 5-13 Privilege and stack of different activation levels 5-31
Table 5-14 Exception transitions 5-31
Table 5-15 Exception subtype transitions 5-32
Table 6-1 Cortex-M3 processor clocks 6-2
Table 6-2 Cortex-M3 macrocell clocks 6-2
Table 6-3 Reset inputs 6-4
Table 6-4 Reset modes 6-5
Table 7-1 Supported sleep modes 7-3
Table 8-1 NVIC registers 8-3
Table 8-2 Interrupt Controller Type Register bit assignments 8-7
Table 8-3 SysTick Control and Status Register bit assignments 8-8
Table 8-4 SysTick Reload Value Register bit assignments 8-9
Table 8-5 SysTick Current Value Register bit assignments 8-10
Table 8-6 SysTick Calibration Value Register bit assignments 8-11
Table 8-7 Bit functions of the Interrupt Set-Enable Register 8-12
Table 8-8 Bit functions of the Interrupt Clear-Enable Register 8-12
Table 8-9 Bit functions of the Interrupt Set-Pending Register 8-13
Table 8-10 Bit functions of the Interrupt Clear-Pending Registers 8-14
Table 8-11 Bit functions of the Active Bit Register 8-14
Table 8-12 Interrupt Priority Registers 0-31 bit assignments 8-16
Table 8-13 CPUID Base Register bit assignments 8-16
Table 8-14 Interrupt Control State Register bit assignments 8-18
Table 8-15 Vector Table Offset Register bit assignments 8-20
Table 8-16 Application Interrupt and Reset Control Register bit assignments 8-21
Table 8-17 System Control Register bit assignments 8-23
Table 8-18 Configuration Control Register bit assignments 8-24
Table 8-19 System Handler Priority Registers bit assignments 8-26
Table 8-20 System Handler Control and State Register bit assignment 8-27
Table 8-21 Memory Manage Fault Status Register bit assignments 8-30
Table 8-22 Bus Fault Status Register bit assignments 8-31
Table 8-23 Usage Fault Status Register bit assignments 8-33
Table 8-24 Hard Fault Status Register bit assignments 8-34
Table 8-25 Debug Fault Status Register bit assignments 8-36
Table 8-26 Bit functions of the Memory Manage Fault Address Register 8-37
Table 8-27 Bit functions of the Bus Fault Address Register 8-37
Table 8-28 Software Trigger Interrupt Register bit assignments 8-38
Table 9-1 MPU registers 9-3

Table 9-2 MPU Type Register bit assignments 9-4
Table 9-3 MPU Control Register bit assignments 9-6
Table 9-4 MPU Region Number Register bit assignments 9-7
Table 9-5 MPU Region Base Address Register bit assignments 9-8
Table 9-6 MPU Region Attribute and Size Register bit assignments 9-9
Table 9-7 MPU protection region size field 9-11
Table 9-8 TEX, C, B encoding 9-14
Table 9-9 Cache policy for memory attribute encoding 9-15
Table 9-10 AP encoding 9-15
Table 9-11 XN encoding 9-15
Table 10-1 Core debug registers 10-2
Table 10-2 Debug Halting Control and Status Register 10-4
Table 10-3 Debug Core Selector Register 10-7
Table 10-4 Debug Exception and Monitor Control Register 10-9
Table 10-5 Application registers for use in core debug 10-13
Table 11-1 Flash patch register summary 11-7
Table 11-2 Flash Patch Control Register bit assignments 11-8
Table 11-3 COMP mapping 11-10
Table 11-4 Flash Patch Remap Register bit assignments 11-11
Table 11-5 Flash Patch Comparator Registers bit assignments 11-12
Table 11-6 DWT register summary 11-13
Table 11-7 DWT Control Register bit assignments 11-16
Table 11-8 DWT Current PC Sampler Cycle Count Register bit assignments 11-19
Table 11-9 DWT CPI Count Register bit assignments 11-20
Table 11-10 DWT Exception Overhead Count Register bit assignments 11-21
Table 11-11 DWT Sleep Count Register bit assignments 11-21
Table 11-12 DWT LSU Count Register bit assignments 11-22
Table 11-13 DWT Fold Count Register bit assignments 11-23
Table 11-14 DWT Comparator Registers 0-3 bit assignments 11-23
Table 11-15 DWT Mask Registers 0-3 bit assignments 11-24
Table 11-16 Bit functions of DWT Function Registers 0-3 11-25
Table 11-17 Settings for DWT Function Registers 11-26
Table 11-18 ITM register summary 11-28
Table 11-19 Bit functions of the ITM Trace Enable Register 11-30
Table 11-20 Bit functions of the ITM Trace Privilege Register 11-31
Table 11-21 Bit functions of the ITM Control Register 11-32
Table 11-22 Bit functions of the ITM Integration Write Register 11-34
Table 11-23 Bit functions of the ITM Integration Read Register 11-34
Table 11-24 Bit functions of the ITM Integration Mode Control Register 11-35
Table 11-25 Bit functions of the ITM Lock Access Register 11-35
Table 11-26 Bit functions of the ITM Lock Status Register 11-36
Table 11-27 AHB-AP register summary 11-37
Table 11-28 Bit functions of the AHB-AP Control and Status Word Register 11-39
Table 11-29 AHB-AP Transfer Address Register bit functions 11-40
Table 11-30 Bit functions of the AHB-AP Data Read/Write Register 11-41
Table 11-31 Bit functions of the AHB-AP Banked Data Register 11-41
Table 11-32 Bit functions of the AHB-AP Debug ROM Address Register 11-42

Table 11-33 Bit functions of the AHB-AP ID Register 11-42
Table 12-1 JTAG-DP signal connections 12-3
Table 12-2 Standard IR instructions 12-8
Table 12-3 Recommended implementation-defined IR instructions for IEEE 1149.1-compliance 12-9
Table 12-4 DPACC and APACC ACK responses 12-12
Table 12-5 JTAG target response summary 12-17
Table 12-6 Summary of JTAG host responses 12-18
Table 12-7 Target response summary for DP read transaction requests 12-33
Table 12-8 Target response summary for AP read transaction requests 12-34
Table 12-9 Target response summary for DP write transaction requests 12-35
Table 12-10 Target response summary for AP write transaction requests 12-36
Table 12-11 Summary of host (debugger) responses to the SW-DP acknowledge 12-37
Table 12-12 Terms used in SW-DP timing 12-38
Table 12-13 JTAG-DP register map 12-47
Table 12-14 SW-DP register map 12-49
Table 12-15 Abort Register bit assignments 12-50
Table 12-16 Identification Code Register bit assignments 12-52
Table 12-17 JEDEC JEP-106 manufacturer ID code, with ARM Limited values 12-53
Table 12-18 Control/Status Register bit assignments 12-54
Table 12-19 Control of pushed operation comparisons by MASKLANE 12-56
Table 12-20 Transfer Mode, TRNMODE, bit definitions 12-57
Table 12-21 Bit assignments for the AP Select Register, SELECT 12-58
Table 12-22 CTRLSEL field bit definitions 12-59
Table 12-23 Bit assignments for the Wire Control Register (SW-DP only) 12-61
Table 12-24 Turnaround tri-state period field, TURNROUND, bit definitions 12-61
Table 12-25 Wire operating mode, WIREMODE, bit definitions 12-62
Table 13-1 Trace Out Port signals 13-5
Table 13-2 ATB Port signals 13-6
Table 13-3 Miscellaneous configuration inputs 13-7
Table 13-4 TPIU registers 13-8
Table 13-5 Current Output Speed Divisors Register bit assignments 13-9
Table 13-6 Selected Pin Protocol Register bit assignments 13-10
Table 13-7 Formatter and Flush Status Register bit assignments 13-11
Table 13-8 Integration Test Register bit assignments 13-13
Table 13-9 Integration Test Register bit assignments 13-13
Table 14-1 Instruction fetches 14-3
Table 14-2 Bus mapper unaligned accesses 14-9
Table 14-3 Memory attributes 14-13
Table 15-1 Cortex-M3 resources 15-4
Table 15-2 Exception tracing mapping 15-11
Table 15-3 ETM registers 15-14
Table 16-1 ETM interface ports 16-3
Table 17-1 Instruction timings 17-3
Table A-1 Clock signals A-2
Table A-2 Reset signals A-3
Table A-3 Miscellaneous signals A-4

Table A-4 Interrupt interface A-5
Table A-5 ICode interface A-6
Table A-6 DCode interface A-8
Table A-7 System bus interface A-9
Table A-8 Private Peripheral Bus interface A-10
Table A-9 ITM interface A-11
Table A-10 AHB-AP interface A-12
Table A-11 ETM interface A-13
Table A-12 Test interface A-15

List of Figures
Cortex-M3 Technical Reference Manual

	Key to timing diagram conventions	xxi
Figure 1-1	Cortex-M3 block diagram	1-5
Figure 2-1	Cortex-M3 register set	2-4
Figure 2-2	Application Program Status Register bit assignments	2-5
Figure 2-3	Interrupt Program Status Register bit assignments	2-6
Figure 2-4	Execution Program Status Register	2-8
Figure 2-5	Little-endian and big-endian memory formats	2-12
Figure 4-1	The Cortex-M3 Memory Map	4-2
Figure 4-2	Bit-band mapping	4-6
Figure 5-1	Stack contents after a pre-emption	5-10
Figure 5-2	Exception entry timing	5-12
Figure 5-3	Tail-chaining timing	5-13
Figure 5-4	Late-arriving exception timing	5-14
Figure 5-5	Exception exit timing	5-17
Figure 5-6	Interrupt handling flowchart	5-33
Figure 5-7	Pre-emption flowchart	5-34
Figure 5-8	Return from interrupt flowchart	5-35
Figure 6-1	Reset signals	6-6
Figure 6-2	Power-on reset	6-6
Figure 6-3	Internal reset synchronization	6-7
Figure 7-1	SLEEPING power control example	7-4
Figure 7-2	SLEEPDEEP power control example	7-5
Figure 8-1	Interrupt Controller Type Register bit assignments	8-7

Figure 8-2 SysTick Control and Status Register bit assignments 8-8
Figure 8-3 SysTick Reload Value Register bit assignments 8-9
Figure 8-4 SysTick Current Value Register bit assignments 8-10
Figure 8-5 SysTick Calibration Value Register bit assignments 8-10
Figure 8-6 Interrupt Priority Registers 0-31 bit assignments 8-15
Figure 8-7 CPUID Base Register bit assignments 8-16
Figure 8-8 Interrupt Control State Register bit assignments 8-18
Figure 8-9 Vector Table Offset Register bit assignments 8-20
Figure 8-10 Application Interrupt and Reset Control Register bit assignments 8-21
Figure 8-11 System Control Register bit assignments 8-23
Figure 8-12 Configuration Control Register bit assignments 8-24
Figure 8-13 System Handler Priority Registers bit assignments 8-26
Figure 8-14 System Handler Control and State Register bit assignments 8-27
Figure 8-15 Local fault status registers bit assignments 8-29
Figure 8-16 Memory Manage Fault Register bit assignments 8-30
Figure 8-17 Bus Fault Status Register bit assignments 8-31
Figure 8-18 Usage Fault Status Register bit assignments 8-33
Figure 8-19 Hard Fault Status Register bit assignments 8-34
Figure 8-20 Debug Fault Status Register bit assignments 8-35
Figure 8-21 Software Trigger Interrupt Register bit assignments 8-38
Figure 9-1 MPU Type Register bit assignments 9-4
Figure 9-2 MPU Control Register bit assignments 9-5
Figure 9-3 MPU Region Number Register bit assignments 9-7
Figure 9-4 MPU Region Base Address Register bit assignments 9-8
Figure 9-5 MPU Region Attribute and Size Register bit assignments 9-9
Figure 10-1 Debug Halting Control and Status Register format 10-4
Figure 10-2 Debug Core Selector Register format 10-6
Figure 10-3 Debug Exception and Monitor Control Register format 10-9
Figure 11-1 System debug access block diagram 11-4
Figure 11-2 Flash Patch Control Register bit assignments 11-8
Figure 11-3 Flash Patch Remap Register bit assignments 11-10
Figure 11-4 Flash Patch Comparator Registers bit assignments 11-11
Figure 11-5 DWT Control Register bit assignments 11-15
Figure 11-6 DWT CPI Count Register bit assignments 11-20
Figure 11-7 DWT Exception Overhead Count Register bit assignments 11-20
Figure 11-8 DWT Sleep Count Register bit assignments 11-21
Figure 11-9 DWT LSU Count Register bit assignments 11-22
Figure 11-10 DWT Fold Count Register bit assignments 11-23
Figure 11-11 DWT Mask Registers 0-3 bit assignments 11-24
Figure 11-12 DWT Function Registers 0-3 bit assignments 11-25
Figure 11-13 ITM Trace Privilege Register bit assignments 11-31
Figure 11-14 ITM Control Register bit assignments 11-32
Figure 11-15 ITM Integration Write Register bit assignments 11-33
Figure 11-16 ITM Integration Read Register bit assignments 11-34
Figure 11-17 ITM Integration Mode Control bit assignments 11-35
Figure 11-18 ITM Lock Status Register bit assignments 11-36
Figure 11-19 AHB-AP Control and Status Word Register 11-38

Figure 11-20 AHB-AP ID Register 11-42
Figure 12-1 JTAG-DP physical connection 12-4
Figure 12-2 The DAP State Machine (JTAG) 12-5
Figure 12-3 JTAG Instruction Register bit order 12-7
Figure 12-4 JTAG Bypass Register operation 12-10
Figure 12-5 JTAG Device ID Code Register bit order 12-11
Figure 12-6 Bit order of JTAG DP and AP Access Registers 12-13
Figure 12-7 JTAG-DP ABORT scan chain bit order 12-19
Figure 12-8 Serial Wire Debug successful write operation 12-25
Figure 12-9 Serial Wire Debug successful read operation 12-25
Figure 12-10 Serial Wire Debug WAIT response to a packet request 12-26
Figure 12-11 Serial Wire Debug FAULT response to a packet request 12-26
Figure 12-12 Serial Wire Debug protocol error after a packet request 12-27
Figure 12-13 Serial Wire WAIT or FAULT response to a read operation when overrun detection is enabled 12-31
Figure 12-14 Serial Wire WAIT or FAULT response to a write operation when overrun detection is enabled 12-31
Figure 12-15 SW-DP acknowledgement timing 12-38
Figure 12-16 SW-DP to DAP bus timing for writes 12-39
Figure 12-17 SW-DP to DAP bus timing for reads 12-39
Figure 12-18 SW-DP idle timing 12-40
Figure 12-19 Pushed operations overview 12-44
Figure 12-20 Abort Register bit assignments 12-50
Figure 12-21 Identification Code Register bit assignments 12-52
Figure 12-22 Control/Status Register bit assignments 12-54
Figure 12-23 Bit assignments for the AP Select Register, SELECT 12-58
Figure 12-24 Bit assignments for the Wire Control Register (SW-DP only) 12-60
Figure 13-1 Block diagram of the TPIU (non-ETM version) 13-3
Figure 13-2 Block diagram of the TPIU (ETM version) 13-4
Figure 13-3 Supported Port Size Register bit assignments 13-9
Figure 13-4 Current Output Speed Divisors Register bit assignments 13-9
Figure 13-5 Selected Pin Protocol Register bit assignments 13-10
Figure 13-6 Formatter and Flush Status Register bit assignments 13-11
Figure 13-7 Integration Test Register bit assignments 13-12
Figure 13-8 Integration Test Register bit assignments 13-13
Figure 15-1 ETM block diagram 15-3
Figure 15-2 Exception return packet encoding 15-10
Figure 15-3 Exception encoding for branch packet 15-13
Figure 16-1 Conditional branch backwards not taken 16-5
Figure 16-2 Conditional branch backwards taken 16-5
Figure 16-3 Conditional branch forwards not taken 16-6
Figure 16-4 Conditional branch forwards taken 16-6
Figure 16-5 Unconditional branch without pipeline stalls 16-6
Figure 16-6 Unconditional branch with pipeline stalls 16-7
Figure 16-7 Unconditional branch in execute aligned 16-7
Figure 16-8 Unconditional branch in execute unaligned 16-7

Preface

This preface introduces the *Cortex-M3 r0p0 Technical Reference Manual*. It contains the following sections:

- *About this manual* on page xviii
- *Feedback* on page xxiii.

About this manual

This is the *Technical Reference Manual* (TRM) for the Cortex-M3 processor.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this manual, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This manual is written to help system designers, system integrators, and verification engineers who are implementing a *System-on-a-Chip* (SoC) device based on the Cortex-M3 processor.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter to learn about the components of the Cortex-M3 processor, and about the processor instruction set.

Chapter 2 *Programmer's Model*

Read this chapter to learn about the Cortex-M3 register set, modes of operation, and other information for programming the Cortex-M3 processor.

Chapter 3 *System Control*

Read this chapter to learn about the registers and programmer's model for system control.

Chapter 4 *Memory Map*

Read this chapter to learn about the processor memory map and bit-banding feature.

Chapter 5 *Exceptions*

Read this chapter to learn about the processor exception model.

Chapter 6 *Clocking and Resets*

Read this chapter to learn about the processor clocking and resets.

Chapter 7 ***Power Management***

Read this chapter to learn about the processor power management and power saving.

Chapter 8 ***Nested Vectored Interrupt Controller***

Read this chapter to learn about the processor interrupt processing and control.

Chapter 9 ***Memory Protection Unit***

Read this chapter to learn about the processor Memory Protection Unit.

Chapter 10 ***Core Debug***

Read this chapter to learn about debugging and testing the processor processor core.

Chapter 11 ***System Debug***

Read this chapter to learn about the processor system debug components.

Chapter 12 ***Debug Port***

Read this chapter to learn about the processor debug port, and the JTAG Debug Port and Serial Wire Debug Port.

Chapter 13 ***Trace Port Interface Unit***

Read this chapter to learn about the processor *Trace Port Interface Unit* (TPIU).

Chapter 14 ***Bus Interface***

Read this chapter to learn about the processor Bus Interfaces.

Chapter 15 ***Embedded Trace Macrocell***

Read this chapter to learn about the processor *Embedded Trace Macrocell* (ETM).

Chapter 16 ***Embedded Trace Macrocell Interface***

Read this chapter to learn about the processor ETM interface.

Chapter 17 ***Instruction Timing***

Read this chapter to learn about the processor instruction timing and clock cycles.

Appendix A ***Signal Descriptions***

Read this appendix for a summary of Cortex-M3 signals.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams*
- *Signals* on page xxi
- *Numbering* on page xxi.

Typographical

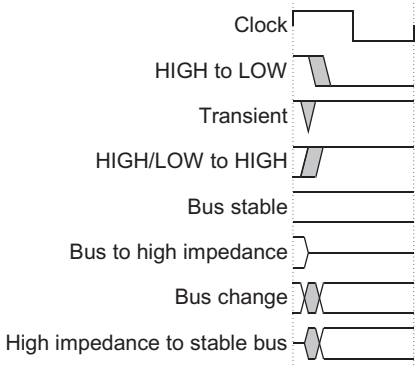
The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
< 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.

Timing diagrams

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

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



Key to timing diagram conventions

Signals

The signal conventions are:

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

Lower-case n Denotes an active-LOW signal.

Prefix H Denotes *Advanced High-performance Bus* (AHB) signals.

Prefix P Denotes *Advanced Peripheral Bus* (APB) signals.

Numbering

The numbering convention is:

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.

- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

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

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

ARM publications

This manual contains information that is specific to the Cortex-M3 processor. See the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM Architecture Reference Manual, Thumb-2 Supplement* (ARM DDI 0308)
- *ARMv7-M Architecture Reference Manual* (PRD03-GENC-006471)
- *ARM Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014)
- *ARM AMBA® Specification (Rev 2.0)* (ARM IHI 0011)

Other publications

This section lists relevant documents published by third parties:

- IEEE Standard, *Test Access Port and Boundary-Scan Architecture specification* 1149.1-1990 (JTAG).

Feedback

ARM Limited welcomes feedback both on the Cortex-M3 processor, and on the documentation.

Feedback on the Cortex-M3 processor

If you have any comments or suggestions about this product, please contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this manual

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

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

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the processor and instruction set. It contains the following sections:

- *About the processor* on page 1-2
- *Components of the processor* on page 1-4
- *Configurable options* on page 1-12
- *Instruction set summary* on page 1-13.

1.1 About the processor

The processor is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. It is intended for deeply embedded applications that require fast interrupt response features. The processor implements the ARM architecture v7-M.

The processor incorporates:

- Processor core. A low gate count processor, with low latency interrupt processing that features:
  - ARMv7-M: A Thumb-2 ISA subset, consisting of all base Thumb-2 instructions, 16-bit and 32-bit, and excluding blocks for media, SIMD, E (DSP), and ARM system access.
  - Banked SP only.
  - Hardware divide instructions, SDIV and UDIV (Thumb-2 instructions).
  - Handler and Thread modes.
  - Thumb and Debug states.
  - Interruptible-continued LDM/STM, PUSH/POP for low interrupt latency.
  - Automatic processor state saving and restoration for low latency *Interrupt Service Routine* (ISR) entry and exit.
  - ARM architecture v6 style BE8/LE support.
  - ARMv6 unaligned accesses.
- *Nested Vectored Interrupt Controller* (NVIC) closely integrated with the processor core to achieve low latency interrupt processing. Features include:
  - Configurable number, 1 to 240, of external interrupts.
  - Configurable number, 3 to 8, of bits of priority.
  - Dynamic reprioritization of interrupts.
  - Priority grouping. This allows selection of pre-empting interrupt levels and non pre-empting interrupt levels.
  - Support for tail-chaining, and late arrival, of interrupts. This enables back-to-back interrupt processing without the overhead of state saving and restoration between interrupts.
  - Processor state automatically saved on interrupt entry, and restored on interrupt exit, with no instruction overhead.

- *Memory Protection Unit* (MPU). An optional MPU for memory protection.
  - Eight memory regions.
  - *Sub Region Disable* (SRD), enabling efficient use of memory regions.
  - Background region can be enabled which implements the default memory map attributes.
- Bus interfaces:
  - AHBLite ICode, DCode and System bus interfaces.
  - APB *Private Peripheral Bus* (PPB) Interface
  - Bit band support. Atomic bit-band write and read operations.
  - Memory access alignment.
  - Write buffer. For buffering of write data.
- Low-cost debug solution that features:
  - Debug access to all memory and registers in the system, including Cortex-M3 register bank when the core is running, halted, or held in reset.
  - *Serial Wire* (SW-DP) or *JTAG* (JTAG-DP) debug access, or both.
  - *Flash Patch and Breakpoint* unit (FPB) for implementing breakpoints and code patches.
  - *Data Watchpoint and Trigger* unit (DWT) for implementing watchpoints, trigger resources, and system profiling.
  - *Instrumentation Trace Macrocell* (ITM) for support of printf style debugging.
  - *Trace Port Interface Unit* (TPIU) for bridging to a Trace Port Analyzer.
  - Optional *Embedded Trace Macrocell* (ETM) for instruction trace.

1.2 Components of the processor

This section describes the components of the processor. The main blocks of the Cortex-M3 system are:

- *Processor core* on page 1-6
- *NVIC* on page 1-8
- *Bus Matrix* on page 1-8
- *FPB* on page 1-9
- *DWT* on page 1-9
- *ITM* on page 1-10
- *MPU* on page 1-10
- *ETM* on page 1-10
- *TPIU* on page 1-10.

Figure 1-1 on page 1-5 shows the structure of the Cortex-M3.

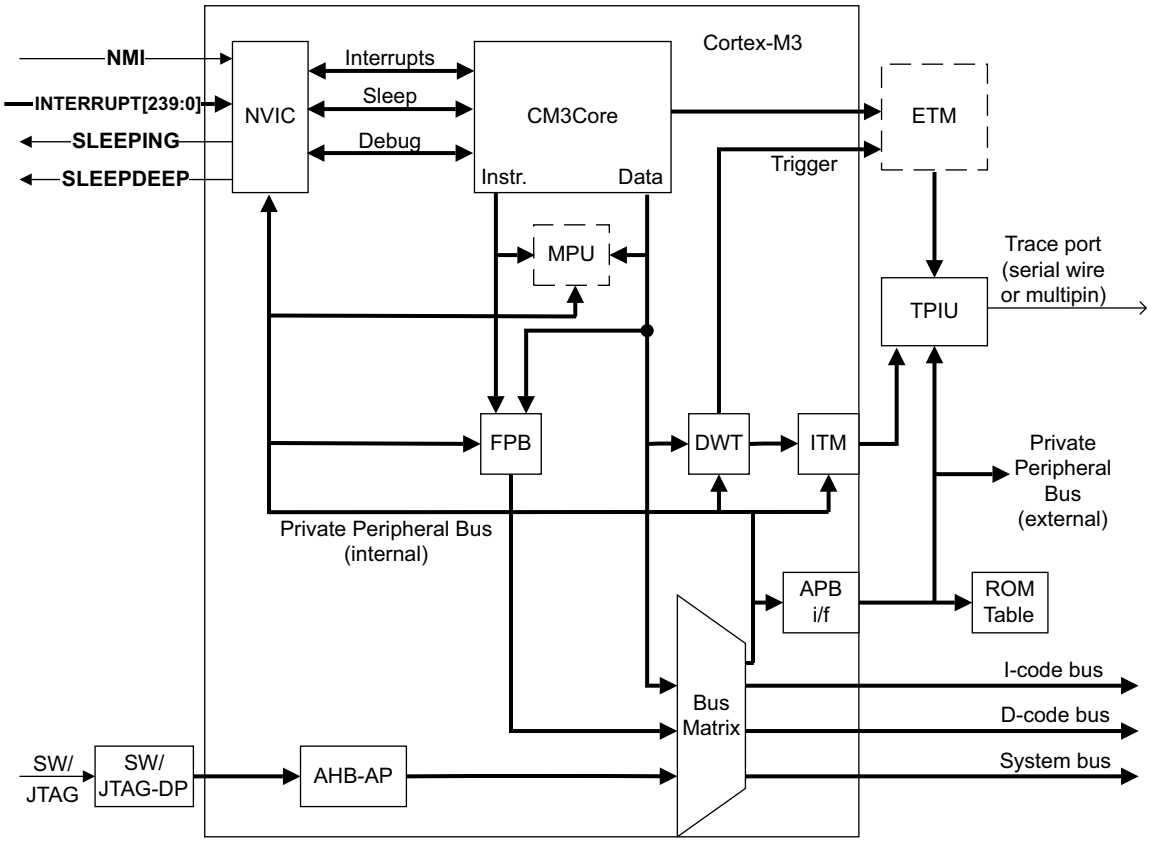


Figure 1-1 Cortex-M3 block diagram

Note

The ETM and the MPU are optional components and might not exist in your implementation.

1.2.1 Cortex-M3 hierarchy and implementation options

The processor components exist in two levels of hierarchy, as shown in Figure 1-1. This represents the RTL hierarchy of the design. Four components, ETM, TPIU, SW/JTAG-DP, and ROM table, are shown outside the Cortex-M3 level because these components are either optional, or there is flexibility in how they are implemented and used. Your implementation might differ from that shown in Figure 1-1. The possible implementation options are shown in the next three subsections.

TPIU

The implementation options for the TPIU are:

- If the ETM is present in your system, then the TPIU formatter is included. Otherwise the formatter is not included.
- A multi-core implementation can be traced by either single or multiple TPIUs.
- The ARM TPIU block might have been replaced by a partner-specific CoreSight compliant TPIU.
- In a production device, the TPIU might have been removed.

  Note

  There is no Cortex-M3 trace capability if the TPIU has been removed.

SW/JTAG-DP

The implementation options for the SW/JTAG-DP are:

- Your implementation might contain either or both SW-DP and JTAG-DP.
- The ARM SW-DP might have been replaced by a partner specific CoreSight compliant SW-DP.
- The ARM JTAG-DP might have been replaced by a partner specific CoreSight compliant JTAG-DP.
- A partner specific test interface might have been included in parallel with SW-DP or JTAG-DP.

ROM table

The ROM table is modified from that described in *ROM memory table* on page 4-8 if:

- Additional debug components have been added into the system.

1.2.2 Processor core

The processor core implements the ARMv7-M architecture. It has the following main features:

- Thumb-2 *Instruction Set Architecture* (ISA) subset consisting of all base Thumb-2 instructions, 16-bit and 32-bit.

- Harvard processor architecture enabling simultaneous instruction fetch with data load/store.
- Three-stage pipeline.
- Single cycle 32-bit multiply.
- Hardware divide.
- Thumb and Debug states.
- Handler and Thread modes.
- Low latency ISR entry and exit.
  - Processor state saving and restoration, with no instruction fetch overhead. Exception vector is fetched from memory in parallel with the state saving, enabling faster ISR entry.
  - Tightly coupled interface to interrupt controller enabling efficient processing of late-arriving interrupts.
  - Tail-chaining of interrupts, enabling back-to-back interrupt processing without the overhead of state saving and restoration between interrupts.
  - Support for late arriving interrupts.
- Interruptible-continued LDM/STM, PUSH/POP.
- ARMv6 style BE8/LE support.
- ARMv6 unaligned.

The processor core is described further in the following sections.

Registers

The processor contains:

- 13 general purpose 32-bit registers
- *Link Register* (LR)
- *Program Counter* (PC)
- *Program Status Register*, xPSR
- two banked SP registers.

Memory interface

The processor has a Harvard interface to enable simultaneous instruction fetches with data/load stores. Memory accesses are controlled by:

- A separate *Load Store Unit* (LSU) that decouples load and store operations from the ALU.
- A three-word entry prefetch unit. One word is fetched at a time. This can be two Thumb instructions, one word-aligned Thumb-2 instruction or the upper/lower halfword of a halfword aligned Thumb-2 instruction. All fetch addresses from the core are word aligned. If a Thumb-2 instruction is halfword aligned, two fetches are necessary to fetch the Thumb-2 instruction. However, the 3-entry prefetch buffer ensures that a stall cycle is only necessary for the first halfword Thumb-2 instruction fetched.

1.2.3 NVIC

The NVIC is tightly coupled to the processor core. This facilitates low latency exception processing. The main features include:

- a configurable number of external interrupts, from 1 to 240
- a configurable number of bits of priority, from three to eight bits
- level and pulse interrupt support
- dynamic reprioritization of interrupts
- priority grouping
- support for tail-chaining of interrupts
- processor state automatically saved on interrupt entry, and restored on interrupt exit, with no instruction overhead.

Chapter 8 *Nested Vectored Interrupt Controller* describes the NVIC in detail.

1.2.4 Bus Matrix

The bus matrix connects the processor and debug interface to the external buses. The bus matrix interfaces to the following external buses:

- ICode bus. This is for instruction and vector fetches from code space. This is a 32-bit AHBLite bus.
- DCode bus. This is for data load/stores and debug accesses to code space. This is a 32-bit AHBLite bus.
- System bus. This is for instruction and vector fetches, data load/stores and debug accesses to system space. This is a 32-bit AHBLite bus.

- PPB. This is for data load/stores and debug accesses to PPB space. This is a 32-bit APB (v2.0) bus.

The bus matrix also controls the following:

- Unaligned accesses. The bus matrix converts unaligned processor accesses into aligned accesses.
- Bit-banding. The bus matrix converts bit-band alias accesses into bit-band region accesses. It performs:
  - bit field extract for bit-band loads
  - atomic read-modify-write for bit-band stores.
- Write buffering. The bus matrix contains a one-entry write buffer to decouple bus stalls from the processor core.

Chapter 14 *Bus Interface* describes the bus interfaces in detail.

1.2.5 FPB

The FPB unit implements hardware breakpoints and patches accesses from code space to system space. The FPB has eight comparators as follows:

- Six instruction comparators that can be individually configured to either remap instruction fetches from code space to system space, or perform a hardware breakpoint.
- Two literal comparators that can remap literal accesses from code space to system space.

Chapter 11 *System Debug* describes the FPB in detail.

1.2.6 DWT

The DWT unit incorporates the following debug functionality:

- It contains four comparators each of which can be configured either as a hardware watchpoint, an ETM trigger, a PC sampler event trigger, or a data address sampler event trigger.
- It contains several counters for performance profiling.
- It can be configured to emit PC samples at defined intervals, and to emit interrupt event information.

Chapter 11 *System Debug* describes the DWT in detail.

1.2.7 ITM

The ITM is a an application driven trace source that supports application event trace and *printf* style debugging.

The ITM provides the following sources of trace information:

- Software trace. Software can write directly to ITM stimulus registers. This causes packets to be emitted.
- Hardware trace. These packets are generated by the DWT, and emitted by the ITM.
- Time stamping. Timestamps are emitted relative to packets.

Chapter 11 *System Debug* describes the ITM in detail.

1.2.8 MPU

An optional MPU is available for the processor to provide memory protection. The MPU checks access permissions and memory attributes. It contains eight regions, and an optional background region that implements the default memory map attributes.

Chapter 9 *Memory Protection Unit* describes the MPU.

1.2.9 ETM

The ETM is a low-cost trace macrocell that supports instruction trace only.

Chapter 15 *Embedded Trace Macrocell* describes the ETM in detail.

1.2.10 TPIU

The TPIU acts as a bridge between the Cortex-M3 trace data from the ITM, and ETM if present, and an off-chip Trace Port Analyzer. The TPIU can be configured to support either serial pin trace for low cost debug, or multi pin trace for higher bandwidth trace. The TPIU is CoreSight compatible.

Chapter 13 *Trace Port Interface Unit* describes the TPIU in detail.

1.2.11 SW/JTAG-DP

The processor can be configured to have SW-DP or JTAG-DP debug port interfaces, or both. The debug port provides debug access to all registers and memory in the system, including the processor registers.

Chapter 12 *Debug Port* describes the SW/JTAG-DP in detail.

1.3 Configurable options

This section shows the configuration options for the processor. Contact your implementor to confirm the configuration of your implementation.

1.3.1 Interrupts

The number of external interrupts can be configured at implementation from 1 to 240. The number of bits of interrupt priority can be configured at implementation from three to eight bits.

1.3.2 MPU

The Cortex-M3 system can be configured at implementation to include an MPU.

Chapter 9 *Memory Protection Unit* describes the MPU.

1.3.3 ETM

The Cortex-M3 system can be configured at implementation to include an ETM.

Chapter 16 *Embedded Trace Macrocell Interface* describes the ETM.

1.4 Instruction set summary

This section provides:

- a summary of the processor 16-bit instructions
- a summary of the processor 32-bit instructions.

Table 1-1 lists the 16-bit Cortex-M3 instructions.

Table 1-1 16-bit Cortex-M3 instruction summary

Operation	Assembler
Add register value and C flag to register value	ADC <Rd>, <Rm>
Add immediate 3-bit value to register	ADD <Rd>, <Rn>, #<immed_3>
Add immediate 8-bit value to register	ADD <Rd>, #<immed_8>
Add low register value to low register value	ADD <Rd>, <Rn>, <Rm>
Add high register value to low or high register value	ADD <Rd>, <Rm>
Add 4 (immediate 8-bit value) to PC	ADD <Rd>, PC, #<immed_8> * 4
Add 4 (immediate 8-bit value) to SP	ADD <Rd>, SP, #<immed_8> * 4
Add 4 (immediate 7-bit value) to SP	ADD <Rd>, SP, #<immed_7> * 4 or ADD SP, SP, #<immed_7> * 4
Bitwise AND register values	AND <Rd>, <Rm>
Arithmetic shift right by immediate number	ASR <Rd>, <Rm>, #<immed_5>
Arithmetic shift right by number in register	ASR <Rd>, <Rs>
Branch conditional	B<cond> <target address>
Branch unconditional	B <target_address>
Bit clear	BIC <Rd>, <Rm>
Software breakpoint	BKPT <immed_8>
Branch with link	BL <Rm>
Compare not zero and branch	CBNZ <Rn>,<label>
Compare zero and branch	CBZ <Rn>,<label>
Compare negation of register value with another register value	CMN <Rn>, <Rm>
Compare immediate 8-bit value	CMP <Rn>, #<immed_8>

Table 1-1 16-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Compare registers	`CMP <Rn>, <Rm>`
Compare high register to low or high register	`CMP <Rn>, <Rm>`
Change processor state	`CPS <effect>, <iflags>`
Copy high or low register value to another high or low register	`CPY <Rd> <Rm>`
Bitwise exclusive OR register values	`EOR <Rd>, <Rm>`
Condition the following instruction, Condition the following two instructions, Condition the following three instructions, Condition the following four instructions	`IT <cond> IT<x> <cond> IT<x><y> <cond> IT<x><y><z> <cond>`
Multiple sequential memory words	`LDMIA <Rn>!, <registers>`
Load memory word from base register address + 5-bit immediate offset	`LDR <Rd>, [<Rn>, #<immed_5> * 4]`
Load memory word from base register address + register offset	`LDR <Rd>, [<Rn>, <Rm>]`
Load memory word from PC address + 8-bit immediate offset	`LDR <Rd>, [PC, #<immed_8> * 4]`
Load memory word from SP address + 8-bit immediate offset	`LDR, <Rd>, [SP, #<immed_8> * 4]`
Load memory byte [7:0] from register address + 5-bit immediate offset	`LDRB <Rd>, [<Rn>, #<immed_5>]`
Load memory byte [7:0] from register address + register offset	`LDRB <Rd>, [<Rn>, <Rm>]`
Load memory halfword [15:0] from register address + 5-bit immediate offset	`LDRH <Rd>, [<Rn>, #<immed_5> * 2]`
Load halfword [15:0] from register address + register offset	`LDRH <Rd>, [<Rn>, <Rm>]`
Load signed byte [7:0] from register address + register offset	`LDRSB <Rd>, [<Rn>, <Rm>]`
Load signed halfword [15:0] from register address + register offset	`LDRSH <Rd>, [<Rn>, <Rm>]`
Logical shift left by immediate number	`LSL <Rd>, <Rm>, #<immed_5>`
Logical shift left by number in register	`LSL <Rd>, <Rs>`
Logical shift right by immediate number	`LSR <Rd>, <Rm>, #<immed_5>`
Logical shift right by number in register	`LSR <Rd>, <Rs>`
Move immediate 8-bit value to register	`MOV <Rd>, #<immed_8>`
Move low register value to low register	`MOV <Rd>, <Rn>`
Move high or low register value to high or low register	`MOV <Rd>, <Rm>`

Table 1-1 16-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Multiply register values	`MUL <Rd>, <Rm>`
Move complement of register value to register	`MVN <Rd>, <Rm>`
Negate register value and store in register	`NEG <Rd>, <Rm>`
No operation	`NOP <c>`
Bitwise logical OR register values	`ORR <Rd>, <Rm>`
Pop registers from stack	`POP <registers>`
Pop registers and PC from stack	`POP <registers, PC>`
Push registers onto stack	`PUSH <registers>`
Push LR and registers onto stack	`PUSH <registers, LR>`
Reverse bytes in word and copy to register	`REV <Rd>, <Rn>`
Reverse bytes in two halfwords and copy to register	`REV16 <Rd>, <Rn>`
Reverse bytes in low halfword [15:0], sign-extend, and copy to register	`REVSH <Rd>, <Rn>`
Rotate right by amount in register	`ROR <Rd>, <Rs>`
Subtract register value and C flag from register value	`SBC <Rd>, <Rm>`
Send event	`SEV <c>`
Store multiple register words to sequential memory locations	`STMIA <Rn>!, <registers>`
Store register word to register address + 5-bit immediate offset	`STR <Rd>, [<Rn>, #<immed_5> * 4]`
Store register word to register address	`STR <Rd>, [<Rn>, <Rm>]`
Store register word to SP address + 8-bit immediate offset	`STR <Rd>, [SP, #<immed_8> * 4]`
Store register byte [7:0] to register address + 5-bit immediate offset	`STRB <Rd>, [<Rn>, #<immed_5>]`
Store register byte [7:0] to register address	`STRB <Rd>, [<Rn>, <Rm>]`
Store register halfword [15:0] to register address + 5-bit immediate offset	`STRH <Rd>, [<Rn>, #<immed_5> * 2]`
Store register halfword [15:0] to register address + register offset	`STRH <Rd>, [<Rn>, <Rm>]`
Subtract immediate 3-bit value from register	`SUB <Rd>, <Rn>, #<immed_3>`
Subtract immediate 8-bit value from register value	`SUB <Rd>, #<immed_8>`

Table 1-1 16-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Subtract register values	`SUB <Rd>, <Rn>, <Rm>`
Subtract 4 (immediate 7-bit value) from SP	`SUB SP, #<immed_7> * 4`
Operating system service call with 8-bit immediate call code	`SVC <immed_8>`
Extract byte [7:0] from register, move to register, and sign-extend to 32 bits	`SXTB <Rd>, <Rm>`
Extract halfword [15:0] from register, move to register, and sign-extend to 32 bits	`SXTH <Rd>, <Rm>`
Test register value for set bits by ANDing it with another register value	`TST <Rn>, <Rm>`
Extract byte [7:0] from register, move to register, and zero-extend to 32 bits	`UXTB <Rd>, <Rm>`
Extract halfword [15:0] from register, move to register, and zero-extend to 32 bits	`UXTH <Rd>, <Rm>`
Wait for event	`WFE <c>`
Wait for interrupt	`WFI <c>`

Table 1-2 lists the 32-bit Cortex-M3 instructions.

Table 1-2 32-bit Cortex-M3 instruction summary

Operation	Assembler
Add register value, immediate 12-bit value, and C bit	`ADC{S}.W <Rd>, <Rn>, #<modify_constant(immed_12>`
Add register value, shifted register value, and C bit	`ADC{S}.W <Rd>, <Rn>, <Rm>{, <shift>}`
Add register value and immediate 12-bit value	`ADD{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>`
Add register value and shifted register value	`ADD{S}.W <Rd>, <Rm>{, <shift>}`
Add register value and immediate 12-bit value	`ADDW.W <Rd>, <Rn>, #<immed_12>`
Bitwise AND register value with immediate 12-bit value	`AND{S}.W <Rd>, <Rn>, #<modify_constant(immed_12>`
Bitwise AND register value with shifted register value	`AND{S}.W <Rd>, <Rn>, Rm>{, <shift>}`
Arithmetic shift right by number in register	`ASR{S}.W <Rd>, <Rn>, <Rm>`
Conditional branch	`B{cond}.W <label>`
Clear bit field	`BFC.W <Rd>, #<lsb>, #<width>`
Insert bit field from one register value into another	`BFI.W <Rd>, <Rn>, #<lsb>, #<width>`

Table 1-2 32-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Bitwise AND register value with complement of immediate 12-bit value	`BIC{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>`
Bitwise AND register value with complement of shifted register value	`BIC{S}.W <Rd>, <Rn>, <Rm>{, <shift>}`
Branch with link	`BL <label>`
Branch with link (immediate)	`BL<c> <label>`
Unconditional branch	`B.W <label>`
Return number of leading zeros in register value	`CLZ.W <Rd>, <Rn>`
Compare register value with two's complement of immediate 12-bit value	`CMN.W <Rn>, #<modify_constant(immed_12)>`
Compare register value with two's complement of shifted register value	`CMN.W <Rn>, <Rm>{, <shift>}`
Compare register value with immediate 12-bit value	`CMP.W <Rn>, #<modify_constant(immed_12)>`
Compare register value with shifted register value	`CMP.W <Rn>, <Rm>{, <shift>}`
Data memory barrier	`DMB <c>`
Data synchronization barrier	`DSB <c>`
Exclusive OR register value with immediate 12-bit value	`EOR{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>`
Exclusive OR register value with shifted register value	`EOR{S}.W <Rd>, <Rn>, <Rm>{, <shift>}`
Instruction synchronization barrier	`ISB <c>`
Load multiple memory registers, increment after or decrement before	`LDM{IA|DB}.W <Rn>{!}, <registers>`
Memory word from base register address + immediate 12-bit offset	`LDR.W <Rxf>, [<Rn>, #<offset_12>]`
Memory word to PC from register address + immediate 12-bit offset	`LDR.W PC, [<Rn>, #<offset_12>]`
Memory word to PC from base register address immediate 8-bit offset, postindexed	`LDR.W PC, #<+/-<offset_8>`
Memory word from base register address immediate 8-bit offset, postindexed	`LDR.W <Rxf>, [<Rn>], #+/-<offset_8>`

Table 1-2 32-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Memory word from base register address immediate 8-bit offset, preindexed	`LDR.W <Rxf>, [<Rn>, #<+/-<offset_8>]!`
Memory word to PC from base register address immediate 8-bit offset, preindexed	`LDR.W PC, [<Rn>, #+/-<offset_8>]!`
Memory word from register address shifted left by 0, 1, 2, or 3 places	`LDR.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Memory word to PC from register address shifted left by 0, 1, 2, or 3 places	`LDR.W PC, [<Rn>, <Rm>{, LSL #<shift>}]`
Memory word from PC address immediate 12-bit offset	`LDR.W <Rxf>, [PC, #+/-<offset_12>]`
Memory word to PC from PC address immediate 12-bit offset	`LDR.W PC, [PC, #+/-<offset_12>]`
Memory byte [7:0] from base register address + immediate 12-bit offset	`LDRB.W <Rxf>, [<Rn>, #<offset_12>]`
Memory byte [7:0] from base register address immediate 8-bit offset, postindexed	`LDRB.W <Rxf>. [<Rn>], #+/-<offset_8>`
Memory byte [7:0] from register address shifted left by 0, 1, 2, or 3 places	`LDRB.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Memory byte [7:0] from base register address immediate 8-bit offset, preindexed	`LDRB.W <Rxf>, [<Rn>, #<+/-<offset_8>]!`
Memory byte from PC address immediate 12-bit offset	`LDRB.W <Rxf>, [PC, #+/-<offset_12>]`
Memory doubleword from register address 8-bit offset 4, preindexed	`LDRD.W <Rxf>, <Rxf2>, [<Rn>, #+/-<offset_8> * 4]{!}`
Memory doubleword from register address 8-bit offset 4, postindexed	`LDRD.W <Rxf>, <Rxf2>, [<Rn>], #+/-<offset_8> * 4`
Memory halfword [15:0] from base register address + immediate 12-bit offset	`LDRH.W <Rxf>, [<Rn>, #<offset_12>]`
Memory halfword [15:0] from base register address immediate 8-bit offset, preindexed	`LDRH.W <Rxf>, [<Rn>, #<+/-<offset_8>]!`
Memory halfword [15:0] from base register address immediate 8-bit offset, postindexed	`LDRH.W <Rxf>. [<Rn>], #+/-<offset_8>`
Memory halfword [15:0] from register address shifted left by 0, 1, 2, or 3 places	`LDRH.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`

Table 1-2 32-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Memory halfword from PC address immediate 12-bit offset	`LDRH.W <Rxf>, [PC, #+/-<offset_12>]`
Memory signed byte [7:0] from base register address + immediate 12-bit offset	`LDRSB.W <Rxf>, [<Rn>, #<offset_12>]`
Memory signed byte [7:0] from base register address immediate 8-bit offset, postindexed	`LDRSB.W <Rxf>. [<Rn>], #+/-<offset_8>`
Memory signed byte [7:0] from base register address immediate 8-bit offset, preindexed	`LDRSB.W <Rxf>, [<Rn>, #<+/-<offset_8>]!`
Memory signed byte [7:0] from register address shifted left by 0, 1, 2, or 3 places	`LDRSB.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Memory signed byte from PC address immediate 12-bit offset	`LDRSB.W <Rxf>, [PC, #+/-<offset_12>]`
Memory signed halfword [15:0] from base register address + immediate 12-bit offset	`LDRSH.W <Rxf>, [<Rn>, #<offset_12>]`
Memory signed halfword [15:0] from base register address immediate 8-bit offset, postindexed	`LDRSH.W <Rxf>. [<Rn>], #+/-<offset_8>`
Memory signed halfword [15:0] from base register address immediate 8-bit offset, preindexed	`LDRSH.W <Rxf>, [<Rn>, #<+/-<offset_8>]!`
Memory signed halfword [15:0] from register address shifted left by 0, 1, 2, or 3 places	`LDRSH.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Memory signed halfword from PC address immediate 12-bit offset	`LDRSH.W <Rxf>, [PC, #+/-<offset_12>]`
Logical shift left register value by number in register	`LSL{S}.W <Rd>, <Rn>, <Rm>`
Logical shift right register value by number in register	`LSR{S}.W <Rd>, <Rn>, <Rm>`
Multiply two signed or unsigned register values and add the low 32 bits to a register value	`MLA.W <Rd>, <Rn>, <Rm>, <Racc>`
Multiply two signed or unsigned register values and subtract the low 32 bits from a register value	`MLS.W <Rd>, <Rn>, <Rm>, <Racc>`
Move immediate 12-bit value to register	`MOV{S}.W <Rd>, #<modify_constant(immed_12)>`
Move shifted register value to register	`MOV{S}.W <Rd>, <Rm>{, <shift>}`
Move immediate 16-bit value to top halfword [31:16] of register	`MOVT.W <Rd>, #<immed_16>`

Table 1-2 32-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Move immediate 16-bit value to bottom halfword [15:0] of register and clear top halfword [31:16]	MOVW.W <Rd>, #<immed_16>
Move to register from status	MRS<c> <Rd>, <psr>
Move to status register	MSR<c> <psr>_<fields>,<Rn>
Multiply two signed or unsigned register values	MUL.W <Rd>, <Rn>, <Rm>
No operation	NOP.W
Logical OR NOT register value with immediate 12-bit value	ORN{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Logical OR NOT register value with shifted register value	ORN[S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Logical OR register value with immediate 12-bit value	ORR{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Logical OR register value with shifted register value	ORR{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Reverse bit order	RBIT.W <Rd>, <Rm>
Reverse bytes in word	REV.W <Rd>, <Rm>
Reverse bytes in each halfword	REV16.W <Rd>, <Rn>
Reverse bytes in bottom halfword and sign-extend	REVSH.W <Rd>, <Rn>
Rotate right by number in register	ROR{S}.W <Rd>, <Rn>, <Rm>
Subtract a register value from an immediate 12-bit value	RSB{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Subtract a register value from a shifted register value	RSB{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Subtract immediate 12-bit value and C bit from register value	SBC{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>
Subtract shifted register value and C bit from register value	SBC{S}.W <Rd>, <Rn>, <Rm>{, <shift>}
Copy selected bits to register and sign-extend	SBFX.W <Rd>, <Rn>, #<lsb>, #<width>
Signed divide	SDIV<c> <Rd>,<Rn>,<Rm>
Send event	SEV<c>
Multiply signed halfwords and add signed-extended value to 2-register value	SMLAL.W <RdLo>, <RdHi>, <Rn>, <Rm>
Multiply two signed register values	SMULL.W <RdLo>, <RdHi>, <Rn>, <Rm>
Signed saturate	SSAT <c> <Rd>, #<imm>, <Rn>{, <shift>}

Table 1-2 32-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Multiple register words to consecutive memory locations	`STM{IA|DB}.W <Rn>{!}, <registers>`
Register word to register address + immediate 12-bit offset	`STR.W <Rxf>, [<Rn>, #<offset_12>]`
Register word to register address immediate 8-bit offset, postindexed	`STR.W <Rxf>, [<Rn>], #+/-<offset_8>`
Register word to register address shifted by 0, 1, 2, or 3 places	`STR.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Register word to register address immediate 8-bit offset, preindexed Store, preindexed	`STR{T}.W <Rxf>, [<Rn>, #+/-<offset_8>]{!}`
Register byte [7:0] to register address immediate 8-bit offset, preindexed	`STRB{T}.W <Rxf>, [<Rn>, #+/-<offset_8>]{!}`
Register byte [7:0] to register address + immediate 12-bit offset	`STRB.W <Rxf>, [<Rn>, #<offset_12>]`
Register byte [7:0] to register address immediate 8-bit offset, postindexed	`STRB.W <Rxf>, [<Rn>], #+/-<offset_8>`
Register byte [7:0] to register address shifted by 0, 1, 2, or 3 places	`STRB.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Store doubleword, preindexed	`STRD.W <Rxf>, <Rxf2>, [<Rn>, #+/-<offset_8> * 4]{!}`
Store doubleword, postindexed	`STRD.W <Rxf>, <Rxf2>, [<Rn>], #+/-<offset_8> * 4`
Register halfword [15:0] to register address + immediate 12-bit offset	`STRH.W <Rxf>, [<Rn>, #<offset_12>]`
Register halfword [15:0] to register address shifted by 0, 1, 2, or 3 places	`STRH.W <Rxf>, [<Rn>, <Rm>{, LSL #<shift>}]`
Register halfword [15:0] to register address immediate 8-bit offset, preindexed	`STRH{T}.W <Rxf>, [<Rn>, #+/-<offset_8>]{!}`
Register halfword [15:0] to register address immediate 8-bit offset, postindexed	`STRH.W <Rxf>, [<Rn>], #+/-<offset_8>`
Subtract immediate 12-bit value from register value	`SUB{S}.W <Rd>, <Rn>, #<modify_constant(immed_12)>`
Subtract shifted register value from register value	`SUB{S}.W <Rd>, <Rn>, <Rm>{, <shift>}`
Subtract immediate 12-bit value from register value	`SUBW.W <Rd>, <Rn>, #<immed_12>`
Sign extend byte to 32 bits	`SXTB.W <Rd>, <Rm>{, <rotation>}`

Table 1-2 32-bit Cortex-M3 instruction summary (continued)

Operation	Assembler
Sign extend halfword to 32 bits	`SXTH.W <Rd>, <Rm>{, <rotation>}`
Table branch byte	`TBB [<Rn>, <Rm>]`
Table branch halfword	`TBH [<Rn>, <Rm>, LSL #1]`
Exclusive OR register value with immediate 12-bit value	`TEQ.W <Rn>, #<modify_constant(immed_12)>`
Exclusive OR register value with shifted register value	`TEQ.W <Rn>, <Rm>{, <shift}`
Logical AND register value with 12-bit immediate value	`TST.W <Rn>, #<modify_constant(immed_12)>`
Logical AND register value with shifted register value	`TST.W <Rn>, <Rm>{, <shift>}`
Copy bit field from register value to register and zero-extend to 32 bits	`UBFX.W <Rd>, <Rn>, #<lsb>, #<width>`
Unsigned divide	`UDIV<c> <Rd>,<Rn>,<Rm>`
Multiply two unsigned register values and add to a 2-register value	`UMLAL.W <RdLo>, <RdHi>, <Rn>, <Rm>`
Multiply two unsigned register values	`UMULL.W <RdLo>, <RdHi>, <Rn>, <Rm>`
Unsigned saturate	`USAT <c> <Rd>, #<imm>, <Rn>{, <shift>}`
Copy unsigned byte to register and zero-extend to 32 bits	`UXTB.W <Rd>, <Rm>{, <rotation>}`
Copy unsigned halfword to register and zero-extend to 32 bits	`UXTH.W <Rd>, <Rm>{, <rotation>}`
Wait for event	`WFE.W`
Wait for interrupt	`WFI.W`

Chapter 2
Programmer's Model

This chapter describes the processor programmer's model. It contains the following sections:

- *About the programmer's model* on page 2-2
- *Privileged access and User access* on page 2-3
- *Registers* on page 2-4
- *Data types* on page 2-10
- *Memory formats* on page 2-11
- *Instruction set* on page 2-13.

2.1 About the programmer's model

The processor implements the ARM v7-M Architecture. This includes the entire 16-bit Thumb instruction set and the base Thumb-2 32-bit instruction set architecture. The processor cannot execute ARM instructions.

The Thumb instruction set is a subset of the ARM instruction set, re-encoded to 16 bits. It supports higher code density and systems with memory data buses that are 16 bits wide or narrower.

Thumb-2 is a major enhancement to the Thumb *Instruction Set Architecture* (ISA). Thumb-2 enables higher code density than Thumb and offers higher performance with 16/32-bit instructions.

2.1.1 Operating modes

The processor supports two modes of operation, Thread mode and Handler mode:

- Thread mode is entered on Reset, and can be entered as a result of an exception return. Privileged and User (Unprivileged) code can run in Thread mode.
- Handler mode is entered as a result of an exception. All code is privileged in Handler mode.

2.1.2 Operating states

The Cortex-M3 processor can operate in one of two operating states:

- Thumb state. This is normal execution running 16-bit and 32-bit halfword aligned Thumb and Thumb-2 instructions.
- Debug State. When in halting debug.

2.2 Privileged access and User access

Code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources. Handler mode is always privileged. Thread mode can be privileged or unprivileged.

Thread mode is Privileged out of reset, but can be configured to User (Unprivileged) by clearing CONTROL[0] by means of the MSR instruction. User access prevents:

- use of some instructions, such as CPS to set FAULTMASK and PRIMASK.
- access to most registers in System Control Space (SCS).

When Thread mode has been changed from Privileged to User , it cannot change itself back to Privileged. Only a Handler can change the privilege of Thread mode. Handler mode is always Privileged.

2.2.1 Main stack and process stack

Out of reset, all code uses the main stack. An exception handler such as SVC can change the stack used by Thread mode by changing the EXC_RETURN value it uses on exit. All exceptions continue to use the main stack. The stack pointer, r13, is a banked register that switches between SP_main and SP_process. Only one stack, the process stack or the main stack, is visible, by means of r13, at any time.

It is also possible to switch from Main Stack to Process Stack while in Thread Mode by writing to CONTROL[1] using the MSR instruction, as well as being selectable using the EXC_RETURN value from an exit from Handler Mode.

2.3 Registers

The Cortex-M3 processor has the following 32-bit registers:

- 13 general-purpose registers, r0-r12
- stack point alias of banked registers, SP_process and SP_main
- link register, r14
- program counter, r15
- one program status register, *x*PSR.

Figure 2-1 shows the Cortex-M3 register set.

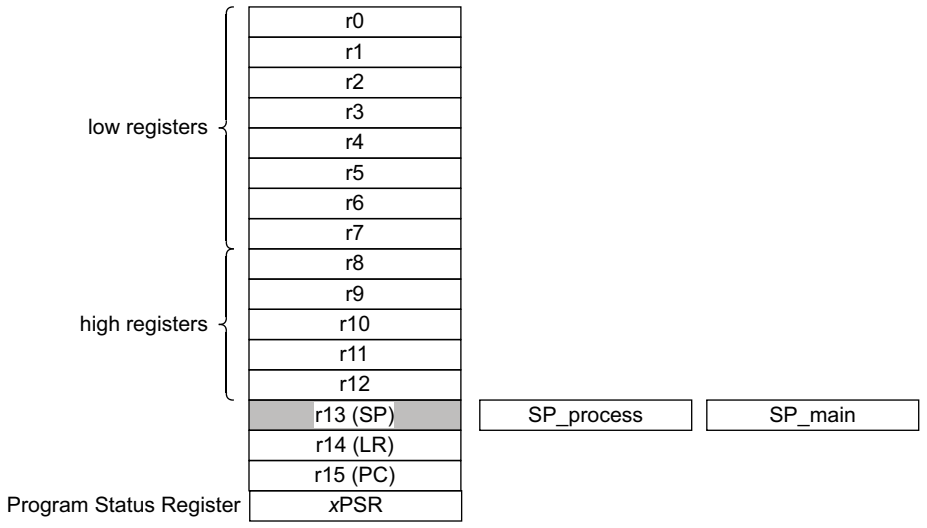


Figure 2-1 Cortex-M3 register set

2.3.1 General-purpose registers

The general-purpose registers r0-r12 have no special architecturally-defined uses. Most instructions that can specify a general-purpose register can specify r0-r12.

Low registers Registers r0-r7 are accessible by all instructions that specify a general-purpose register.

High registers Registers r8-r12 are accessible by all 32-bit instructions that specify a general-purpose register.

Registers r8-r12 are not accessible by all 16-bit instructions.

The r13, r14, and r15 registers have the following special functions:

Stack pointer Register r13 is used as the *Stack Pointer* (SP). Because the SP ignores writes to bits [1:0], it is autoaligned to a word, four-byte boundary.

Handler mode always uses SP_main, but Thread mode can be configured to use either SP_main or SP_process.

Link register Register r14 is the subroutine *Link Register* (LR).

The LR receives the return address from PC when a *Branch and Link* (BL) or *Branch and Link with Exchange* (BLX) instruction is executed.

The LR is also used for exception return.

At all other times, you can treat r14 as a general-purpose register.

Program counter Register r15 is the *Program Counter* (PC).

Bit 0 is always 0, so instructions are always aligned to word or halfword boundaries.

2.3.2 Special-purpose Program Status Registers (xPSR)

Processor status at the system level breaks down into three categories. They can be accessed as individual registers, a combination of any two from three, or a combination of all three using the MRS and MSR instructions:

- *Application PSR*
- *Interrupt PSR* on page 2-6
- *Execution PSR* on page 2-7.

Application PSR

The *Application PSR* (APSR) contains the condition code flags. Before entering an exception, the Cortex-M3 processor saves the condition code flags on the stack. You can access the APSR with the MSR(2) and MRS(2) instructions.

Figure 2-2 shows the fields of the APSR.

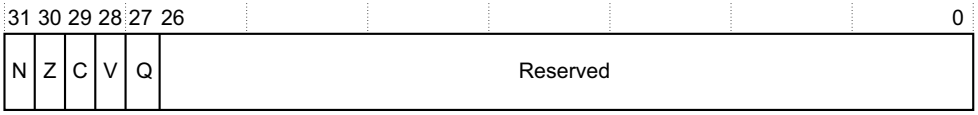


Figure 2-2 Application Program Status Register bit assignments

Table 2-1 describes the fields of the APSR.

Table 2-1 Application Program Status Register bit assignments

Field	Name	Definition
[31]	N	Negative or less than flag: 1 = result negative or less than 0 = result positive or greater than.
[30]	Z	Zero flag: 1 = result of 0 0 = nonzero result.
[29]	C	Carry/borrow flag: 1 = carry or borrow 0 = no carry or borrow.
[28]	V	Overflow flag: 1 = overflow 0 = no overflow.
[27]	Q	Sticky saturation flag.
[26:0]	-	Reserved.

Interrupt PSR

The *Interrupt PSR* (IPSR) contains the ISR number of the current exception activation.

Figure 2-2 on page 2-5 shows the fields of the IPSR.

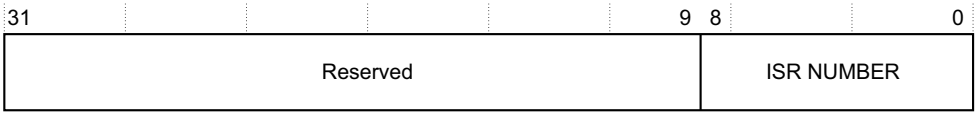


Figure 2-3 Interrupt Program Status Register bit assignments

Table 2-2 describes the fields of the IPSR.

Table 2-2 Interrupt Program Status Register bit assignments

Field	Name	Definition
[31:9]	-	Reserved.
[8:0]	ISR NUMBER	Number of pre-empted exception. Base level = 0 NMI = 2 SVCall = 11 INTISR[0] = 16 INTISR[1] = 17 . . . INTISR[15] = 31 . . . INTISR[239] = 255

Execution PSR

The *Execution PSR* (EPSR) contains two overlapping fields:

- the Interruptible-Continuable Instruction (ICI) field for interrupted load multiple and store multiple instructions
- the execution state field for the If-Then (IT) instruction, and the T-bit (Thumb state bit).

Interruptible-continuable instruction field

Load Multiple (LDM) operations and *Store Multiple* (STM) operations are interruptible. The ICI field of the EPSR holds the information required to continue the load or store multiple from the point at which the interrupt occurred.

If-then state field

The IT field of the EPSR contain the execution state bits for the If-Then instruction.

Note

Because the ICI field and the IT field overlap, load or store multiples within an If-Then block cannot be interrupt-continued.

Figure 2-4 shows the fields of the EPSR.

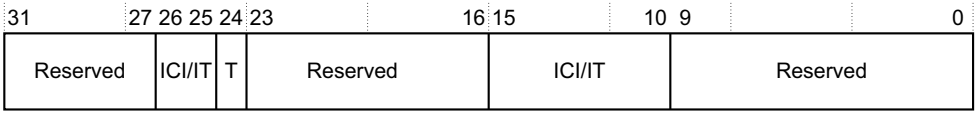


Figure 2-4 Execution Program Status Register

The EPSR is not directly accessible. Two events can modify the EPSR:

- an interrupt occurring during an LDM or STM instruction
- execution of the If-Then instruction.

Table 2-3 describes the fields of the EPSR.

Table 2-3 Bit functions of the Execution PSR

Field	Name	Definition
[31:27]	-	Reserved.
[15:12]	ICI	Interruptible-continuable instruction bits. When an interrupt occurs during an LDM or STM operation, the multiple operation stops temporarily. The EPSR uses bits [15:12] to store the number of the next register operand in the multiple operation. After servicing the interrupt, the processor returns to the register pointed to by [15:12] and resumes the multiple operation. If the ICI field points to a register that is not in the register list of the instruction, the processor continues with the next register in the list, if any.
[15:10]:[26:25]	IT	If-Then bits. These are the execution state bits of the If-Then instruction. They contain the number of instructions in the if-then block and the conditions for their execution.
[24]	T	The T-bit can be cleared using an interworking instruction where bit [0] of the written PC is 0. It can also be cleared by unstacking from an exception where the stacked T bit is 0. Executing an instruction while the T bit is clear causes an INVSTATE exception.
[23:16]	-	Reserved.
[9:0]	-	Reserved.

Base register update in LDM and STM operations

There are cases in which an LDM or STM updates the base register:

- When the instruction specifies base register write-back, the base register changes to the updated address. An abort restores the original base value.
- When the base register is in the register list of an LDM, and is not the last register in the list, the base register changes to the loaded value.

An LDM/STM is restarted rather than continued if:

- the LDM/STM faults
- the LDM/STM is inside an IT.

If an LDM has completed a base load, it is continued from before the base load.

Saved *x*PSR bits

On entering an exception, the processor saves the combined information from the three status registers on the stack.

2.4 Data types

The processor supports the following data types:

- 32-bit words
- 16-bit halfwords
- 8-bit bytes.

Note

Memory systems are expected to support all data types. In particular, the system must support subword writes without corrupting neighboring bytes in that word.

2.5 Memory formats

The processor views memory as a linear collection of bytes numbered in ascending order from 0. For example:

- bytes 0-3 hold the first stored word
- bytes 4-7 hold the second stored word.

The processor can access data words in memory in little-endian format or big-endian format. It always accesses code in little-endian format.

Note

Little-endian is the default memory format for ARM processors.

In little-endian format, the byte with the lowest address in a word is the least-significant byte of the word. The byte with the highest address in a word is the most significant. The byte at address 0 of the memory system connects to data lines 7-0.

In big-endian format, the byte with the lowest address in a word is the most significant byte of the word. The byte with the highest address in a word is the least significant. The byte at address 0 of the memory system connects to data lines 31-24.

Figure 2-5 on page 2-12 shows the difference between little-endian and big-endian memory formats.

The processor contains a configuration pin, **BIGEND**, that enables you to select either the little-endian or BE-8 big-endian format. This configuration pin is sampled on reset. Endianness cannot be changed when out of reset.

Note

- Accesses to System Control Space (SCS) are always performed as little endian.
- Attempts to change endianness while not in reset are ignored.
- PPB space is little-endian only irrespective of the setting of **BIGEND**.

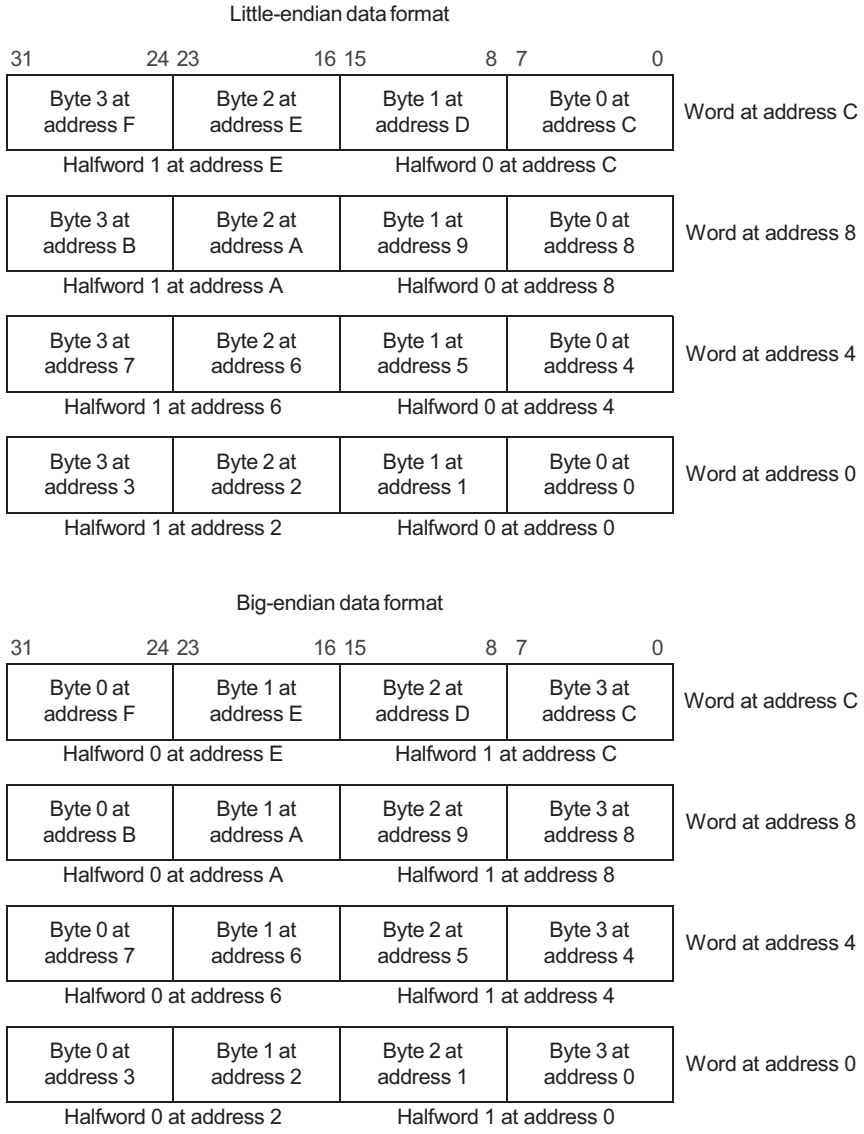


Figure 2-5 Little-endian and big-endian memory formats

2.6 Instruction set

The Cortex-M3 processor does not support ARM instructions.

The Cortex-M3 processor supports all ARMv6 Thumb instructions except those listed in Table 2-4.

Table 2-4 Nonsupported Thumb instructions

Instruction		Action if executed
BLX(1)	Branch with link and exchange	BLX(1) always faults.
SETEND	Set endianness	SETEND always faults. A configuration pin selects Cortex-M3 endianness.

The Cortex-M3 processor supports the Thumb-2 instructions listed in Table 2-5.

Table 2-5 Supported Thumb-2 instructions

Instruction type	Size	Instructions
Data operations	16	ADC, ADD, AND, ASR, BIC, CMN, CMP, CPY, EOR, LSL, LSR, MOV, MUL, MVN, NEG, ORR, ROR, SBC, SUB, TST, REV, REVH, REVSH, SXTB, SXTH, UXTB, and UXTH.
Branches	16	B<cond>, B, BL, BX, and BLX. Note, no BLX with immediate.
Load-store single	16	LDR, LDRB, LDRH, LDRSB, LDRSH, STR, STRB, STRH, and T variants.
Load-store multiple	16	LDMIA, POP, PUSH, and STMIA.
Exception generating	16	BKPT stops in debug if debug enabled, fault if debug disabled. SVC faults to the SVCall handler.
Data operations with immediate	32	ADC{S}. ADD{S}, CMN, RSB{S}, SBC{S}, SUB{S}, CMP, AND{S}, TST, BIC{S}, EOR{S}, TEQ, ORR{S}, MOV{S}, ORN{S}, and MVN{S}.
Data operations with large immediate	32	MOVW, MOVT, ADDW, and SUBW. MOVW and MOVT have a 16-bit immediate. This means they can replace literal loads from memory. ADDW and SUBW have a 12-bit immediate. This means they can replace many from memory literal loads.
Bit-field operations	32	BFI, BFC, UBFX, and SBFX. These are bitwise operations enabling control of position and size in bits. These both support C/C++ bit fields, in structs, in addition to many compare and some AND/OR assignment expressions.

Table 2-5 Supported Thumb-2 instructions (continued)

Instruction type	Size	Instructions
Data operations with three registers	32	ADC{S}. ADD{S}, CMN, RSB{S}, SBC{S}, SUB{S}, CMP, AND{S}, TST, BIC{S}, EOR{S}, TEQ, ORR{S}, MOV{S}, ORN{S}, and MVN{S}. No PKxxx instructions.
Shift operations	32	ASR{S}, LSL{S}, LSR{S}, and ROR {S}.
Miscellaneous	32	REV, REVH, REVSH, RBIT, CLZ, SXTB, SXTH, UXTB, and UXTH. Extension instructions same as corresponding v6 16-bit instructions.
Table branch	32	TBB and TBH table branches for switch/case use. These are LDR with shifts and then branch.
Multiply	32	MUL, MLA, and MLS.
Multiply with 64-bit result	32	UMULL, SMULL, UMLAL, and SMLAL.
Load-store addressing	32	Supports Format PC+/-imm12, Rbase+imm12, Rbase+/-imm8, and adjusted register including shifts. T variants used when in Privilege mode.
Load-store single	32	LDR, LDRB, LDRSB, LDRH, LDRSH, STR, STRB, STRH, and T variants. PLD is a hint, so acts as NOP if no cache.
Load-store multiple	32	STM, LDM, LDRD, STRD, LDC, and STC.
Load-store exclusive	32	LDREX, STREX, LDREXB, LDREXH, STREXB, STREXH, CLREX. Fault if no local monitor. This is IMP DEF. DREXD and STREXD are not included in this profile.
Branches	32	B, BL, and B<cond>. No BLX (1) because always changes state. No BXJ.
System	32	MSR(2) and MRS(2) replace MSR/MRS but also do more. These are used to access the other stacks and also the status registers. CPSIE/CPSID 32-bit forms are not supported. No RFE or SRS.
System	16	CPSIE and CPSID are quick versions of MSR(2) instructions and use the standard Thumb-2 encodings, but only allow use of "i" and "f" and not "a".
Extended32	32	NOP (all forms), Coprocessor (LDC, MCR, MCR2, MCRR, MRC, MRC2, MRRC, and STC), and YIELD (hinted NOP). Note, no MRS(1), MSR(1), or SUBS (PC return link).
Combined branch	16	CBZ and CBNZ (Compare and Branch if register is Zero or Non-Zero).
Extended	16	IT and NOP. This includes YIELD.

Table 2-5 Supported Thumb-2 instructions (continued)

Instruction type	Size	Instructions
Divide	32	SDIV and UDIV. 32/32 divides both signed and unsigned with 32-bit quotient result, no remainder, it can be derived by subtraction. Early out is allowed.
Sleep	16, 32	WFI, WFE, and SEV are in the class of "hinted NOP" instructions used to control sleep behavior.
Barriers	32	ISB, DSB, and DMB are barrier instructions that ensure certain actions have taken place before the next instruction is executed.
Saturation	32	SSAT and USAT perform saturation on a register. They perform the following: Normalize the value using shift test for overflow from a selected bit position, the Q value. Set the xPSR Q bit if so, saturate the value if overflow detected. Saturation refers to the largest unsigned value or the largest/smallest signed value for the size selected.

Note

All coprocessor instructions generate a NOCP fault.

Chapter 3
System Control

This chapter lists the registers that are used to program the Cortex-M3 system. It contains the following section:

- *Summary of processor registers* on page 3-2.

3.1 Summary of processor registers

This section describes the registers that control the system functionality. It contains the following:

- *Nested Vectored Interrupt Controller registers*
- *Core debug registers* on page 3-5
- *System debug registers* on page 3-6
- *Debug interface port registers* on page 3-11
- *Trace Port Interface Unit registers* on page 3-12
- *Embedded Trace Macrocell registers* on page 3-13.

3.1.1 Nested Vectored Interrupt Controller registers

Table 3-1 gives a summary of the *Nested Vectored Interrupt Controller* (NVIC) registers. For a detailed description of the NVIC registers, see Chapter 8 *Nested Vectored Interrupt Controller*.

Table 3-1 NVIC registers

Name of register	Type	Address	Reset value
Interrupt Control Type Register	Read-only	0xE000E004	a
SysTick Control and Status Register	Read/write	0xE000E010	0x00000000
SysTick Reload Value Register	Read/write	0xE000E014	Unpredictable
SysTick Current Value Register	Read/write clear	0xE000E018	Unpredictable
SysTick Calibration Value Register	Read-only	0xE000E01C	STCALIB
Irq 0 to 31 Set Enable Register	Read/write	0xE000E100	0x00000000
.	.	.	.
.	.	.	.
.	.	.	.
Irq 224 to 239 Set Enable Register	Read/write	0xE000E11C	0x00000000
Irq 0 to 31 Clear Enable Register	Read/write	0xE000E180	0x00000000
.	.	.	.
.	.	.	.

Table 3-1 NVIC registers (continued)

Name of register	Type	Address	Reset value
.	.	.	.
Irq 224 to 239 Clear Enable Register	Read/write	0xE000E19C	0x00000000
Irq 0 to 31 Set Pending Register	Read/write	0xE000E200	0x00000000
.	.	.	.
.	.	.	.
.	.	.	.
Irq 224 to 239 Set Pending Register	Read/write	0xE000E21C	0x00000000
Irq 0 to 31 Clear Pending Register	Read/write	0xE000E280	0x00000000
.	.	.	.
.	.	.	.
.	.	.	.
Irq 224 to 239 Clear Pending Register	Read/write	0xE000E29C	0x00000000
Irq 0 to 31 Active Bit Register	Read-only	0xE000E300	0x00000000
.	.	.	.
.	.	.	.
.	.	.	.
Irq 224 to 239 Active Bit Register	Read-only	0xE000E31C	0x00000000
Irq 0 to 31 Priority Register	Read/write	0xE000E400	0x00000000
.	.	.	.
.	.	.	.
.	.	.	.
Irq 236 to 239 Priority Register	Read/write	0xE000E4F0	0x00000000
CPUID Base Register	Read-only	0xE000ED00	0x410FC230
Interrupt Control State Register	Read/write or read-only	0xE000ED04	0x00000000

Table 3-1 NVIC registers (continued)

Name of register	Type	Address	Reset value
Vector Table Offset Register	Read/write	0xE000ED08	0x00000000
Application Interrupt/Reset Control Register	Read/write	0xE000ED0C	0x00000000
System Control Register	Read/write	0xE000ED10	0x00000000
Configuration Control Register	Read/write	0xE000ED14	0x00000000
System Handlers 4-7 Priority Register	Read/write	0xE000ED18	0x00000000
System Handlers 8-11 Priority Register	Read/write	0xE000ED1C	0x00000000
System Handlers 12-15 Priority Register	Read/write	0xE000ED20	0x00000000
System Handler Control and State Register	Read/write	0xE000ED24	0x00000000
Configurable Fault Status Registers	Read/write	0xE000ED28	0x00000000
Hard Fault Status Register	Read/write	0xE000ED2C	0x00000000
Debug Fault Status Register	Read/write	0xE000ED30	0x00000000
Mem Manage Address Register	Read/write	0xE000ED34	Unpredictable
Bus Fault Address Register	Read/write	0xE000ED38	Unpredictable
PFR0: Processor Feature register0	Read-only	0xE000ED40	0x00000030
PFR1: Processor Feature register1	Read-only	0xE000ED44	0x00000200
DFR0: Debug Feature register0	Read-only	0xE000ED48	0x00100000
AFR0: Auxiliary Feature register0	Read-only	0xE000ED4C	0x00000000
MMFR0: Memory Model Feature register0	Read-only	0xE000ED50	0x00000030
MMFR1: Memory Model Feature register1	Read-only	0xE000ED54	0x00000000
MMFR2: Memory Model Feature register2	Read-only	0xE000ED58	0x00000000
MMFR3: Memory Model Feature register3	Read-only	0xE000ED5C	0x00000000
ISAR0: ISA Feature register0	Read-only	0xE000ED60	0x01141110
ISAR1: ISA Feature register1	Read-only	0xE000ED64	0x02111000
ISAR2: ISA Feature register2	Read-only	0xE000ED68	0x21112231
ISAR3: ISA Feature register3	Read-only	0xE000ED6C	0x01111110

Table 3-1 NVIC registers (continued)

Name of register	Type	Address	Reset value
ISAR4: ISA Feature register4	Read-only	0xE000ED70	0x01310102
Software Trigger Interrupt Register	Write Only	0xE000EF00	-
Peripheral identification register (PERIPHID4)	Read-only	0xE000EFD0	0x04
Peripheral identification register (PERIPHID5)	Read-only	0xE000EFD4	0x00
Peripheral identification register (PERIPHID6)	Read-only	0xE000EFD8	0x00
Peripheral identification register (PERIPHID7)	Read-only	0xE000EFDC	0x00
Peripheral identification register Bits 7:0 (PERIPHID0)	Read-only	0xE000EFE0	0x00
Peripheral identification register Bits 15:8 (PERIPHID1)	Read-only	0xE000EFE4	0xB0
Peripheral identification register Bits 23:16 (PERIPHID2)	Read-only	0xE000EFE8	0x0B
Peripheral identification register Bits 31:24 (PERIPHID3)	Read-only	0xE000EFEC	0x00
Component identification register Bits 7:0 (PCELLID0)	Read Only	0xE000EFF0	0x0D
Component identification register Bits 15:8 (PCELLID1)	Read-only	0xE000EFF4	0xE0
Component identification register Bits 23:16 (PCELLID2)	Read-only	0xE000EFF8	0x05
Component identification register Bits 31:24 (PCELLID3)	Read-only	0xE000EFFC	0xB1

a. Reset value depends on the number of interrupts defined.

3.1.2 Core debug registers

Table 3-2 gives a summary of the core debug registers. For a detailed description of the core debug registers, see Chapter 10 *Core Debug*.

Table 3-2 Core debug registers

Name of register	Type	Address	Reset Value
Debug Halting Control and Status Register	Read/Write	0xE000EDF0	0x00000000[a]
Debug Core Register Selector Register	Write-only	0xE000EDF4	-
Debug Core Register Data Register	Read/Write	0xE000EDF8	-
Debug Exception and Monitor Control Register.	Read/Write	0xE000EDFC	0x00000000[b]

a. Bits 5, 3, 2, 1, 0 are reset by **PORESETn**. Bit[1] is also reset by **SYSRESETn** and writing a 1 to the **VECTRESET** bit of the Application Interrupt and Reset Control Register.
b. Bits 16,17,18,19 are also reset by **SYSRESETn** and writing a 1 to the **VECTRESET** bit of the Application Interrupt and Reset Control Register.

3.1.3 System debug registers

This section lists the system debug registers.

Flash Patch and Breakpoint registers

Table 3-3 gives a summary of the *Flash Patch and Breakpoint* (FPB) registers. For a detailed description of the FPB registers, see Chapter 11 *System Debug*.

Table 3-3 Flash patch register summary

Name	Type	Address	Reset value	Description
FP_CTRL	Read/write	0xE0002000	Bit[0] is reset to 1'b0	Flash Patch Control Register
FP_REMAP	Read/write	0xE0002004	-	Flash Patch Remap Register
FP_COMP0	Read/write	0xE0002008	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP1	Read/write	0xE000200C	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP2	Read/write	0xE0002010	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP3	Read/write	0xE0002014	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP4	Read/write	0xE0002018	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP5	Read/write	0xE000201C	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP6	Read/write	0xE0002020	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
FP_COMP7	Read/write	0xE0002024	Bit[0] is reset to 1'b0	Flash Patch Comparator Registers
PERIPID4	Read-only	0xE0002FD0	-	Value 0x04
PERIPID5	Read-only	0xE0002FD4	-	Value 0x00
PERIPID6	Read-only	0xE0002FD8	-	Value 0x00
PERIPID7	Read-only	0xE0002FDC	-	Value 0x00
PERIPID0	Read-only	0xE0002FE0	-	Value 0x03
PERIPID1	Read-only	0xE0002FE4	-	Value 0xB0

Table 3-3 Flash patch register summary (continued)

Name	Type	Address	Reset value	Description
PERIPID2	Read-only	0xE0002FE8	-	Value 0x0B
PERIPID3	Read-only	0xE0002FEC	-	Value 0x00
PCELLID0	Read-only	0xE0002FF0	-	Value 0x0D
PCELLID1	Read-only	0xE0002FF4	-	Value 0xE0
PCELLID2	Read-only	0xE0002FF8	-	Value 0x05
PCELLID3	Read-only	0xE0002FFC	-	Value 0xB1

Data Watchpoint and Trigger registers

Table 3-4 gives a summary of the *Data Watchpoint and Trigger* (DWT) registers. For a detailed description of the DWT registers, see Chapter 11 *System Debug*.

Table 3-4 DWT register summary

Name	Type	Address	Reset value	Description
DWT_CTRL	Read/write	0xE0001000	0x00000000	DWT Control Register
DWT_CYCCNT	Read/write	0xE0001004	0x00000000	DWT Current PC Sampler Cycle Count Register
DWT_CPICNT	Read/write	0xE0001008	-	DWT Current CPI Count Register
DWT_EXCCNT	Read/write	0xE000100C	-	DWT Current Interrupt Overhead Count Register
DWT_SLEEPCNT	Read/write	0xE0001010	-	DWT Current Sleep Count Register
DWT_LSUCNT	Read/write	0xE0001014	-	DWT Current LSU Count Register
DWT_FOLDCNT	Read/write	0xE0001018	-	DWT Current Fold Count Register
DWT_COMP0	Read/write	0xE0001020	-	DWT Comparator Register
DWT_MASK0	Read/write	0xE0001024	-	DWT Mask Registers
DWT_FUNCTION0	Read/write	0xE0001028	0x00000000	DWT Function Registers
DWT_COMP1	Read/write	0xE0001030	-	DWT Comparator Register
DWT_MASK1	Read/write	0xE0001034	-	DWT Mask Registers

Table 3-4 DWT register summary (continued)

Name	Type	Address	Reset value	Description
DWT_FUNCTION1	Read/write	0xE0001038	0x00000000	DWT Function Registers
DWT_COMP2	Read/write	0xE0001040	-	DWT Comparator Register
DWT_MASK2	Read/write	0xE0001044	-	DWT Mask Registers
DWT_FUNCTION2	Read/write	0xE0001048	0x00000000	DWT Function Registers
DWT_COMP3	Read/write	0xE0001050	-	DWT Comparator Register
DWT_MASK3	Read/write	0xE0001054	-	DWT Mask Registers
DWT_FUNCTION3	Read/write	0xE0001058	0x00000000	DWT Function Registers
PERIPHID4	Read-only	0xE0001FD0	0x04	Value 0x04
PERIPHID5	Read-only	0xE0001FD4	0x00	Value 0x00
PERIPHID6	Read-only	0xE0001FD8	0x00	Value 0x00
PERIPHID7	Read-only	0xE0001FDC	0x00	Value 0x00
PERIPHID0	Read-only	0xE0001FE0	0x02	Value 0x02
PERIPHID1	Read-only	0xE0001FE4	0xB0	Value 0xB0
PERIPHID2	Read-only	0xE0001FE8	0x0B0	Value 0x0B
PERIPHID3	Read-only	0xE0001FEC	0x00	Value 0x00
PCELLID0	Read-only	0xE0001FF0	0x0D	Value 0x0D
PCELLID1	Read-only	0xE0001FF4	0xE0	Value 0xE0
PCELLID2	Read-only	0xE0001FF8	0x05	Value 0x05
PCELLID3	Read-only	0xE0001FFC	0xB1	Value 0xB1

Instrumentation Trace Macrocell registers

Table 3-5 gives a summary of the *Instrumentation Trace Macrocell* (ITM) registers. For a detailed description of the ITM registers, see Chapter 11 *System Debug*

Table 3-5 ITM register summary

Name	Type	Address	Reset value
Stimulus Ports 0-31	Read/write	0xE0000000-0xE000007C	-
Trace Enable	Read/write	0xE0000E00	0x00000000
Trace Privilege	Read/write	0xE0000E40	0x00000000
Control Register	Read/write	0xE0000E80	0x00000000
Integration Write	Write-only	0xE0000EF8	0x00000000
Integration Read	Read-only	0xE0000EFC	0x00000000
Integration Mode Control	Read/write	0xE0000F00	0x00000000
Lock Access Register	Write-only	0xE0000FB0	0x00000000
Lock Status Register	Read-only	0xE0000FB4	0x00000003
PERIPHID4	Read-only	0xE0000FD0	0x00000004
PERIPHID5	Read-only	0xE0000FD4	0x00000000
PERIPHID6	Read-only	0xE0000FD8	0x00000000
PERIPHID7	Read-only	0xE0000FDC	0x00000000
PERIPHID0	Read-only	0xE0000FE0	0x00000002
PERIPHID1	Read-only	0xE0000FE4	0x000000B0
PERIPHID2	Read-only	0xE0000FE8	0x0000000B
PERIPHID3	Read-only	0xE0000FEC	0x00000000
PCELLID0	Read-only	0xE0000FF0	0x0000000D
PCELLID1	Read-only	0xE0000FF4	0x000000E0
PCELLID2	Read-only	0xE0000FF8	0x00000005
PCELLID3	Read-only	0xE0000FFC	0x000000B1

AHB-AP registers

Table 3-6 gives a summary of the *Advanced High Performance Bus Access Port* (AHB-AP) registers. For a detailed description of the AHB-AP registers, see Chapter 11 *System Debug*.

Table 3-6 AHB-AP register summary

Name	Type	Address	Reset value	Description
Control and Status Word (CSW)	Read/write	0x00	See Register	AHB-AP Control and Status Word Register
Transfer Address (TAR)	Read/write	0x04	0x00000000	AHB-AP Transfer Address Register
Data Read/write (DRW)	Read/write	0x0C	-	AHB-AP Data Read/Write Register
Banked Data 0 (BD0)	Read/write	0x10	-	AHB-AP Banked Data Registers
Banked Data 1 (BD1)	Read/write	0x14	-	AHB-AP Banked Data Registers
Banked Data 2 (BD2)	Read/write	0x18	-	AHB-AP Banked Data Registers
Banked Data 3 (BD3)	Read/write	0x1C	-	AHB-AP Banked Data Registers
Debug ROM Address	Read only	0xF8	0xE000E000	AHB-AP Debug ROM Address Register
Identification Register (IDR)	Read only	0xFC	0x04770011	AHB-AP ID Register

3.1.4 Debug interface port registers

Table 3-7 gives a summary of the debug interface port registers. For a detailed description of the debug interface port registers, see Chapter 12 *Debug Port*.

Table 3-7 Summary of Debug Port registers

Name	Description	JTAG-DP	SW-DP	For description see section
ABORT	DAP Abort Register	Yes	Yes	The Abort Register
IDCODE	ID Code Register	Yes	Yes	The Identification Code Register
CTRL/STAT	DP Control/Status Register	Yes	Yes	The Control/Status Register
SELECT	Select Register	Yes	Yes	The AP Select Register
RDBUFF	Read Buffer	Yes	Yes	The Read Buffer Register
WCR	Wire Control Register	No	Yes	The Wire Control Register
RESEND	Read Resend Register	No	Yes	The Read Resend Register

3.1.5 Memory Protection Unit registers

Table 3-8 gives a summary of the *Memory Protection Unit* (MPU) registers. For a detailed description of the MPU registers, see Chapter 9 *Memory Protection Unit*.

Table 3-8 MPU registers

Name of register	Type	Address	Reset value
MPU Type Register	Read Only	`0xE000ED90`	`0x00000800`
MPU Control Register	Read/Write	`0xE000ED94`	`0x00000000`
MPU Region Number register	Read/Write	`0xE000ED98`	-
MPU Region Base Address register	Read/Write	`0xE000ED9C`	-
MPU Region Attribute and Size registers	Read/Write	`0xE000EDA0`	-
MPU Alias 1 Region Base Address register	Alias of `D9C`	`0xE000EDA4`	-
MPU Alias 1 Region Attribute and Size register	Alias of `DA0`	`0xE000EDA8`	-
MPU Alias 2 Region Base Address register	Alias of `D9C`	`0xE000EDAC`	-

Table 3-8 MPU registers (continued)

Name of register	Type	Address	Reset value
MPU Alias 2 Region Attribute and Size register	Alias of DA0	0xE000EDB0	-
MPU Alias 3 Region Base Address register	Alias of D9C	0xE000EDB4	-
MPU Alias 3 Region Attribute and Size register	Alias of DA0	0xE000EDB8	-

3.1.6 Trace Port Interface Unit registers

Table 3-9 gives a summary of the *Trace Port Interface Unit* (TPIU) registers. For a detailed description of the TPIU registers, see Chapter 13 *Trace Port Interface Unit*.

Table 3-9 TPIU registers

Name of register	Type	Address	Reset value
Supported Port Sizes Register	Read-only	0xE0040000	0bxx0x
Current Port Size Register	Read/write	0xE0040004	0x01
Current Output Speed Divisors Register	Read/write	0xE0040010	0x0000
Selected Pin Protocol Register	Read/write	0xE00400F0	0x01
Formatter and Flush Status Register	Read/write	0xE0040300	0x08
Formatter and Flush Control Register	Read-only	0xE0040304	0x00 or 0x102
Formatter Synchronization Counter Register	Read-only	0xE0040308	0x00
Integration Register: ITATBCTR2	Read-only	0xE0040EF0	0x0
Integration Register: ITATBCTR0	Read-only	0xE0040EF8	0x0

3.1.7 Embedded Trace Macrocell registers

Table 3-10 gives a summary of the *Embedded Trace Macrocell* (ETM) registers. For a detailed description of the ETM registers, see Chapter 15 *Embedded Trace Macrocell*.

Table 3-10 ETM registers

Name	Type	Address	Present
ETM Control	Read/write	0xE0041000	Yes
Configuration Code	Read-only	0xE0041004	Yes
Trigger event	Write-only	0xE0041008	Yes
ASIC Control	Write-only	0xE004100C	No
ETM Status	Read-only or read/write	0xE0041010	Yes
System Configuration	Read-only	0xE0041014	Yes
TraceEnable	Write-only	0xE0041018, 0xE004101C	No
TraceEnable Event	Write-only	0xE0041020	Yes
TraceEnable Control 1	Write-only	0xE0041024	Yes
FIFOFULL Region	Write-only	0xE0041028	No
FIFOFULL Level	Write-only or read/write	0xE004102C	Yes
ViewData	Write-only	0xE0041030-0xE004103C	No
Address Comparators	Write-only	0xE0041040- 0xE004113C	No
Counters	Write-only	0xE0041140-0xE004157C	No
Sequencer	Read/write	0xE0041180- 0xE0041194, 0xE0041198	No
External Outputs	Write-only	0xE00411A0-0xE00411AC	No
CID Comparators	Write-only	0xE00411B0-0xE00411BC	No
Implementation specific	Write-only	0xE00411C0-0xE00411DC	No
Synchronization Frequency	Write-only	0xE00411E0	No
ETM ID	Read-only	0xE00411E4	Yes
Configuration Code Extension	Read-only	0xE00411E8	Yes
Extended External Input Selector	Write-only	0xE00411EC	No

Table 3-10 ETM registers (continued)

Name	Type	Address	Present
TraceEnable Start/Stop Embedded ICE	Read/write	0xE00411F0	Yes
Embedded ICE Behavior Control	Write-only	0xE00411F4	No
CoreSight Trace ID	Read/write	0xE0041200	Yes
OS Save/Restore	Write-only	0xE0041304-0xE0041308	No
ITMISCIN	Read-only	0xE0041EE0	Yes
ITTRIGOUT	Write-only	0xE0041EE8	Yes
ITATBCTR2	Read-only	0xE0041EF0	Yes
ITATBCTR0	Write-only	0xE0041EF8	Yes
Integration Mode Control	Read/write	0xE0041F00	Yes
Claim Tag	Read/write	0xE0041FA0-0xE0041FA4	Yes
Lock Access	Write-only	0xE0041FB0-0xE0041FB4	Yes
Authentication Status	Read-only	0xE0041FB8	Yes
Device Type	Read-only	0xE0041FCC	Yes
Peripheral ID 4	Read-only	0xE0041FD0	Yes
Peripheral ID 5	Read-only	0xE0041FD4	Yes
Peripheral ID 6	Read-only	0xE0041FD8	Yes
Peripheral ID 7	Read-only	0xE0041FDC	Yes
Peripheral ID 0	Read-only	0xE0041FE0	Yes
Peripheral ID 1	Read-only	0xE0041FE4	Yes
Peripheral ID 2	Read-only	0xE0041FE8	Yes
Peripheral ID 3	Read-only	0xE0041FEC	Yes
Component ID 0	Read-only	0xE0041FF0	Yes
Component ID 1	Read-only	0xE0041FF4	Yes
Component ID 2	Read-only	0xE0041FF8	Yes
Component ID 3	Read-only	0xE0041FFC	Yes

Chapter 4
Memory Map

This chapter describes the processor fixed memory map and its bit-banding feature. It contains the following sections:

- *About the memory map* on page 4-2
- *Bit-banding* on page 4-5
- *ROM memory table* on page 4-8.

4.1 About the memory map

Figure 4-1 shows the fixed Cortex-M3 memory map.

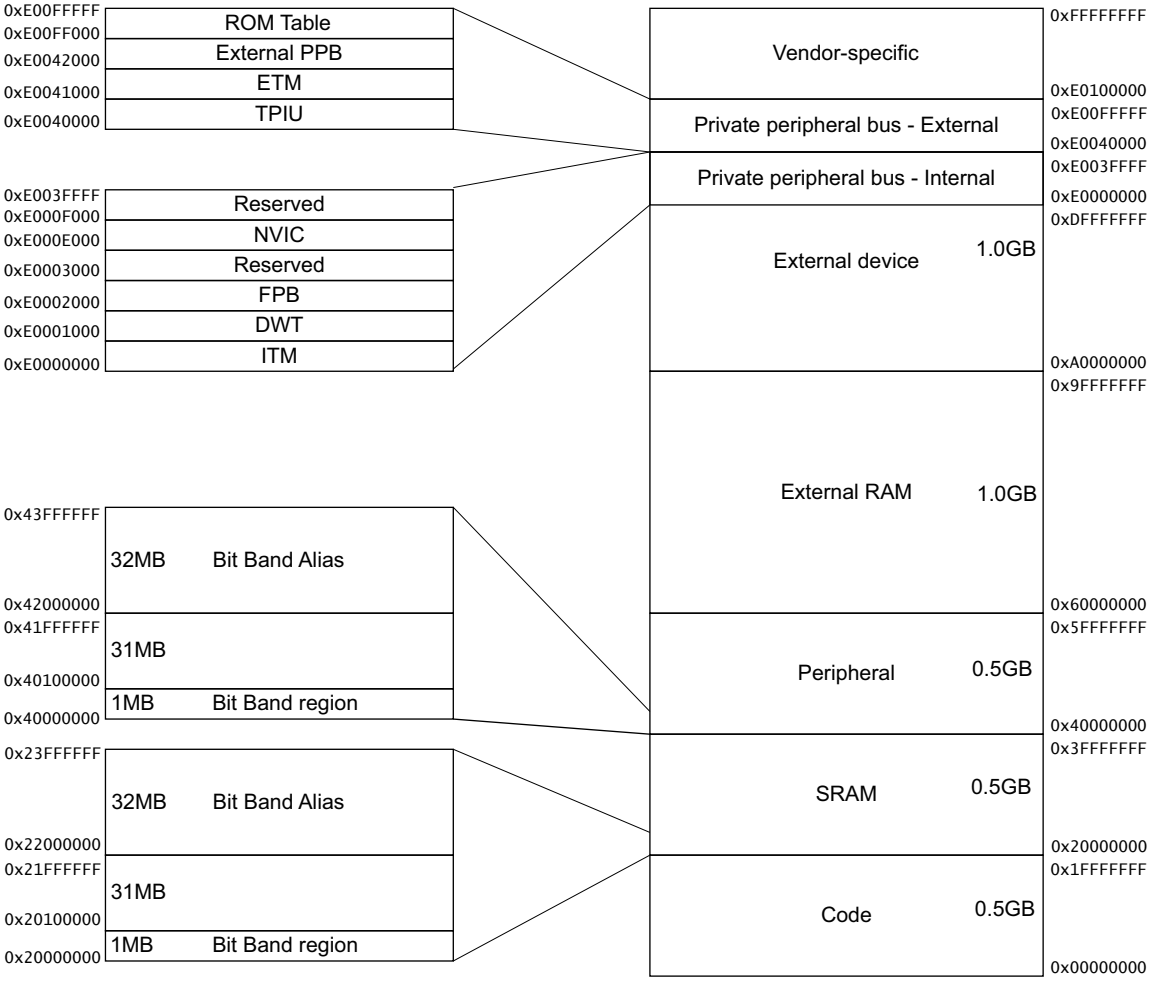


Figure 4-1 The Cortex-M3 Memory Map

Table 4-1 shows the processor interfaces that are addressed by the different memory map regions

Table 4-1 Memory interfaces

Memory Map	Interface
Code	Instruction fetches are performed over the ICode bus. Data accesses are performed over the DCode bus.
SRAM	Instruction fetches and data accesses are performed over the system bus.
SRAM_bitband	Alias region. Data accesses are aliases. Instruction accesses are not aliases.
Peripheral	Instruction fetches and data accesses are performed over the system bus.
Periph_bitband	Alias region. Data accesses are aliases. Instruction accesses are not aliases.
External RAM	Instruction fetches and data accesses are performed over the system bus.
External Device	Instruction fetches and data accesses are performed over the system bus.
Private Peripheral Bus	Accesses to ITM, NVIC, FPB, DWT and MPU are performed to the processor internal Private Peripheral Bus. Accesses to the TPIU, ETM, and System areas of the PPB memory map are performed over the external Private Peripheral Bus interface. This memory region is Execute Never (XN), and so instruction fetches are prohibited. This cannot be changed by an MPU, if present.
System	System segment for vendor system peripherals. This memory region is Execute Never (XN), and so instruction fetches are prohibited. This cannot be changed by an MPU, if present.

Table 4-2 shows the permissions of the processor memory regions.

Table 4-2 Memory region permissions

Name	Region	Device type	XN	Cache
Code	0x00000000-0x1FFFFFFF	Normal	-	WT
SRAM	0x20000000-0x3FFFFFFF	Normal	-	WBWA
SRAM_bitband	0x22000000-0x23FFFFFF	Internal	-	-
Peripheral	0x40000000-0x5FFFFFFF	Device	XN	-
Periph_bit band	0x42000000-0x43FFFFFF	Internal	XN	-
External RAM	0x60000000-0x7FFFFFFF	Normal	-	WBWA
External RAM	0x80000000-0x9FFFFFFF	Normal	-	WT

Table 4-2 Memory region permissions (continued)

Name	Region	Device type	XN	Cache
External Device	0xA0000000-0xBFFFFFFF	Device	XN	Shared
External Device	0xC0000000-0xDFFFFFFF	Device	XN	-
Private Peripheral Bus	0xE0000000-0xE00FFFFF	SO	XN	-
System	0xE0100000-0xFFFFFFFF	Device	XN	-

Note

Private Peripheral Bus and System space at 0xE0100000 - 0xFFFFFFFF are permanently XN. This cannot be overridden by the *Memory Protection Unit* (MPU).

For a description of the processor bus interfaces, see Chapter 14 *Bus Interface*.

4.2 Bit-banding

The processor memory map includes two bit-band regions. These occupy the lowest 1MB of the SRAM and Peripheral memory regions respectively. These bit-band regions map each word in an alias region of memory to a bit in a bit-band region of memory.

The Cortex-M3 memory map has two 32-MB alias regions that map to two 1-MB bit-band regions:

- Accesses to the 32-MB SRAM alias region map to the 1-MB SRAM bit-band region.
- Accesses to the 32-MB peripheral alias region map to the 1-MB peripheral bit-band region.

A mapping formula shows how to reference each word in the alias region to a corresponding bit, or target bit, in the bit-band region. The mapping formula is:

```
bit_word_offset = (byte_offset x 32) + (bit_number × 4)

bit_word_addr = bit_band_base + bit_word_offset
```

where:

- `bit_word_offset` is the position of the target bit in the bit-band memory region
- `bit_word_addr` is the address of the word in the alias memory region that maps to the targeted bit.
- `bit_band_base` is the starting address of the alias region
- `byte_offset` is the number of the byte in the bit-band region that contains the targeted bit
- `bit_number` is the bit position (0-7) of the targeted bit.

Figure 4-2 on page 4-6 shows examples of bit-band mapping between the SRAM bit-band alias region and the SRAM bit-band region:

- the alias word at `0x23FFFFE0` maps to bit 0 of the bit-band byte at `0x200FFFFC`: `0x23FFFFE0` = `0x22000000` + (`0xFFFFF`∗`32`) + 0*4
- the alias word at `0x23FFFFFC` maps to bit 7 of the bit-band byte at `0x200FFFFC`: `0x23FFFFFC` = `0x22000000` + (`0xFFFFFF`∗`32`) + 7*4
- the alias word at `0x22000000` maps to bit 0 of the bit-band byte at `0x20000000`: `0x22000000` = `0x22000000` + (0*32) + 0 *4
- the alias word at `0x2200001C` maps to bit 7 of the bit-band byte at `0x20000000`: `0x2200001C` = `0x22000000` + (0*32) + 7*4.

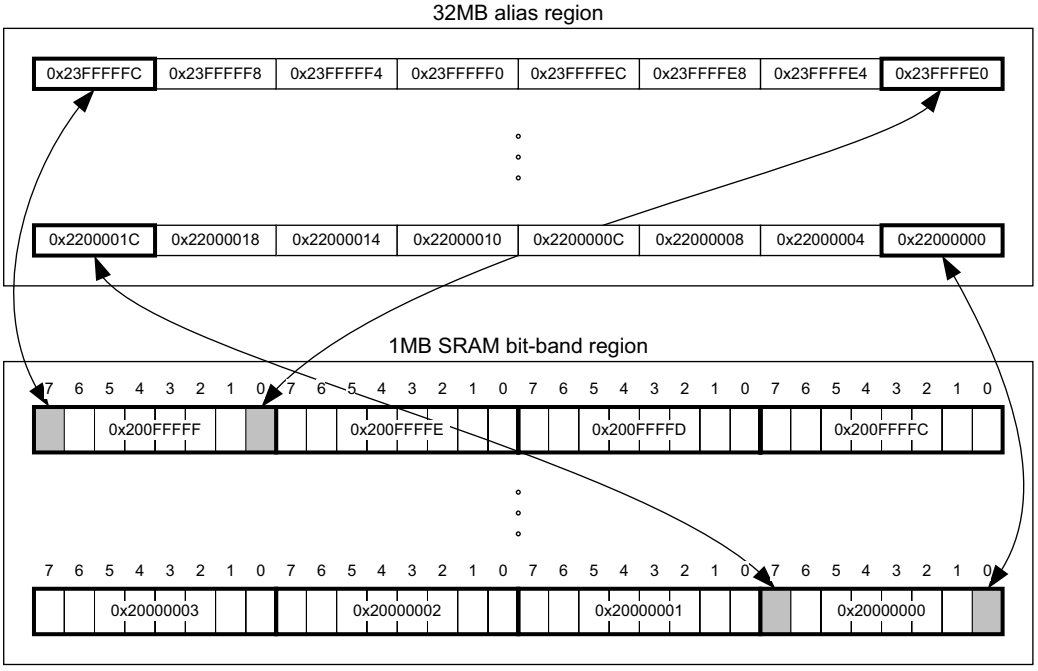


Figure 4-2 Bit-band mapping

4.2.1 Directly accessing an alias region

Writing to a word in the alias region has the same effect as a read-modify-write operation on the targeted bit in the bit-band region.

Bit 0 of the value written to a word in the alias region determines the value written to the targeted bit in the bit-band region. Writing a value with bit 0 set writes a 1 to the bit-band bit, and writing a value with bit 0 cleared writes a 0 to the bit-band bit.

Bits [31:1] of the alias word have no effect on the bit-band bit. Writing `0x01` has the same effect as writing `0xFF`. Writing `0x00` has the same effect as writing `0x0E`.

Reading a word in the alias region returns either `0x01` or `0x00`. A value of `0x01` indicates that the targeted bit in the bit-band region is set. A value of `0x00` indicates that the targeted bit is clear. Bits [31:1] will be zero.

Note

Big endian accesses to the bit-band alias region must be byte-sized. Otherwise, the accesses are unpredictable.

4.2.2 Directly accessing a bit-band region

The bit-band region can be directly accessed with normal reads and writes and writes to that region.

4.3 ROM memory table

The ROM memory is described in Table 4-3.

Table 4-3 Cortex-M3 ROM table

Offset	Value	Name	Description
0x000	0xFFF0F003	NVIC	Points to the NVIC at 0xE000E000.
0x004	0xFFF02003	DWT	Points to the Data Watchpoint and Trace block at 0xE0001000.
0x008	0xFFF03003	FPB	Points to the Flash Patch and Breakpoint block at 0xE0002000.
0x00C	0xFFF01003	ITM	Points to the Instrumentation Trace block at 0xE0000000.
0x010	0xFFF41002 or 003 if present	TPIU	Points to the TPIU. Value has bit 0 set to 1 if TPIU is fitted. TPIU is at 0xE0040000.
0x014	0xFFF41002 or 003 if present	ETM	Points to the ETM. Value has bit 0 set to 1 if ETM is fitted. ETM is at 0xE0041000.
0x018	0	End	Marks the end of the ROM table. If CoreSight components are added, they are added starting from this location and the End marker is moved to the next location after the additional components.
0xFCC	0x1	MEMTYPE	MEMTYPE field has bit 0 defined for “System memory access” if 1, debug only if 0.
0xFD0	0x0	PID4	-
0xFD4	0x0	PID5	-
0xFD8	0x0	PID6	-
0xFDC	0x0	PID7	-
0xFE0	0x0	PID0	-
0xFE4	0x0	PID1	-
0xFE8	0x0	PID2	-
0xFEC	0x0	PID3	-
0xFF0	0x0D	CID0	-

Table 4-3 Cortex-M3 ROM table

Offset	Value	Name	Description
0xFF4	0x10	CID1	-
0xFF8	0x05	CID2	-
0xFFC	0xB1	CID3	-

Chapter 5
Exceptions

This chapter describes the exception model of the processor. It contains the following sections:

- *About the exception model* on page 5-2
- *Exception types* on page 5-3
- *Exception priority* on page 5-5
- *Pre-emption* on page 5-10
- *Tail-chaining* on page 5-13
- *Late-arriving* on page 5-14
- *Exit* on page 5-16
- *Resets* on page 5-19
- *Exception control transfer* on page 5-23
- *Setting up multiple stacks* on page 5-24
- *Abort model* on page 5-26
- *Activation levels* on page 5-31
- *Flowcharts* on page 5-33.

5.1 About the exception model

The processor and the *Nested Vectored Interrupt Controller* (NVIC) prioritize and handle all exceptions. All exceptions are handled in Handler Mode. Processor state is automatically stored to the stack on an exception, and automatically restored from the stack at the end of the *Interrupt Service Routine* (ISR). The vector is fetched in parallel to the state saving, which enables efficient interrupt entry. The processor supports tail-chaining, which enables back-to-back interrupts to be performed without the overhead of state saving and restoration. The following features enable efficient, low latency exception handling:

- Automatic state saving and restoring. The processor pushes state registers on the stack before entering the ISR, and pops them after exiting the ISR with no instruction overhead.
- Automatic reading of the vector table entry that contains the ISR address in code memory or data SRAM. This is performed in parallel to the state saving.
- Support for tail-chaining. In tail-chaining, the processor handles back-to-back interrupts without popping and pushing registers between ISRs.
- Dynamic reprioritization of interrupts.
- Closely-coupled interface between the Cortex-M3 core and the NVIC to enable early processing of interrupts and processing of late-arriving interrupts with higher priority.
- Configurable number of interrupts, from 1 to 240.
- Configurable number of interrupt priorities, from 1 to 8 bits (1 to 256 levels).
- Separate stacks and privilege levels for Handler and Thread modes.
- ISR control transfer using the calling conventions of the C/C++ standard *Procedure Call Standard for the ARM Architecture* (PCSAA).
- Priority masking to support critical regions.

Note

The number of interrupts, and bits of interrupt priority, are configured during implementation. Software can choose only to enable a subset of the configured number of interrupts, and can choose how many bits of the configured priorities to use.

5.2 Exception types

Various types of exceptions exist in the processor. A fault is an exception which results from an error condition due to instruction execution. Faults can be reported synchronously or asynchronously to the instruction which caused them. In general, faults are reported synchronously. The Imprecise BusFault is an asynchronous fault supported in the ARMv7-M profile. A synchronous fault is always reported with the instruction which caused the fault. An asynchronous fault does not guarantee how it is reported with respect to the instruction which caused the fault.

For more information on exceptions see the *ARMv7-M Architecture Reference Manual*.

Table 5-1 shows the exception types, priority, and position. Position refers to the word offset from the start of the vector table. The lower numbers shown in the Priority column of the table are higher priority. Also shown is how the types are activated, synchronously or asynchronously. The exact meaning and use of priorities is explained in *Exception priority* on page 5-5.

Table 5-1 Exception types

Exception type	Position	Priority	Description
-	0	-	Stack top is loaded from first entry of vector table on reset.
Reset	1	-3 (highest)	Invoked on power up and warm reset. On first instruction, drops to lowest priority (Thread Mode). This is asynchronous.
Non-maskable Interrupt	2	-2	Cannot be stopped or preempted by any exception but reset. This is asynchronous.
Hard Fault	3	-1	All classes of Fault, when the fault cannot activate because of priority or the Configurable fault handler has been disabled. This is synchronous.
Memory Management	4	settable[a]	MPU mismatch, including access violation and no match. This is synchronous. This is used even if the MPU is disabled or not present, to support the XN regions of the default memory map.
Bus Fault	5	settable[b]	Pre-fetch fault, memory access fault, and other address/memory related. This is synchronous when precise and asynchronous when imprecise.
Usage Fault	6	settable	Usage fault, such as Undefined instruction executed or illegal state transition attempt. This is synchronous.
-	7-10	-	Reserved
SVCall	11	settable	System service call with SVC instruction. This is synchronous.

Table 5-1 Exception types

Exception type	Position	Priority	Description
Debug Monitor	12	settable	Debug monitor, when not halting. This is synchronous, but only active when enabled. It does not activate if lower priority than the current activation.
-	13	-	Reserved
PendSV	14	settable	Pendable request for system service. This is asynchronous and only pended by software.
SysTick	15	settable	System tick timer has fired. This is asynchronous.
External Interrupt	16 and above	settable	Asserted from outside the core, **INTISR[239:0]**, and fed through the NVIC (prioritized). These are all asynchronous.

a. You can change this priority of this exception, See *System Handler Priority Registers bit assignments* on page 8-27. *Settable* is an NVIC priority value of 0 to N, where N is the largest priority value implemented. Internally, the highest user-settable priority (0) is treated as 4.

b. You can enable or disable this fault. See *System Handler Control and State Register bit assignments* on page 8-28.

5.3 Exception priority

Table 5-2 shows how priority affects when and how the processor takes an exception. It lists the actions an exception can take based on priority.

Table 5-2 Priority-based actions of exceptions

Action	Description
Pre-emption	New exception has higher priority than current exception priority or thread and interrupts current flow. This is the response to a pended interrupt, causing an ISR to be entered if the pended interrupt is higher priority than the active ISR or thread. When one ISR pre-empts another, the interrupts are nested. On exception entry the processor automatically saves processor state, which is pushed on to the stack. In parallel with this, the vector corresponding to the interrupt is fetched. Execution of the first instruction of the ISR starts when processor state is saved and the first instruction of the ISR enters the execute stage of the processor pipeline. The state saving is performed over the System bus. The vector fetch is performed over either the System bus or the DCode bus depending on where the vector table is located, see *Vector Table Offset Register* on page 8-20.
Tail-chain	A mechanism used by the processor to speed up interrupt servicing. On completion of an ISR, if there is a pending interrupt of higher priority than the ISR or thread that is being returned to, the stack pop is skipped and control is transferred to the new ISR.
Return	With no pending exceptions or no pending exceptions with higher priority than a stacked ISR, the processor pops the stack and returns to stacked ISR or Thread Mode. On completion of an ISR the processor automatically restores the processor state by popping the stack to the state prior to the interrupt that caused the ISR to be entered. If a new interrupt arrives during the state restoration, and that interrupt is higher priority than the ISR or thread that is being returned to, then the state restoration is abandoned and the new interrupt is handled as a tail-chain.
Late-arriving	A mechanism used by the processor to speed up pre-emption. If a higher priority interrupt arrives during state saving for a previous pre-emption, the processor switches to handling the higher priority interrupt instead and initiates the vector fetch for that interrupt. The state saving is not effected by late arrival because the state saved is the same for both interrupts, and the state saving continues uninterrupted. Late arriving interrupts are managed until the first instruction of the ISR enters the execute stage of the processor pipeline. On return, the normal tail-chaining rules apply.

In the processor exception model, priority determines when and how the processor takes exceptions. You can:

- assign software priority levels to interrupts
- group priorities by splitting priority levels into pre-emption priorities and subpriorities.

5.3.1 Priority levels

The NVIC supports software-assigned priority levels. You can assign a priority level from 0 to 255 to an interrupt by writing to the eight-bit PRI_N field in an Interrupt Priority Register, see *Interrupt Priority Registers* on page 8-15. Hardware priority decreases with increasing interrupt number. Priority level 0 is the highest priority level, and priority level 255 is the lowest. The priority level overrides the hardware priority. For example, if you assign priority level 1 to **INTISR[0]** and priority level 0 to **INTISR[31]**, then **INTISR[31]** has higher priority than **INTISR[0]**.

Note

Software prioritization does not affect Reset, NMI, and Hard Fault. They always have higher priority than the external interrupts.

If you assign the same priority level to two or more interrupts, their hardware priorities determine the order in which the processor activates them. For example, if both **INTISR[0]** and **INTISR[1]** are priority level 1, then **INTISR[0]** has higher priority than **INTISR[1]**.

For more information on the PRI_N fields, see *Interrupt Priority Registers* on page 8-15.

5.3.2 Priority grouping

To increase priority control in systems with large numbers of interrupts, the NVIC supports priority grouping. You can use the PRIGROUP field in the *Application Interrupt and Reset Control Register* on page 8-21 to split the value in every PRI_N field into a pre-emption priority field and a subpriority field. The pre-emption priority group is referred to as the group priority. Where multiple pending exceptions share the same group priority, the sub-priority bit field is then used to resolve the priority within a group. This is referred to as the sub-priority within the group. The combination of the group priority and the sub-priority is referred to generally as the priority. Where two pending exceptions have the same priority, the lower pending exception number has priority over the higher pending exception number. This is consistent with the priority precedence scheme.

Table 5-3 shows how writing to PRIGROUP splits an eight bit PRI_N field into a pre-emption priority field (x) and a subpriority field (y).

Table 5-3 Priority grouping

	Interrupt priority level field, PRI_N[7:0]				
PRIGROUP[2:0]	**Binary point position**	**Pre-emption field**	**Subpriority field**	**Number of pre-emption priorities**	**Number of subpriorities**
b000	bxxxxxxx.y	[7:1]	[0]	128	2
b001	bxxxxxx.yy	[7:2]	[1:0]	64	4
b010	bxxxxx.yyy	[7:3]	[2:0]	32	8
b011	bxxxx.yyyy	[7:4]	[3:0]	16	16
b100	bxxx.yyyyy	[7:5]	[4:0]	8	32
b101	bxx.yyyyyy	[7:6]	[5:0]	4	64
b110	bx.yyyyyyy	[7]	[6:0]	2	128
b111	b.yyyyyyyy	None	[7:0]	0	256

Note

- Table 5-3 shows the priorities for the processor configured with 8 bits of priority.
- For a processor configured with less than eight bits of priority, the lower bits of the register are always 0. For example, if four bits of priority are implemented, **PRI_N[7:4]** sets the priority, and **PRI_N[3:0]** is 4'b0000.

An interrupt can preempt another interrupt in progress only if its pre-emption priority is higher than that of the interrupt in progress.

For more information on priority optimizations, Priority Level grouping, and Priority masking, see the *ARMv7-M Architecture Reference Manual*.

5.4 Privilege and stacks

The processor supports two separate stacks:

Process stack

Thread mode can be configured to use the process stack. Thread mode uses the Main stack out of reset. SP_process is the SP register for the process stack.

Main stack

Handler mode uses the main stack. SP_main is the SP register for the main stack.

Only one stack, the process stack or the main stack, is visible at any time. After pushing the eight registers, the ISR uses the main stack, and all subsequent interrupt pre-emptions use the main stack. The rules for which stack saves context are:

- Thread mode uses either the main stack or the process stack, depending on the value of the CONTROL bit [1], which can be accessed by MSR or MRS. This bit can also be set using appropriate EXC_RETURN values when exiting an ISR. An exception that pre-empts a user thread saves the context of the user thread on the stack that the Thread Mode is using.
- All exceptions use the main stack for their own local variables.

Using the process stack for the Thread mode and the main stack for exceptions supports *Operating System* (OS) scheduling. To reschedule, the kernel has to save only the eight registers not pushed by hardware, r4-r11, and copy SP_process into the *Thread Control Block* (TCB). If the processor saved the context on the main stack, the kernel would have to copy the 16 registers to the TCB.

Note

MSR/MRS instructions have visibility of both stacks.

5.4.1 Stacks

The stack model is independent of privileged mode. That is, Thread mode can use the process or main stack and be in user or privileged mode. All four combinations of stack and privilege are possible. For a basic protected thread model, the user threads run in Thread mode using the Process stack, and the kernel and the interrupts run privileged using the Main stack.

Note

Privilege alone does not prevent corruption of stacks, whether malicious or accidental. A memory protection scheme of one form or another is required to isolate the user code. That is, the user code must be prevented from writing to memory it does not own, including other stacks.

5.4.2 Privilege

Privilege controls access rights, and is decoupled from all other concepts in ARMv7-M. Code can be privileged, with full access rights, or unprivileged/user, with limited access rights. Access rights affect ability to:

- Use or not use certain instructions such as MSR fields.
- Access *System Control Space* (SCS) registers.
- Use certain coprocessors or coprocessor registers.
- Access memory or peripherals, based on system design. The processor tells the system whether the code making an access is privileged and so the system can enforce restrictions on non-privileged access.
- Access rules to memory locations based on an MPU. When fitted with an MPU, the access restrictions can control what memory can be read, written, and executed.

Only Thread mode can be unprivileged. All exceptions are privileged.

5.5 Pre-emption

The following sections describe the behavior of the processor when it takes an exception:

- *Stacking*
- *Late-arriving* on page 5-14
- *Tail-chaining* on page 5-13.

5.5.1 Stacking

When the processor invokes an exception, it automatically pushes the following eight registers to the stack (SP) in the following order:

- PC
- xPSR
- r0-r3
- r12
- LR.

The SP is decremented by eight words by the completion of the stack push. Figure 5-1 shows the contents of the stack after an exception pre-empts the current program flow.

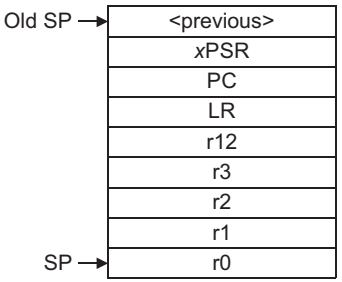


Figure 5-1 Stack contents after a pre-emption

Note

Figure 5-1 shows the order on the stack.

After returning from the ISR, the processor automatically pops the eight registers from the stack. Interrupt return is passed as a data field in the LR, so ISR functions can be normal C/C++ functions, and do not require a veneer.

Table 5-4 describes the steps that the Cortex-M3 processor takes before it enters an ISR.

Table 5-4 Exception entry steps

Action	Restartable?	Description
Push eight registers[a]	No.	Pushes *x*PSR, PC, r0, r1, r2, r3, r12, and LR on selected stack.
Read vector table	Yes. Late-arriving exception can cause restart.	Reads vector table from memory based on table base + (exception number 4). Read on the ICode bus can be done simultaneously with register pushes on the DCode bus.
Read SP from vector table	No.	On Reset only, updates SP to top of stack from vector table. Other exceptions do not modify SP except to select stack, push, and pop.
Update PC	No.	Updates PC with vector table read location. Late-arriving exceptions cannot be processed until the first instruction starts to execute.
Load pipeline	Yes. Pre-emption reloads pipeline from new vector table read.	Loads instructions from location pointed to by vector table. This is done in parallel with register push.
Update LR	No.	LR is set to EXC_RETURN to exit from exception. EXC_RETURN is one of 16 values as defined in *ARMv7-M Architecture Reference Manual.*

a. When tail-chaining, this step is skipped.

Figure 5-2 on page 5-12 shows an example of exception entry timing.

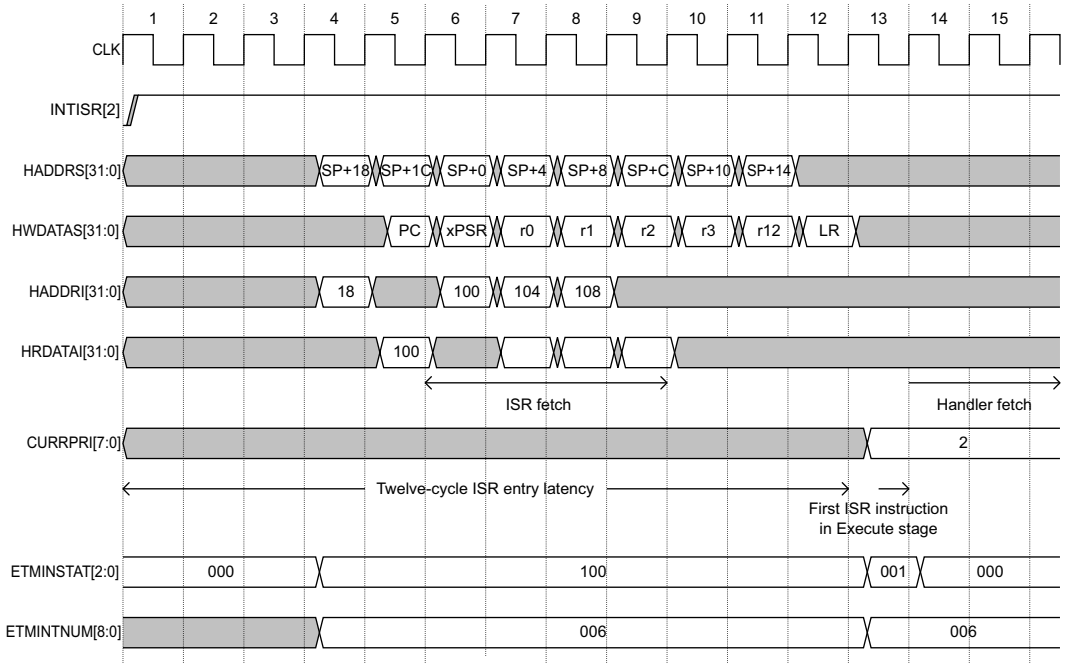


Figure 5-2 Exception entry timing

The NVIC indicates to the processor core, in the cycle after **INTISR[2]** was received, that an interrupt has been received, and the processor initiates the stack push and vector fetch in the following cycle.

When the stack push has completed, the first instruction of the ISR enters the execute stage of the pipeline. In the cycle that the ISR enters execute:

- **ETMINSTAT[2:0]** indicates that the ISR has been entered (3'b001). This is a 1 cycle pulse.
- **CURRPRI[7:0]** indicates the priority of the active interrupt. **CURRPRI** remains asserted throughout the duration of the ISR. **CURRPRI** becomes valid when **ETMINTSTAT** indicates that the ISR has been entered (3'b001).
- **ETMINTNUM[8:0]** indicates the number of the active interrupt. **ETMINTNUM** remains asserted throughout the duration of the ISR. **ETMINTNUM** becomes valid when **ETMINTSTAT** indicates that the ISR has been entered (3'b001). Prior to that it indicates which ISR is being fetched.

Figure 5-2 shows that there is a 12-cycle latency from asserting the interrupt to the first instruction of the ISR executing.

5.6 Tail-chaining

Tail-chaining is back-to-back processing of exceptions without the overhead of state saving and restoration between interrupts. The processor skips the pop of eight registers and push of eight registers when exiting one ISR and entering another because this has no effect on the stack contents.

The processor tail-chains if a pending interrupt has higher priority than all stacked exceptions.

Figure 5-3 shows an example of tail-chaining. If a pending interrupt has higher priority than the highest-priority stacked exception, the stack push or pop is omitted, and the processor immediately fetches the vector for the pending interrupt. The ISR that is tail-chained into starts execution six cycles after exiting the previous ISR.

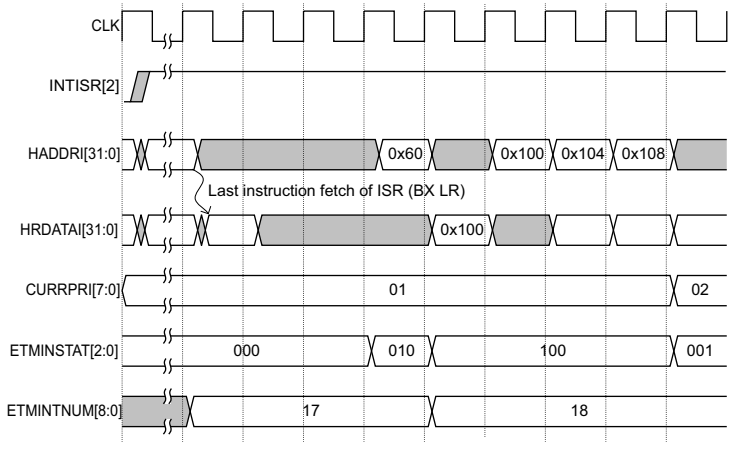


Figure 5-3 Tail-chaining timing

On return from the last ISR, **INTISR[2]** is higher priority than any stacked ISR, or other pended interrupt, and so the processor tailchains to the ISR corresponding to **INTISR[2]**. In the cycle that the ISR for **INTISR[2]** enters execute:

- **ETMINSTAT[2:0]** indicates that the ISR has been entered (3'b001). This is a 1 cycle pulse.
- **CURRPRI[7:0]** indicates the priority of the active interrupt. **CURRPRI** remains asserted throughout the duration of the ISR.
- **ETMINTNUM[8:0]** indicates the number of the active interrupt. **ETMINTNUM** remains asserted throughout the duration of the ISR.

Figure 5-3 shows that there is a 6-cycle latency from returning from the last ISR to executing the new ISR.

5.7 Late-arriving

A late-arriving interrupt can preempt a previous interrupt if the first instruction of the previous ISR has not entered the Execute stage, and the late-arriving interrupt has a higher priority than the previous interrupt.

A late-arriving interrupt causes a new vector address fetch and ISR prefetch. State saving is not performed for the late-arriving interrupt because it has already been performed for the initial interrupt and so does not have to be repeated.

Figure 5-4 shows an example of late-arriving interrupts.

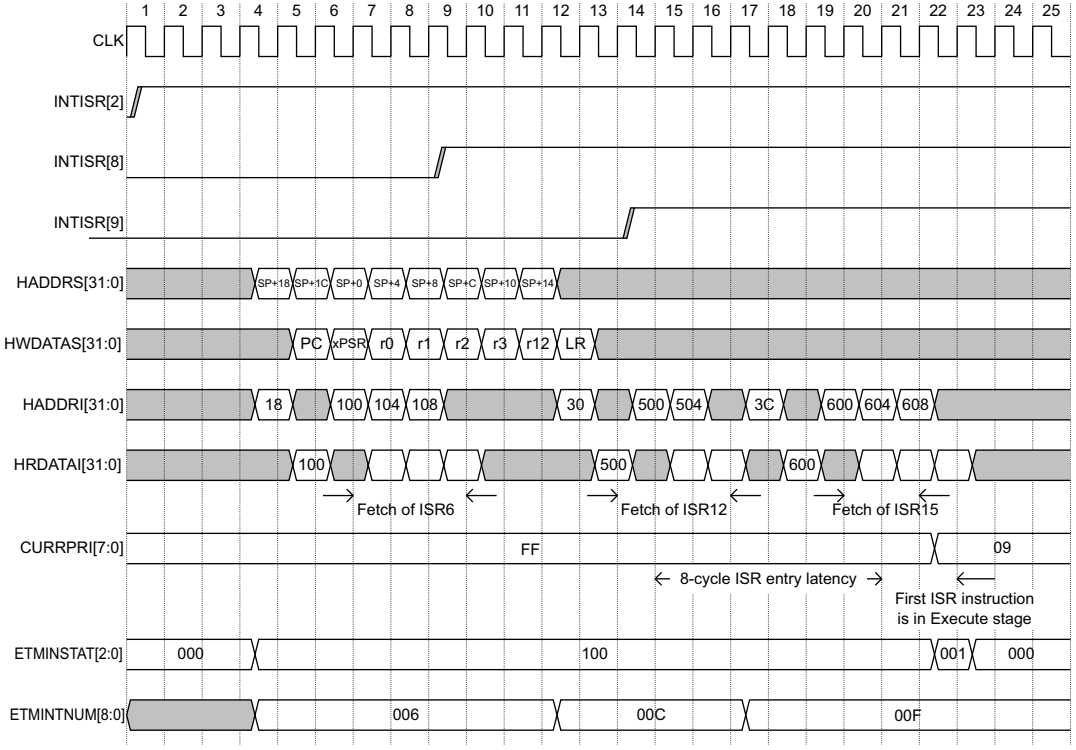


Figure 5-4 Late-arriving exception timing

In Figure 5-4, **INTISR[8]** pre-empts **INTISR[2]**. The state saving for **INTISR[2]** is already done and does not have to be repeated. Figure 5-4 shows the latest point at which **INTISR[8]** can preempt before the first instruction of the ISR for **INTISR[2]** enters Execute stage. A higher priority interrupt after that point is managed as a pre-emption.

Figure 5-4 on page 5-14 shows the latest point at which **INTISR[9]** can preempt before the first instruction of the ISR for **INTISR[8]** enters Execute stage. The ISR fetch for **INTISR[8]** is aborted when **INTISR[9]** is received, and the processor then initiates the vector fetch for **INTISR[9]**. A higher priority interrupt after that point is managed as pre-emption.

In the cycle that the ISR for **INTISR[9]** enters execute:

- **ETMINSTAT[2:0]** indicates that the ISR has been entered (3'b001). This is a one-cycle pulse.
- **CURRPRI[7:0]** indicates the priority of the active interrupt. **CURRPRI** remains asserted throughout the duration of the ISR.
- **ETMINTNUM[8:0]** indicates the number of the active interrupt. **ETMINTNUM** remains asserted throughout the duration of the ISR.

5.8 Exit

The last instruction of an ISR loads the PC with value `0xFFFFFFFX`, which was LR on exception entry. This indicates to the processor that the ISR is complete, and the processor initiates the exception exit sequence. See *Returning the processor from an ISR* on page 5-17 for the instructions that you can use to return the processor from an ISR.

5.8.1 Exception exit

When returning from an exception, the processor is either:

- tail-chaining to a pending exception if the pending exception is higher priority than all stacked exceptions
- returning to the last stacked ISR if there are no pending exceptions or if the highest priority stacked exception is higher priority than the highest priority pending exception
- returning to the Thread mode if there are no pending or stacked exceptions.

Table 5-5 describes the postamble sequence.

Table 5-5 Exception exit steps

Action	Description
Pop eight registers	Pops PC, xPSR, r0, r1, r2, r3, r12 and LR from stack selected by EXC_RETURN and adjusts SP, if not preempted.
Load current active interrupt number[a]	Loads current active interrupt number from bits [8:0] of stacked IPSR word. The processor uses this to track which exception to return to and to clear the activation bit on return. When bits [8:0] are zero, the processor returns to Thread Mode.
Select SP	If returning to an exception, SP is SP_main. If returning to Thread Mode, SP can be SP_main or SP_process.

a. Because of dynamic priority changes, the NVIC uses interrupt numbers instead of interrupt priorities to determine which ISR is current.

Figure 5-5 on page 5-17 shows an example of exception exit timing.

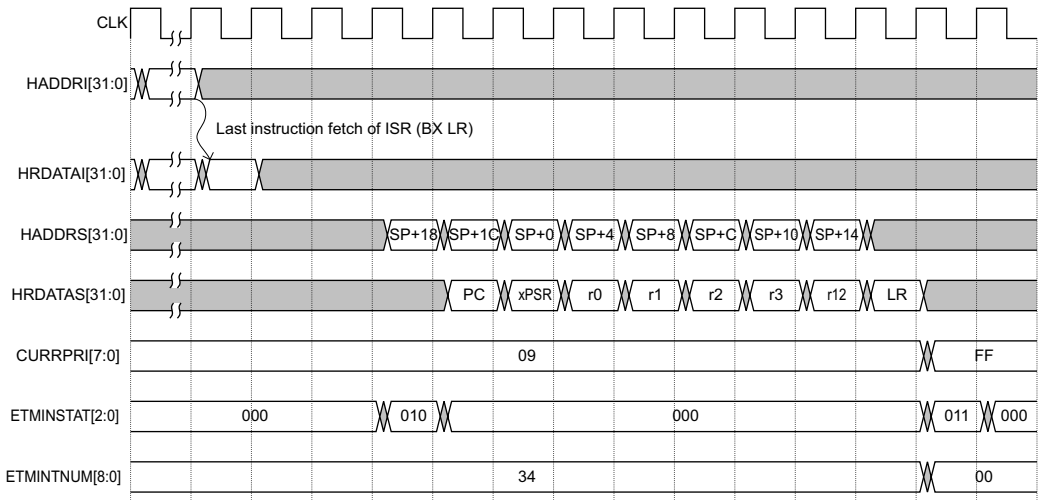


Figure 5-5 Exception exit timing

ETMINSTAT will indicate:

- 3'b010 to show that the ISR has exited. **ETMINTNUM** shows the number of the ISR that exited.
- 3'b011 in the cycle after interrupt exit if a previous stacked ISR is being returned to. **ETMINTNUM** shows the number of the interrupt that is being returned to.

Note

If a higher priority exception occurs during the stack pop, the processor abandons the stack pop, rewinds the stack pointer, and services the exception as a tail-chain case.

5.8.2 Returning the processor from an ISR

Exception returns occur when one of the following instructions loads a value of `0xFFFFFFFX` into the PC:

- POP/LDM which includes loading the PC
- LDR with PC as a destination
- BX with any register.

When used in this way, the value written to the PC is intercepted and is referred to as the **EXC_RETURN** value. **EXC_RETURN[3:0]** provides return information as defined in Table 5-6.

Table 5-6 Exception return behavior

EXC_RETURN[3:0]	
0bXXX0	RESERVED
0b0001	Return to Handler Mode Exception return gets state from the Main stack On return execution uses the Main Stack
0b0011	RESERVED
0b01X1	RESERVED
0b1001	Return to Thread Mode Exception return gets state from the Main stack On return execution uses the Main Stack
0b1101	Return to Thread Mode Exception return gets state from the Process stack On return execution uses the Process Stack
0b1X11	RESERVED

RESERVED entries in this table result in a chained exception to a UsageFault.

If an EXC_RETURN value is loaded into the PC when in Thread mode, or from the vector table, or by any other instruction, the value is treated as an address, not as a special value. This address range is defined to have Execute Never (XN) permissions, and will result in a MemManage fault.

5.9 Resets

The NVIC is reset at the same time as the core and controls the release of reset into the core. As a result, the behavior of reset is predictable. Table 5-7 shows the reset behavior.

Table 5-7 Reset actions

Action	Description
NVIC resets, holds core in reset	NVIC clears most of its registers. The processor is in Thread mode, priority is Privileged, and the stack is set to Main.
NVIC releases core from reset	NVIC releases core from reset.
Core sets Stack	Core reads the start SP, SP_main, from vector-table offset 0.
Core sets PC and LR	Core reads the start PC from vector-table offset. LR is set to `0xFFFFFFFF`.
Reset routine runs	NVIC has interrupts disabled, and NMI and Hard Fault are not disabled.

For more information about resets, see Chapter 6 *Clocking and Resets*.

5.9.1 Vector Table and Reset

The vector table at location 0 is only required to have 4 values:

- stack top address
- reset routine location
- NMI ISR location
- HardFault ISR location.

When interrupts are enabled the vector table, regardless of location, points to all mask-enabled exceptions. Also, the SVCall ISR location is populated if the SVC instruction is used.

An example of a full vector table:

```
unsigned int stack_base[STACK_SIZE];
void ResetISR(void);
void NmiISR(void);
…
ISR_VECTOR_TABLE vector_table_at_0
{
  stack_base + sizeof(stack_base),
  ResetISR,
  NmiSR,
FaultISR,
0,          // Populate if using MemManage (MPU)
```

```
0,          // Populate if using Bus fault
0,          // Populate if using Usage Fault
0, 0, 0, 0, // reserved slots
SVCallISR,
0,          // Populate if using a debug monitor
0,          // Reserved
0,          // Populate if using pendable service request
0,          // Populate if using SysTick
// external interrupts start here
  Timer1ISR,
  GpioInISR
  GpioOutISR,
  I2CIsr
};
```

5.9.2 Intended boot up sequence

A normal reset routine follows the steps shown in Table 5-8. A C/C++ runtime may perform the first three steps and then call `main()`.

Table 5-8 Reset boot-up behavior

Action	Description
Initialize variables	Any global/static variables must be setup. This includes zeroing out BSS (initialized variables), and copying initial values from ROM to RAM for non-constant variables.
[Setup stacks]	If more than one stack will be used, the other banked SPs must be initialized. The current SP may also be changed to Process from Main.
Initialize any runtime	Optionally make calls to C/C++ runtime init code to allow use of heap, floating point, or other features. This is normally done by `__main` from the C/C++ library.
[Initialize any peripherals]	Setup peripherals before interrupts are enabled. This can call to setup each peripheral to be used in the application.
[Switch ISR vector table]	Optionally change vector table from Code area, @0, to a location in SRAM. This is only done to optimize performance or allow dynamic changes.
[Setup Configurable Faults]	Enable Configurable faults and set their priorities.
Setup interrupts	Setup priority levels and masks.

Table 5-8 Reset boot-up behavior

Action	Description
Enable interrupts	Enable interrupts. Enable the interrupt processing in the NVIC. If it is not desirable to have these occur as they are being enabled. If more than 32 interrupts, it will take more than one Set-Enable register. PRIMASK can be used through CPS or MSR to mask interrupts until ready.
[Change Privilege]	[Change Privilege]. The Thread Mode privilege may be changed to user if required. This should normally be handled by invoking the SVCall handler.
"Loop"	If sleep-on-exit is enabled, control will never return after the first interrupt/exception is taken. If sleep-on-exit is selectively enabled/disabled, this loop can handle cleanup and executive tasks. If sleep-on-exit is not used, the loop can do what it wants and can use WFI (sleep-now) when wanted.

Example of reset routine

The reset routine is responsible for starting up the application and then enabling interrupts. There are three methods for involving the reset ISR after interrupt processing is performed. This is called the "main loop" part of the Reset ISR and the three examples are shown below.

Example 5-1 Reset routine with pure sleep on exit (Reset routine does no main loop work)

```
void reset()
{
    // do setup work (initialize variables, initialize runtime if wanted,
    setup peripherals, etc)
    nvic[INT_ENA] = 1;    // enable interrupts
    nvic_regs[NV_SLEEP] |= NVSLEEP_ON_EXIT; // will not normally come back after
                                                1st exception
    while (1)
    wfi();
}
```

Example 5-2 Reset routine with selected Sleep model using WFI

```
void reset()
{
    extern volatile unsigned exc_req;
    // do setup work (initialize variables, initialize runtime if wanted,
```

```
    setup peripherals, etc)
    nvic[INT_ENA] = 1;    // enable interrupts
    while (1)
    {
        // do some work for (exc_req = FALSE; exc_req == FALSE; )
        wfi();  // sleep now - wait for interrupt
    // do some post exception checking/cleanup
    }
}
```

Example 5-3 Reset routine with selected Sleep on exit cancelled by ISRs that require attention

```
void reset()
{
    // do setup work (initialize variables, initialize runtime if wanted,
    setup peripherals, etc)
    nvic[INT_ENA] = 1;    // enable interrupts
    while (1)
    {
        // We are slept until an exception clears sleep on exit state so that we
        can post-process/cleanup.
        nvic_regs[NV_SLEEP] |= NVSLEEP_ON_EXIT;
        while (nvic_regs[NV_SLEEP] & NVSLEEP_ON_EXIT)
        wfi();  // sleep now - wait for interrupt which clears
        // do some post exception checking/cleanup
    }
}
```

Note

An executive does not need to live in the Reset routine because priority level changes can be enacted from an ISR activation. This ensures faster response to changing loads, and uses priority boosting, to solve priority inversions, to ensure fine grain support. For RTOS models using threads and privilege, the Thread Mode is used for the user code.

5.10 Exception control transfer

The processor transfers control to an ISR following the rules shown in Table 5-9.

Table 5-9 Transferring to exception processing

Processor activity at assertion of exception	Transfer to exception processing
Non-memory instruction	Takes exception on completion of cycle, before next instruction.
Load/Store single	Completes or abandons depending on bus status. Takes exception on next cycle, depending on bus wait states.
Load/store multiple	Completes or abandons current register and sets continuation counter into EPSR. Takes exception on next cycle, depending on bus permission and *Interruptible-Continuable Instruction* (ICI) rules. For more information on ICI rules see the *ARMv7-M Architecture Reference Manual*.
Exception entry	This is a late-arriving exception. If it has higher priority than the exception being entered, then the processor cancels the exception entry actions and takes the late-arriving exception. Late arriving results in a decision change (vector table) at interrupt processing time. When you enter a new handler, that is the first ISR instruction, normal pre-emption rules apply, and it is no longer classed as a late-arrival.
Tail-chaining	This is a late-arriving exception. If it has higher priority than the one being tail-chained, the processor cancels the preamble and takes the late-arriving exception.
Exception postamble	If the new exception has higher priority than the stacked exception to which the processor is returning, the processor tail-chains the new exception.

5.11 Setting up multiple stacks

To implement multiple stacks, the application must take these actions:

- use the MSR instruction to set up the Process_SP register
- if using an MPU, protect the stacks appropriately
- Associate interrupts with the stack
- Initialize the stack and privilege of the Thread mode.

If the privilege of Thread mode is changed from Privileged to User, only another ISR, such as SVCall, can change the privilege back from User to Privileged.

The stack in Thread mode can be changed from Main to Process or from Process to Main, but doing so affects its access to the thread's own local variables. It is better to have another ISR change the stack used in Thread mode. Here is an example boot sequence:

1. Call setup routine to:
   a. Set up other stacks using MSR.
   b. Enable the MPU to support base regions, if any.
   c. Invoke all boot routines.
   d. Return from setup routine.
2. Change Thread mode to unprivileged.
3. Use SVC to invoke the kernel. Then the kernel:
   a. Starts threads.
   b. Uses MRS to read the SP for the current user thread and save it in its TCB.
   c. Uses MSR to set the SP for the next thread. This is usually SP_process.
   d. Sets up the MPU for the newly current thread, if necessary.
   e. Returns into the newly current thread.

Example 5-4 shows a modification to the EXC_RETURN value in the ISR to return using PSP.

Example 5-4 Modification to the EXC_RETURN value in the ISR

```
; First time use of PSP, run from a Handler with RETTOBASE == 1

  LDR r0, PSPValue       ; acquire value for new Process stack
  MSR PSP, r0            ; set Process stack value
  ORR lr, lr, #4         ; change EXC_RETURN for return on PSP
  BX lr                  ; return from Handler to Thread
```

Example 5-5 shows how to implement a simple context switcher after the switch to Thread on PSP.

Example 5-5 Implement a simple context switcher

```
; Example Context Switch (Assumes Thread is already on PSP)

   MRS r12, PSP                ; Recover PSP into R12
   STMDB r12!, {r4-r11}        ; Push non-stack registers
   LDR r0, =OldPSPValue        ; Get pointer to old Thread Control Block
   STR r12, [r0]               ; Store SP into Thread Control Block
   LDR r0, =NewPSPValue        ; Get pointer to new Thread Control Block
   LDR r12, [r0]               ; Acquire new Process SP
   LDMIA r12!, {r4-r11}        ; Restore non-stacked registers
   MSR PSP, r12                ; Set PSP to R12
   BX lr                       ; Return back to Thread
```

Note

For Example 5-4 on page 5-24 and Example 5-5 the only time the decision to move Thread from MSP to PSP can be made, or the non-stacked registers can be guaranteed not to have been modified by a stacked Handler, is when there is only one ISR/Handler active.

5.12 Abort model

Four events can generate a fault:

- an instruction fetch or vector table load bus error
- a data access bus error
- internally-detected error such as an undefined instruction or an attempt to change state with a BX instruction. Fault status registers in the NVIC indicate the causes of the Faults
- MPU fault as a result of privilege violation or unmanaged region.

There are two kinds of fault handler:

- the fixed-priority Hard Fault
- the settable-priority local faults.

5.12.1 Hard Fault

Only Reset and NMI can preempt the fixed priority Hard Fault. A Hard Fault can preempt any other exception other than Reset NMI or another Hard Fault.

Note

Code that uses FAULTMASK acts as a Hard Fault and so follows the same rules as a Hard Fault.

Secondary bus faults do not escalate because a preempting fault of the same type cannot preempt itself. This means that if a corrupted stack causes a fault, the fault handler still executes even though the stack pushes for it failed. The fault handler can operate, but the stack contents are corrupted.

5.12.2 Local faults and escalation

Local faults are categorized according to their cause. See Table 5-10 on page 5-27. When enabled, local fault handlers process all normal faults. However, a local fault escalates to a Hard Fault when:

- a local fault handler causes the same kind of fault as the one it is servicing
- a local fault handler causes a fault with the same or higher priority
- an exception handler causes a fault with the same or higher priority
- the local fault is not enabled.

Table 5-10 lists the local faults.

Table 5-10 Faults

Fault	Bit Name	Handler	Notes	Trap enable bit
Reset	Reset cause	Reset	Any form of reset	RESETVCATCH
Vector Read error	VECTTBL	HardFault	Bus error returned when reading the vector table entry,	INTERR
uCode stack push error	STKERR	BusFault	Failure when saving context using hardware - bus error returned	INTERR
uCode stack push error	MSTKERR	MemManage	Failure when saving context using hardware - MPU access violation.	INTERR
uCode stack pop error	UNSTKERR	BusFault	Failure when restoring context using hardware - bus error returned	INTERR
uCode stack pop error	MUNSKERR	MemManage	Failure when restoring context using hardware - MPU access violation.	INTERR
Escalated to Hard Fault	FORCED	HardFault	Fault occurred and handler is equal or higher priority than current, including fault within fault when priority does not enable, or Configurable fault disabled. Includes SVC, BKPT and other kinds of faults.	HARDERR
MPU mismatch	DACCVIOL	MemManage	Violation or fault on MPU as a result of Data access.	MMERR
MPU mismatch	IACCVIOL	MemManage	Violation/fault on MPU as a result of instruction address.	MMERR
Pre-fetch error	IBUSERR	BusFault	Bus error returned because of instruction fetch. Faults only if makes it to execute. Branch shadow can fault and be ignored.	BUSERR
Precise Data bus error	PRECISEERR	BusFault	Bus error returned because of data access, and was precise, points to instruction.	BUSERR

Table 5-10 Faults (continued)

Fault	Bit Name	Handler	Notes	Trap enable bit
Imprecise Data bus error	IMPRECISERR	BusFault	Late bus error because of data access. Exact instruction is no longer known. This is pended and not synchronous. It does not cause FORCED.	BUSERR
No Coprocessor	NOCP	UsageFault	Truly does not exist, or not present bit	NOCPERR
Undefined Instruction	UNDEFINSTR	UsageFault	Unknown instruction	STATERR
Attempt to execute an instruction when in an invalid ISA state. For example, not Thumb	INVSTATE	UsageFault	Attempt to execute in an invalid EPSR state. For example, after a BX type instruction has changed state. This includes states after return from exception including inter-working states.	STATERR
Return to PC=EXC_RETURN when not enabled or with invalid magic number	INVPC	UsageFault	Illegal exit, caused either by an illegal EXC_RETURN value, an EXC_RETURN and stacked EPSR value mismatch, or an exit while the current EPSR is not contained in the list of currently active exceptions.	STATERR
Illegal unaligned load or store	UNALIGNED	UsageFault	This occurs when any load-store multiple instruction attempts to access a non-word aligned location. It can be enabled to occur for any load-store that is unaligned to its size using the UNALIGN_TRP bit.	CHKERR
Divide By 0	DIVBYZERO	UsageFault	This can be enabled to occur when SDIV or UDIV is executed with a divisor of 0, and the DIV_0_TRP bit is set.	CHKERR
SVC	-	SVCall	System request (Service Call).	-

Table 5-11 shows debug faults.

Table 5-11 Debug faults

Fault	Flag	Notes	Trap enable bit
Internal halt request	HALTED	NVIC request from step, core halt, etc.	-
Breakpoint	BKPT	SW breakpoint from patched instruction or FPB	-
Watchpoint	DWTTRAP	Watchpoint match in DWT	-
Divide by zero	DIVBYZERO	Divide by zero when enabled for trap	-
Unaligned access	UNALIGNED	Unaligned access when enabled for trap	-
External	EXTERNAL	EDBGRQ line asserted	-

5.12.3 Fault status registers and fault address registers

Each fault has a fault status register with a flag for that fault.

There are:

- three configurable fault status registers that correspond to the three configurable fault handlers
- one hard fault status register
- one debug fault status register.

Depending on the cause, one of the five status registers has a bit set.

There are two Fault Address Registers (FAR):
- Bus Fault Address Register (BFAR)
- Memory Fault Address Register (MFAR).

A flag in the corresponding fault status register indicates when the address in the fault address register is valid.

Note

BFAR and MMFAR are the same physical register. Because of this the BFARVALID and MMFARVALID bits are mutually exclusive.

Table 5-12 on page 5-30 shows the fault status registers and two fault address registers

Table 5-12 Fault status and fault address registers

Status Register name	Handler	Address Register name	Description
HFSR	Hard Fault	-	Escalation and Special
MMSR	Mem Manage	MMAR	MPU faults
BFSR	Bus Fault	BFAR	Bus faults
UFSR	Usage Fault	-	Usage fault
DFSR	Debug Monitor or Halt	-	Debug traps

5.13 Activation levels

When no exceptions are active, the processor is in Thread mode. When an ISR or fault handler is active, the processor enters Handler mode. Table 5-13 lists the privilege and stacks of the activation levels.

Table 5-13 Privilege and stack of different activation levels

Active exception	Activation level	Privilege	Stack
None	Thread mode	Privileged or user	Main or process
ISR active	Asynchronous pre-emption level	Privileged	Main
Fault handler active	Synchronous pre-emption level	Privileged	Main
Reset	Thread Mode	Privileged	Main

Table 5-14 summarizes the transition rules for all exception types and how they relate to the access rules and stack model.

Table 5-14 Exception transitions

Active Exception	Triggering event	Transition type	Privilege	Stack
Reset	Reset signal	Thread	Privileged or user	Main or process
ISR[a] or NMI[b]	Set-pending software instruction or hardware signal	Asynchronous pre-emption	Privileged	Main
Fault:				
Hard fault	Escalation	Synchronous pre-emption	Privileged	Main
Bus fault	Memory access error			
No CP[c] fault	Absent CP access			
Undefined instruction fault	Undefined instruction			
Debug monitor	Debug event when halting not enabled	Synchronous	Privileged	Main
SVC[d]	SVC instruction			
External interrupt				

a. Interrupt service routine.
b. Nonmaskable interrupt.
c. Coprocessor.
d. Software interrupt.

Table 5-15 shows exception subtype transitions.

Table 5-15 Exception subtype transitions

Intended activation subtype	Triggering event	Activation	Priority effect
Thread	Reset signal	Asynchronous	Immediate, thread is lowest
ISR/NMI	HW signal or set-pend	Asynchronous	Preempt or tail-chain according to priority
Monitor	Debug event[a]	Synchronous	If priority less than or equal to current, hard fault
SVCall	SVC instruction	Synchronous	If priority less than or equal to current, hard fault
PendSV	Software pend request	Chain	Preempt or tail-chain according to priority
UsageFault	Undefined instruction	Synchronous	If priority greater than or equal to current, hard fault
NoCpFault	Access to absent CP	Synchronous	If priority greater than or equal to current, hard fault
BusFault	Memory access error	Synchronous	If priority greater than or equal to current, hard fault
MemManage	MPU mismatch	Synchronous	If priority greater than or equal to current, hard fault
HardFault	Escalation	Synchronous	Higher than all bus NMI
FaultEscalate	Escalate request from Configurable fault handler	Chain	Boosts priority of local handler to same as hard fault so it can return and chain to Configurable Fault handler

a. While halting not enabled.

5.14 Flowcharts

This section summarizes interrupt flow with:

- *Interrupt handling*
- *Pre-emption* on page 5-34
- *Return* on page 5-34.

5.14.1 Interrupt handling

Figure 5-6 shows how instructions execute until pre-empted by a higher-priority interrupt.

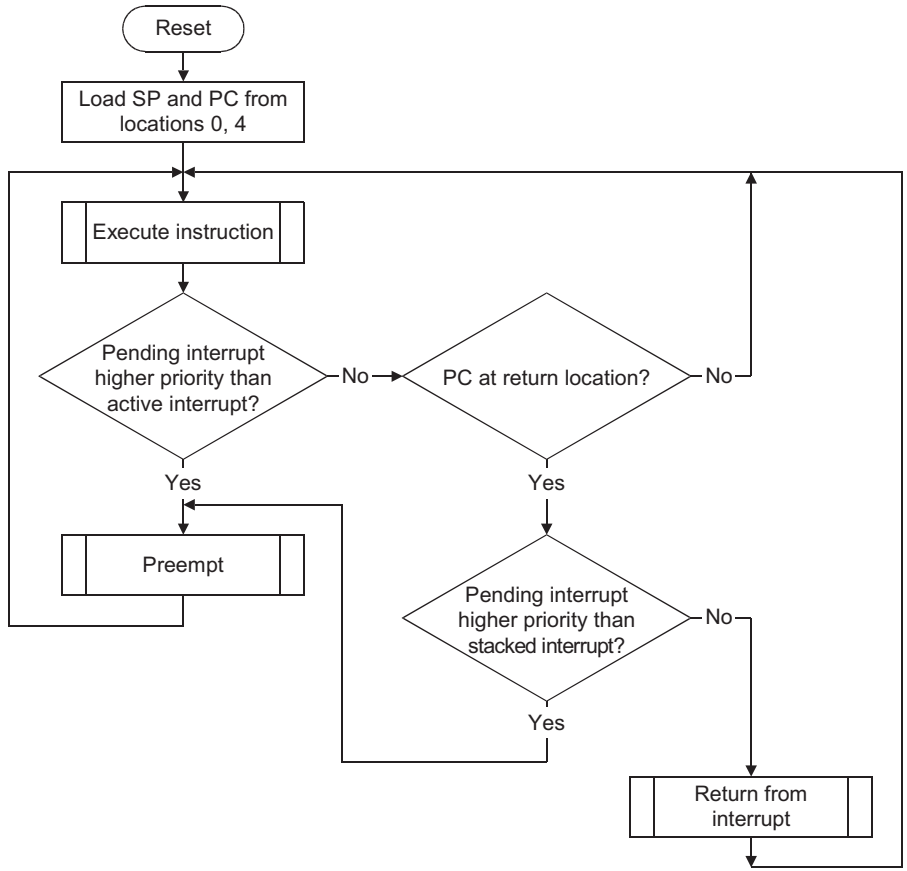


Figure 5-6 Interrupt handling flowchart

5.14.2 Pre-emption

Figure 5-7 shows what happens when an exception pre-empts the current ISR.

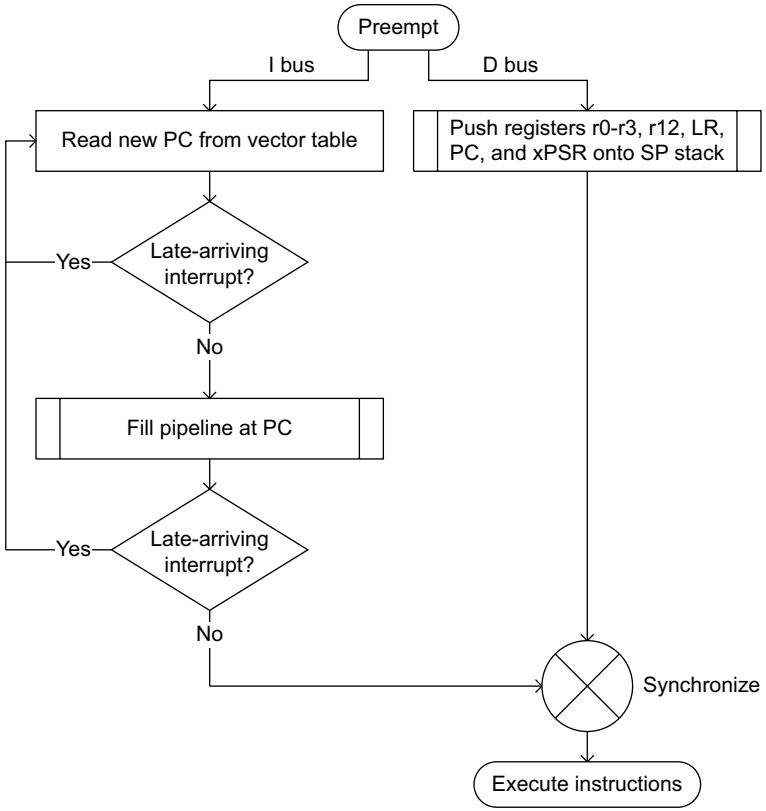


Figure 5-7 Pre-emption flowchart

5.14.3 Return

Figure 5-8 on page 5-35 shows how the processor restores the stacked ISR or tail-chains to a late-arriving interrupt with higher priority than the stacked ISR.

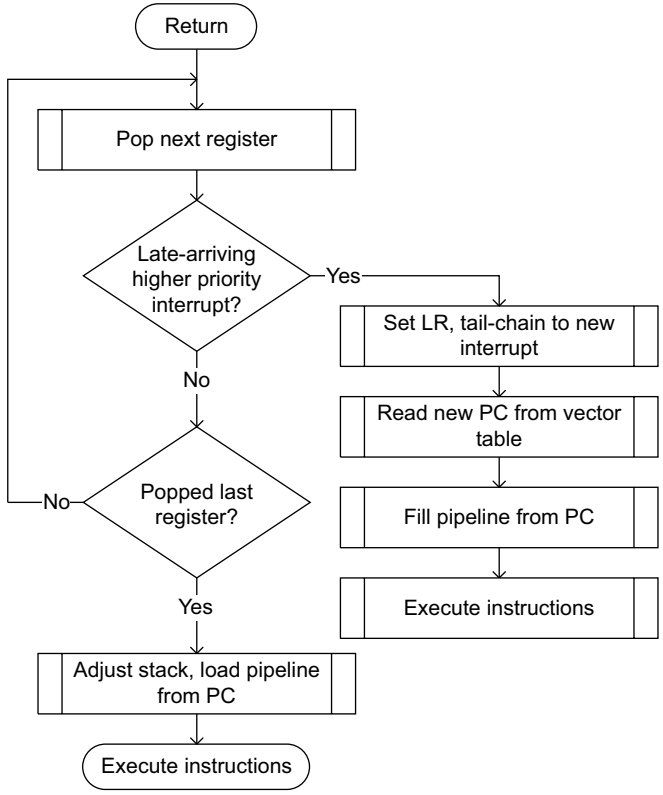


Figure 5-8 Return from interrupt flowchart

Chapter 6
Clocking and Resets

This chapter describes the processor clocking and resets. It contains the following sections:

- *Cortex-M3 clocking* on page 6-2
- *Cortex-M3 resets* on page 6-4
- *Cortex-M3 reset modes* on page 6-5.

6.1 Cortex-M3 clocking

The processor has three functional clock inputs. These are described in Table 6-1.

Table 6-1 Cortex-M3 processor clocks

Clock	Domain	Description
FCLK	Processor	Free running processor clock, used for sampling interrupts and clocking debug blocks. FCLK ensures that interrupts can be sampled, and sleep events can be traced, while the processor is sleeping.
HCLK	Processor	Processor clock
DAPCLK	Processor	Debug port (AHB-AP) clock

FCLK and **HCLK** are synchronous to each other. **FCLK** is a free running version of **HCLK**. For more information, see Chapter 7 *Power Management*. **FCLK** and **HCLK** must be balanced with respect to each other, with equal latencies into Cortex-M3.

The processor is integrated with components for debug and trace. Your macrocell may contain some, or all, of the clocks shown in Table 6-2.

Table 6-2 Cortex-M3 macrocell clocks

Clock	Domain	Description
SWCLK	SW-DP	Serial wire clock
TRACECLKIN	TPIU	Clocks the output of the TPIU
TCK	JTAG-DP	TAP clock

SWCLK is the serial wire clock. It clocks the SWDIN input to the *Serial Wire Debug Port* (SW-DP). It is asynchronous to all other clocks.

TCK is the *Trace Access Port* (TAP) clock. It clocks the JTAG-DP TAP. It is asynchronous to all other clocks.

TRACECLKIN is the reference clock for the *Trace Port Interface Unit* (TPIU). It is asynchronous to the other clocks.

Note

TCK, **SWCLK** and **TRACECLKIN** only have to be driven if your implementation contains JTAG-DP, SW-DP and TPIU blocks respectively. Otherwise, the clock inputs must be tied off.

Note

Cortex-M3 also contains a **STCLK** input. This port is not a clock. It is a reference input for the SysTick counter, and it must be less than half the frequency of **FCLK**. **STCLK** is synchronized internally by the processor to **FCLK**.

6.2 Cortex-M3 resets

The processor has three reset inputs. These are described in Table 6-3.

Table 6-3 Reset inputs

Reset input	Description
PORESETn	Resets the entire processor system with the exception of JTAG-DP
SYSRESETn	Resets the entire processor system with the exception of debug logic in the NVIC, FPB, DWT, ITM, and AHB-AP
nTRST	JTAG-DP reset

Note

nTRST resets JTAG-DP. If your implementation doesn't contain JTAG-DP, this reset must be tied off.

6.3 Cortex-M3 reset modes

The reset signals present in the processor design enable you to reset different parts of the design independently. Table 6-4 shows the reset signals, and the combinations and possible applications that you can use them in.

Table 6-4 Reset modes

Reset mode	SYSRESETn	nTRST	PORESETn	Application
Power-on reset	x	0	0	Reset at power up, full system reset. Cold reset.
System reset	0	x	1	Reset of processor core and system components, excluding debug.
JTAG-DP reset	1	0	1	Reset of JTAG-DP logic.

Note

PORESETn resets a superset of the **SYSRESETn** logic.

6.3.1 Power-on reset

Figure 6-1 on page 6-6 shows the reset signals for the macrocell.

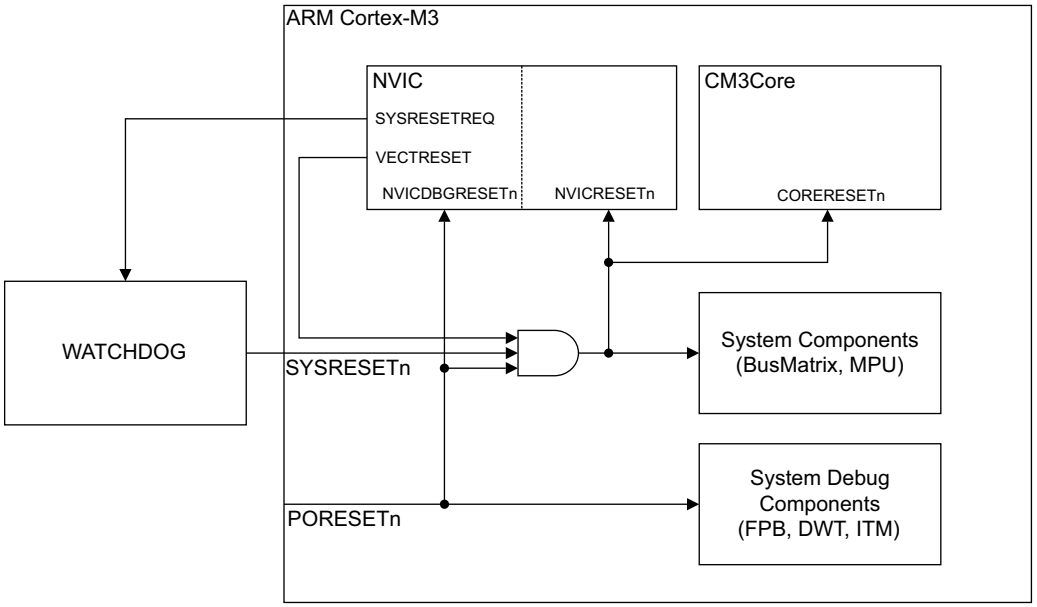


Figure 6-1 Reset signals

You must apply power-on or *cold* reset to the processor when power is first applied to the system. In the case of power-on reset, the falling edge of the reset signal, **PORESETn**, do not have to be synchronous to **HCLK**. Because **PORESETn** is synchronized within the processor, you do not have to synchronize this signal. Figure 6-2 shows the application of power-on reset. Figure 6-3 on page 6-7 shows the reset synchronizers within the processor.

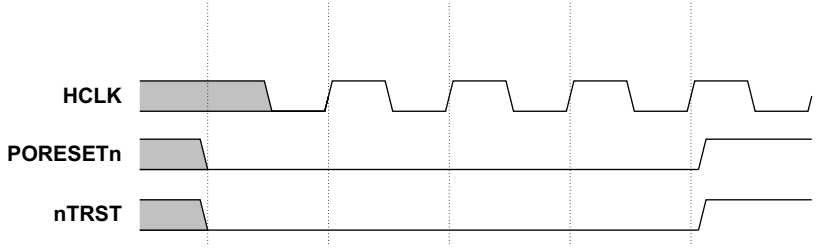


Figure 6-2 Power-on reset

It is recommended that you assert the reset signals for at least three **HCLK** cycles to ensure correct reset behavior. Figure 6-3 on page 6-7 shows the internal reset synchronization.

Note

LOCKUP from the Cortex-M3 system should be considered for inclusion in any external watchdog circuitry when an external debugger is not attached.

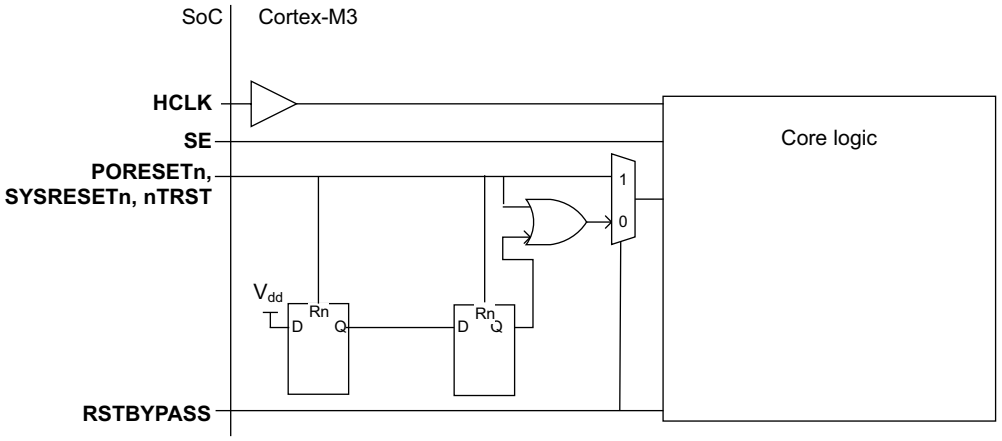


Figure 6-3 Internal reset synchronization

6.3.2 System reset

A system or *warm* reset initializes the majority of the macrocell, excluding the NVIC debug logic, *Flash Patch and Breakpoint* (FPB), *Data Watchpoint and Trigger* (DWT), and *Instruction Trace Macrocell* (ITM). System reset is typically used for resetting a system that has been operating for some time, for example, watchdog reset.

Because the **SYSRESETn** signal is synchronized within the processor, you do not have to synchronize this signal. Figure 6-3 shows the reset synchronization.

Cortex-M3 exports a signal, **SYSRESETREQ**, that is asserted when the **SYSRESETREQ** bit of the Application Interrupt and Reset Control Register is set. This can be used, for example, as an input to a watchdog timer as shown in Figure 6-1 on page 6-6.

6.3.3 JTAG-DP reset

nTRST reset initializes the state of the JTAG-DP controller. **nTRST** reset is typically used by the RealView™ ICE module for hot-plug connection of a debugger to a system.

nTRST enables initialization of the JTAG-DP controller without affecting the normal operation of the processor.

You do not have to synchronize the **nTRST** signal because it is synchronized within the processor, as shown in Figure 6-3 on page 6-7.

6.3.4 SW-DP reset

SW-DP is reset with **SWRSTn**. This reset must be synchronized to **SWCLK**.

6.3.5 Normal operation

During normal operation, neither processor reset nor power-on reset is asserted. If the JTAG-DP port is not being used, the value of **nTRST** does not matter.

Chapter 7
Power Management

This chapter describes the processor power management functions. It contains the following sections:

- *About power management* on page 7-2
- *System power management* on page 7-3

7.1 About power management

The processor makes extensive use of gated clocks to disable unused functionality, and disables inputs to unused functional blocks so only logic actively in use consumes any dynamic power.

The ARMv7-M architecture supports system sleep modes that enable the Cortex-M3 and system clocks to be stopped for greater power reductions. These are described in *System power management* on page 7-3.

7.2 System power management

Writing to the System Control Register (see *System Control Register* on page 8-23) controls the Cortex-M3 system power states. Table 7-1 shows the supported sleep modes.

Table 7-1 Supported sleep modes

Sleep mechanism	Description
Sleep-now	The *Wait For Interrupt* (WFI) or the *Wait For Event* (WFE) instructions request the sleep-now model. These instructions cause the NVIC to put the processor into the low-power state pending another exception.[a]
Sleep-on-exit	When the SLEEPONEXIT bit of the System Control Register is set, the processor enters the low-power state as soon as it exits the lowest priority ISR. The processor enters the low-power state without popping registers and a following exception is taken without having to push registers. The core stays in sleep state until another exception is pended. This is an automated WFI mode. **Note** Sleep-on-exit might return to base under various situations such as debug. Therefore, base code must be provided such as an idle loop or idle thread.
Deep-sleep	Deep-sleep is used in conjunction with Sleep-now and Sleep-on-exit. When the SLEEPDEEP bit of the System Control Register is set, the processor indicates to the system that deeper sleep is possible.

a. The WFI instruction can complete even if no exception becomes active. Do not use it to detect the occurrence of an exception. WFI is normally used in an idle loop in the Thread Mode. For more information on WFI, WFE, BASEPRI, and PRIMASK see the *ARMv7-M Architecture Reference Manual*.

The processor exports the following signals to indicate when the processor is sleeping:

SLEEPING This signal is asserted when in Sleep-now or Sleep-on-exit modes, and indicates that the clock to the processor can be stopped. On receipt of a new interrupt, the NVIC de-asserts this signal, releasing the core from sleep. An example of **SLEEPING** usage is shown in *SLEEPING* on page 7-4.

SLEEPDEEP

This signal is asserted when in Sleep-now or Sleep-on-exit modes when the **SLEEPDEEP** bit of the System Control Register is set. This signal is routed to the clock manager and can be used to gate the processor and system components including the *Phase Locked Loop* (PLL) to achieve greater power savings. On receipt of a new interrupt, the *Nested Vectored*

Interrupt Controller (NVIC) deasserts this signal, and release core sleep when the clock manager indicates that the clock is stable. An example of **SLEEPDEEP** usage is shown in *SLEEPDEEP* on page 7-4.

7.2.1 SLEEPING

Figure 7-1 shows an example of how to reduce power consumption by gating the **HCLK** clock to the processor with **SLEEPING** in the low-power state. If necessary, you can also use **SLEEPING** to gate other system components.

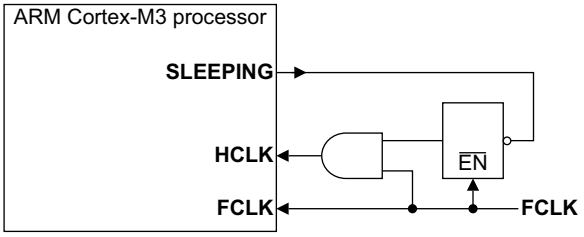


Figure 7-1 SLEEPING power control example

To detect interrupts, the processor must receive the free-running **FCLK** at all times. **FCLK** is used to clock:

- A small amount of logic in the NVIC that detects interrupts.
- The DWT and ITM blocks. These blocks can generate trace packets during sleep when enabled to do so. If the TRCENA bit of the Debug Exception and Monitor Control Register is enabled then the power consumption of those blocks will be minimized. See *Debug Exception and Monitor Control Register* on page 10-8.

FCLK frequency can be reduced during **SLEEPING** assertion.

7.2.2 SLEEPDEEP

Figure 7-2 on page 7-5 shows an example of how to further reduce power consumption by stopping the clock controller with **SLEEPDEEP** in the low-power state. When exiting low-power state, the **LOCK** signal indicates that the PLL is stable, and it is safe to enable the Cortex-M3 clock, ensuring that the processor is not re-started until the clocks are stable.

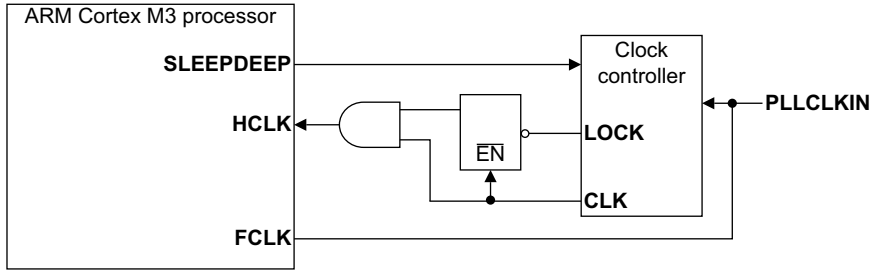


Figure 7-2 SLEEPDEEP power control example

To detect interrupts, the processor must receive the free-running **FCLK** in the low-power state. **FCLK** frequency can be reduced during **SLEEPDEEP** assertion.

Chapter 8
Nested Vectored Interrupt Controller

This chapter describes the *Nested Vectored Interrupt Controller* (NVIC). It contains the following sections:

- *About the NVIC* on page 8-2
- *NVIC programmer's model* on page 8-3
- *Level versus pulse interrupts* on page 8-40.

8.1 About the NVIC

The NVIC:

- facilitates low-latency exception and interrupt handling
- controls power management
- implements System Control Registers.

The NVIC supports up to 240 dynamically reprioritizable interrupts each with up to 256 levels of priority. Low latency interrupt processing is achieved by the closely coupled NVIC and processor core interface, enabling late arriving interrupts to be efficiently processed. The NVIC maintains knowledge of the stacked (nested) interrupts to enable tail-chaining of interrupts.

You can only fully access the NVIC from privileged mode, but you can pend interrupts in user-mode if you enable the Configuration Control Register (see *Configuration Control Register* on page 8-24). Any other user-mode access causes a bus fault.

All NVIC registers are accessible using byte, halfword, and word unless otherwise stated.

All NVIC registers and system debug registers are little endian regardless of the endianness state of the processor.

Processor exception handling is described in Chapter 5 *Exceptions*.

8.2 NVIC programmer's model

This section lists and describes the NVIC registers. It contains the following:

- *NVIC register map*
- *NVIC register descriptions* on page 8-7.

8.2.1 NVIC register map

Table 8-1 lists the NVIC registers. The NVIC space also implements System Control Registers. The NVIC space is split as follows:

- `0xE000E000` - `0xE000E00F`. Interrupt Type Register
- `0xE000E010` - `0xE000E0FF`. System Timer
- `0xE000E100` - `0xE000ECFF`. NVIC
- `0xE000ED00` - `0xE000ED8F`. System Control Block, including:
  - CPUID
  - System control, configuration, and status
  - Fault Reporting
- `0xE000EF00` - `0xE000EF0F`. Software Trigger Exception Register
- `0xE000EFD0` - `0xE000EFFF`. ID space.

Table 8-1 NVIC registers

Name of register	Type	Address	Reset value	Page
Interrupt Control Type Register	Read-only	`0xE000E004`	a	page 8-7
SysTick Control and Status Register	Read/write	`0xE000E010`	`0x00000000`	page 8-8
SysTick Reload Value Register	Read/write	`0xE000E014`	Unpredictable	page 8-9
SysTick Current Value Register	Read/write clear	`0xE000E018`	Unpredictable	page 8-10
SysTick Calibration Value Register	Read-only	`0xE000E01C`	STCALIB	page 8-11
Irq 0 to 31 Set Enable Register	Read/write	`0xE000E100`	`0x00000000`	page 8-12
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Irq 224 to 239 Set Enable Register	Read/write	`0xE000E11C`	`0x00000000`	page 8-12
Irq 0 to 31 Clear Enable Register	Read/write	`0xE000E180`	`0x00000000`	page 8-13

Table 8-1 NVIC registers (continued)

Name of register	Type	Address	Reset value	Page
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Irq 224 to 239 Clear Enable Register	Read/write	0xE000E19C	0x00000000	page 8-13
Irq 0 to 31 Set Pending Register	Read/write	0xE000E200	0x00000000	page 8-13
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Irq 224 to 239 Set Pending Register	Read/write	0xE000E21C	0x00000000	page 8-13
Irq 0 to 31 Clear Pending Register	Read/write	0xE000E280	0x00000000	page 8-14
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Irq 224 to 239 Clear Pending Register	Read/write	0xE000E29C	0x00000000	page 8-14
Irq 0 to 31 Active Bit Register	Read-only	0xE000E300	0x00000000	page 8-15
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Irq 224 to 239 Active Bit Register	Read-only	0xE000E31C	0x00000000	page 8-15
Irq 0 to 31 Priority Register	Read/write	0xE000E400	0x00000000	page 8-15
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
Irq 236 to 239 Priority Register	Read/write	0xE000E4F0	0x00000000	page 8-15

Table 8-1 NVIC registers (continued)

Name of register	Type	Address	Reset value	Page
CPUID Base Register	Read-only	0xE000ED00	0x410FC230	page 8-17
Interrupt Control State Register	Read/write or read-only	0xE000ED04	0x00000000	page 8-18
Vector Table Offset Register	Read/write	0xE000ED08	0x00000000	page 8-20
Application Interrupt/Reset Control Register	Read/write	0xE000ED0C	0x00000000	page 8-21
System Control Register	Read/write	0xE000ED10	0x00000000	page 8-23
Configuration Control Register	Read/write	0xE000ED14	0x00000000	page 8-24
System Handlers 4-7 Priority Register	Read/write	0xE000ED18	0x00000000	page 8-26
System Handlers 8-11 Priority Register	Read/write	0xE000ED1C	0x00000000	page 8-26
System Handlers 12-15 Priority Register	Read/write	0xE000ED20	0x00000000	page 8-26
System Handler Control and State Register	Read/write	0xE000ED24	0x00000000	page 8-27
Configurable Fault Status Registers	Read/write	0xE000ED28	0x00000000	page 8-30
Hard Fault Status Register	Read/write	0xE000ED2C	0x00000000	page 8-35
Debug Fault Status Register	Read/write	0xE000ED30	0x00000000	page 8-36
Mem Manage Address Register	Read/write	0xE000ED34	Unpredictable	page 8-37
Bus Fault Address Register	Read/write	0xE000ED38	Unpredictable	page 8-38
PFR0: Processor Feature register0	Read-only	0xE000ED40	0x00000030	-
PFR1: Processor Feature register1	Read-only	0xE000ED44	0x00000200	-
DFR0: Debug Feature register0	Read-only	0xE000ED48	0x00100000	-
AFR0: Auxiliary Feature register0	Read-only	0xE000ED4C	0x00000000	-
MMFR0: Memory Model Feature register0	Read-only	0xE000ED50	0x00000030	-
MMFR1: Memory Model Feature register1	Read-only	0xE000ED54	0x00000000	-
MMFR2: Memory Model Feature register2	Read-only	0xE000ED58	0x00000000	-
MMFR3: Memory Model Feature register3	Read-only	0xE000ED5C	0x00000000	-
ISAR0: ISA Feature register0	Read-only	0xE000ED60	0x01141110	-

Table 8-1 NVIC registers (continued)

Name of register	Type	Address	Reset value	Page
ISAR1: ISA Feature register1	Read-only	0xE000ED64	0x02111000	-
ISAR2: ISA Feature register2	Read-only	0xE000ED68	0x21112231	-
ISAR3: ISA Feature register3	Read-only	0xE000ED6C	0x01111110	-
ISAR4: ISA Feature register4	Read-only	0xE000ED70	0x01310102	-
Software Trigger Interrupt Register	Write Only	0xE000EF00	-	page 8-38
Peripheral identification register (PERIPHID4)	Read-only	0xE000EFD0	0x04	-
Peripheral identification register (PERIPHID5)	Read-only	0xE000EFD4	0x00	-
Peripheral identification register (PERIPHID6)	Read-only	0xE000EFD8	0x00	-
Peripheral identification register (PERIPHID7)	Read-only	0xE000EFDC	0x00	-
Peripheral identification register Bits 7:0 (PERIPHID0)	Read-only	0xE000EFE0	0x00	-
Peripheral identification register Bits 15:8 (PERIPHID1)	Read-only	0xE000EFE4	0xB0	-
Peripheral identification register Bits 23:16 (PERIPHID2)	Read-only	0xE000EFE8	0x0B	-
Peripheral identification register Bits 31:24 (PERIPHID3)	Read-only	0xE000EFEC	0x00	-
Component identification register Bits 7:0 (PCELLID0)	Read Only	0xE000EFF0	0x0D	-
Component identification register Bits 15:8 (PCELLID1)	Read-only	0xE000EFF4	0xE0	-
Component identification register Bits 23:16 (PCELLID2)	Read-only	0xE000EFF8	0x05	-
Component identification register Bits 31:24 (PCELLID3)	Read-only	0xE000EFFC	0xB1	-

a. Reset value depends on the number of interrupts defined.

8.2.2 NVIC register descriptions

The sections that follow describe how to use the NVIC registers.

Note

The *Memory Protection Unit* (MPU) registers, and the debug registers are described in Chapter 9 *Memory Protection Unit* and Chapter 10 *Core Debug* respectively.

Interrupt Controller Type Register

Read the Interrupt Controller Type Register to see the number of interrupt lines that the NVIC supports.

The register address, access type, and Reset state are:

Access Read-only

Address `0xE000E004`

Reset state Depends on the number of interrupts defined in this implementation of Cortex-M3.

Figure 8-1 shows the fields of the Interrupt Controller Type Register.

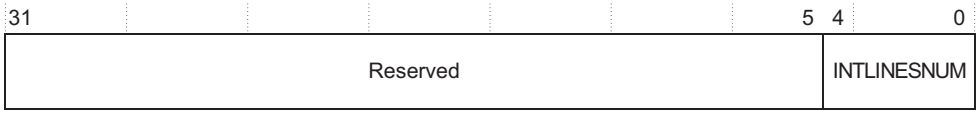


Figure 8-1 Interrupt Controller Type Register bit assignments

Table 8-2 describes the fields of the Interrupt Controller Type Register.

Table 8-2 Interrupt Controller Type Register bit assignments

Field	Name	Definition
[31:5]	-	Reserved.
[4:0]	INTLINESNUM	Total number of interrupt lines in groups of 32: b00000 = 0...32* b00001 = 33...64 b00010 = 65...96 b00011 = 97...128 b00100 = 129...160 b00101 = 161...192 b00110 = 193...224 b00111 = 225...256* *Cortex-M3 processor only supports between 1 and 240 external interrupts.

SysTick Control and Status Register

Use the SysTick Control and Status Register to enable the SysTick features.

The register address, access type, and Reset state are:

Address `0xE000E010`

Access Read/write

Reset state `0x00000000`

Figure 8-2 shows the fields of the SysTick Control and Status Register.

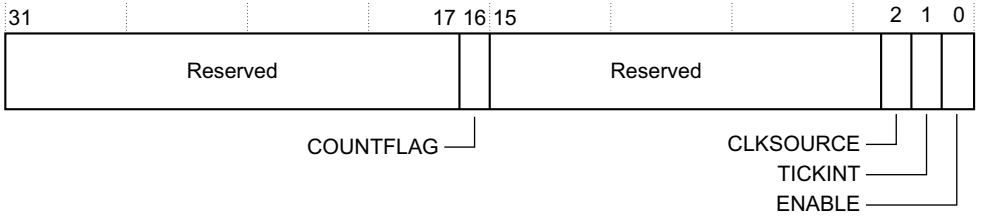


Figure 8-2 SysTick Control and Status Register bit assignments

Table 8-3 describes the fields of the SysTick Control and Status register.

Table 8-3 SysTick Control and Status Register bit assignments

Field	Name	Definition
[16]	COUNTFLAG	Returns 1 if timer counted to 0 since last time this was read. Clears on read.
[2]	CLKSOURCE	0 = external reference clock. 1 = core clock. If no reference clock is provided, it is held at 1 and so gives the same time as the core clock. The core clock must be at least 2.5 times faster than the reference clock. If it is not, the count values are unpredictable.
[1]	TICKINT	1 = counting down to 0 causes the SysTick handler to be pended. 0 = counting down to 0 does not cause the SysTick handler to be pended. Software can use the COUNTFLAG to determine if ever counted to 0.
[0]	ENABLE	1 = counter operates in a multi-shot way. That is, counter loads with the Reload value and then begins counting down. On reaching 0, it sets the COUNTFLAG to 1 and optionally pends the SysTick handler, based on TICKINT. It then loads the Reload value again, and begins counting. 0 = counter disabled.

SysTick Reload Value Register

Use the SysTick Reload Value Register to specify the start value to load into the current value register when the counter reaches 0. It can be any value between 1 and `0x00FFFFFF`. A start value of 0 is possible, but has no effect because the SysTick interrupt and COUNTFLAG are activated when counting from 1 to 0.

Therefore, as a multi-shot timer, repeated over and over, it fires every N+1 clock pulse, where N is any value from 1 to `0x00FFFFFF`. So, if the tick interrupt is required every 100 clock pulses, 99 must be written into the RELOAD. If a new value is written on each tick interrupt, so treated as single shot, then the actual count down must be written. For example, if a tick is next required after 400 clock pulses, 400 must be written into the RELOAD.

The register address, access type, and Reset state are:

Address `0xE000E014`
Access Read/write
Reset state Unpredictable

Figure 8-3 shows the fields of the SysTick Reload Value Register.

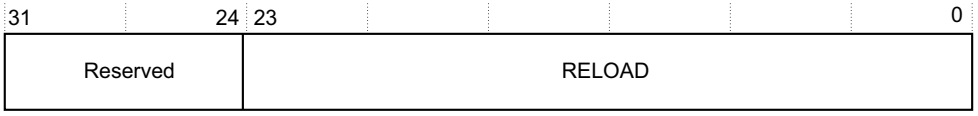


Figure 8-3 SysTick Reload Value Register bit assignments

Table 8-4 describes the fields of the SysTick Reload Value Register.

Table 8-4 SysTick Reload Value Register bit assignments

Field	Name	Definition
[23:0]	RELOAD	Value to load into the Current value register when the counter reaches 0.

SysTick Current Value Register

Use the SysTick Current Value Register to find the current value in the register.

The register address, access type, and Reset state are:

Address `0xE000E018`

Access Read/write clear

Reset state Unpredictable

Figure 8-4 shows the fields of the SysTick Current Value Register.

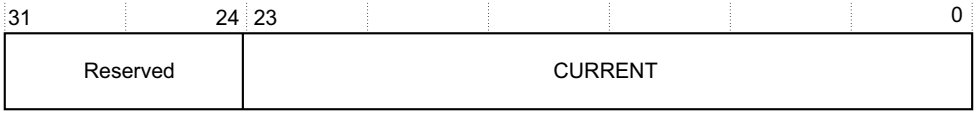


Figure 8-4 SysTick Current Value Register bit assignments

Table 8-5 describes the fields of the SysTick Current Value Register.

Table 8-5 SysTick Current Value Register bit assignments

Field	Name	Definition
[23:0]	CURRENT	Current value at the time the register is accessed. No read-modify-write protection is provided, so change with care. This register is write-clear. Writing to it with any value clears the register to 0. Clearing this register also clears the COUNTFLAG bit of the SysTick Control and Status Register.

SysTick Calibration Value Register

Use the SysTick Calibration Value Register to enable software to scale to any required speed using divide and multiply.

The register address, access type, and Reset state are:

Address `0xE000E01C`

Access Read

Reset state STCALIB

Figure 8-5 describes the fields of the SysTick Calibration Value Register.

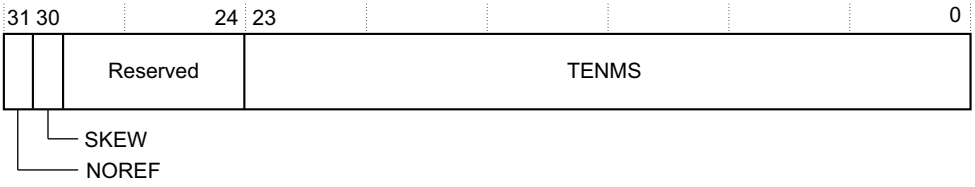


Figure 8-5 SysTick Calibration Value Register bit assignments

Table 8-6 describes the fields of the SysTick Calibration Value Register.

Table 8-6 SysTick Calibration Value Register bit assignments

Field	Name	Definition
[31]	NOREF	1 = the reference clock is not provided.
[30]	SKEW	1 = the calibration value is not exactly 10ms because of clock frequency. This could affect its suitability as a software real time clock.
[23:0]	TENMS	This value is the Reload value to use for 10ms timing. Depending on the value of SKEW, this might be exactly 10ms or might be the closest value. If this reads as 0, then the calibration value is not known. This is probably because the reference clock is an unknown input from the system or scalable dynamically.

Interrupt Set-Enable Registers

Use the Interrupt Set-Enable Registers to:

- enable interrupts
- determine which interrupts are currently enabled.

Each bit in the register corresponds to one of 32 interrupts. Setting a bit in the Interrupt Set-Enable Register enables the corresponding interrupt.

When the enable bit of a pending interrupt is set, the processor activates the interrupt based on its priority. When the enable bit is clear, asserting its interrupt signal pends the interrupt, but the interrupt cannot be activated, regardless of its priority. Therefore, a disabled interrupt can serve as a latched general-purpose I/O bit. You can read it and clear it without invoking an interrupt.

Clear an Interrupt Set-Enable Register bit by writing a 1 to the corresponding bit in the Interrupt Clear-Enable Register (see *Interrupt Clear-Enable Registers* on page 8-13).

Note

Clearing an Interrupt Set-Enable Register bit does not affect currently active interrupts. It only prevents new activations.

The register address, access type, and Reset state are:

Address `0xE000E100-0xE000E11C`

Access Read/write

Reset state `0x00000000`

Table 8-7 describes the field of the Interrupt Set-Enable Register.

Table 8-7 Bit functions of the Interrupt Set-Enable Register

Field	Name	Definition
[31:0]	SETENA	Interrupt set enable bits: 1 = enable interrupt 0 = disable interrupt. Writing 0 to a SETENA bit has no effect. Reading the bit returns its current state. Reset clears the SETENA field.

Interrupt Clear-Enable Registers

Use the Interrupt Clear-Enable Registers to:

- disable interrupts
- determine which interrupts are currently disabled.

Each bit in the register corresponds to one of the 32 interrupts. Setting an Interrupt Clear-Enable Register bit disables the corresponding interrupt.

The register address, access type, and Reset state are:

Address `0xE000E180-0xE000E19C`
Access Read/write
Reset state `0x00000000`

Table 8-8 describes the field of the Interrupt Clear-Enable Register.

Table 8-8 Bit functions of the Interrupt Clear-Enable Register

Field	Name	Definition
[31:0]	CLRENA	Interrupt clear-enable bits: 1 = disable interrupt 0 = enable interrupt. Writing 0 to a CLRENA bit has no effect. Reading the bit returns its current state.

Interrupt Set-Pending Register

Use the Interrupt Set-Pending Register to:

- force interrupts into the pending state
- determine which interrupts are currently pending.

Each bit in the register corresponds to one of the 32 interrupts. Setting an Interrupt Set-Pending Register bit pends the corresponding interrupt.

Clear an Interrupt Set-Pending Register bit by writing a 1 to the corresponding bit in the Interrupt Clear-Pending Register (see *Interrupt Clear-Pending Register*). Clearing the Interrupt Set-Pending Register bit puts the interrupt into the non-pended state.

Note

Writing to the Interrupt Set-Pending Register has no affect on an interrupt that is already pending or is disabled.

The register address, access type, and Reset state are:

Address `0xE000E200-0xE000E21C`

Access Read/write

Reset state `0x00000000`

Table 8-9 describes the field of the Interrupt Set-Pending Register.

Table 8-9 Bit functions of the Interrupt Set-Pending Register

Field	Name	Definition
[31:0]	SETPEND	Interrupt set-pending bits: 1 = pend the corresponding interrupt 0 = corresponding interrupt not pending. Writing 0 to a SETPEND bit has no effect. Reading the bit returns its current state.

Interrupt Clear-Pending Register

Use the Interrupt Clear-Pending Register to:

- clear pending interrupts
- determine which interrupts are currently pending.

Each bit in the register corresponds to one of the 32 interrupts. Setting an Interrupt Clear-Pending Register bit puts the corresponding pending interrupt in the inactive state.

Note

Writing to the Interrupt Clear-Pending Register has no effect on an interrupt that is active unless it is pending as well.

The register address, access type, and Reset state are:

Address `0xE000E280-0xE000E29C`

Access Read/write

Reset state `0x00000000`

Table 8-10 describes the field of the Interrupt Clear-Pending Registers.

Table 8-10 Bit functions of the Interrupt Clear-Pending Registers

Field	Name	Definition
[31:0]	CLRPEND	Interrupt clear-pending bits: 1 = clear pending interrupt 0 = do not clear pending interrupt. Writing 0 to a CLRPEND bit has no effect. Reading the bit returns its current state.

Active Bit Register

Read the Active Bit Register to determine which interrupts are active. Each flag in the register corresponds to one of the 32 interrupts.

The register address, access type, and Reset state are:

Address `0xE000E300-0xE00031C`

Access Read-only

Reset state `0x00000000`

Table 8-11 describes the field of the Active Bit Register.

Table 8-11 Bit functions of the Active Bit Register

Field	Name	Definition
[31:0]	ACTIVE	Interrupt active flags: 1 = interrupt active or preempted and stacked 0 = interrupt not active or stacked.

Interrupt Priority Registers

Use the Interrupt Priority Registers to assign a priority from 0 to 255 to each of the available interrupts. 0 is the highest priority, and 255 is the lowest.

The priority registers are stored with the most significant bit (MSB) first. This means that if there are four bits of priority, the priority value is stored in bits [7:4] of the byte. However, if there are three bits of priority, the priority value is stored in bits [7:5] of the byte. This means that an application can work even if it does not know how many priorities are possible.

The register address, access type, and Reset state are:

Address `0xE000E400-0xE000E41F`

Access Read/write

Reset state `0x00000000`

Figure 8-6 shows the fields of Interrupt Priority Registers 0-7.

	31 24	23 16	15 8	7 0
E000E400	PRI_3	PRI_2	PRI_1	PRI_0
E000E404	PRI_7	PRI_6	PRI_5	PRI_4
E000E408	PRI_11	PRI_10	PRI_9	PRI_8
E000E40C	PRI_15	PRI_14	PRI_13	PRI_12
E000E410	PRI_19	PRI_18	PRI_17	PRI_16
E000E414	PRI_23	PRI_22	PRI_21	PRI_20
E000E418	PRI_27	PRI_26	PRI_25	PRI_24
E000E41C	PRI_31	PRI_30	PRI_29	PRI_28

Figure 8-6 Interrupt Priority Registers 0-31 bit assignments

The lower PRI_*n* bits can specify subpriorities for priority grouping. See *Exception priority* on page 5-5.

Table 8-12 describes the fields of the Interrupt Priority Registers.

Table 8-12 Interrupt Priority Registers 0-31 bit assignments

Field	Name	Definition
[7:0]	PRI_*n*	Priority of interrupt *n*

CPU ID Base Register

Read the CPU ID Base Register to determine:

- the ID number of the Cortex-M3 core
- the version number of the Cortex-M3 core
- the implementation details of the core.

The register address, access type, and Reset state are:

Address `0xE000ED00`
Access Read-only
Reset state `0x410FC230`

Figure 8-7 shows the fields of the CPUID Base Register.

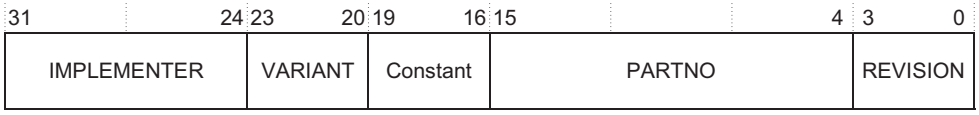


Figure 8-7 CPUID Base Register bit assignments

Table 8-13 describes the fields of the CPUID Base Register.

Table 8-13 CPUID Base Register bit assignments

Field	Name	Definition
[31:24]	IMPLEMENTER	Implementer code. ARM is `0x41`
[23:20]	VARIANT	Implementation defined variant number.

Table 8-13 CPUID Base Register bit assignments

Field	Name	Definition
[19:16]	Constant	Reads as `0xF`
[15:4]	PARTNO	Number of processor within family: [11:10] b11=Cortex family [9:8] b00=version [7:6] b00=reserved [5:4] b10=M (v7-M) [3:0] X=family member. Cortex-M3 is b0011.
[3:0]	REVISION	Implementation defined revision number.

Interrupt Control State Register

Use the Interrupt Control State Register to:

- set a pending NMI
- set or clear a pending SVC
- set or clear a pending SysTick
- check for pending exceptions
- check the vector number of the highest priority pended exception
- check the vector number of the active exception.

The register address, access type, and Reset state are:

Address `0xE000ED04`

Access Read/write or read-only

Reset state `0x00000000`

Figure 8-8 on page 8-19 shows the fields of the Interrupt Control State Register.

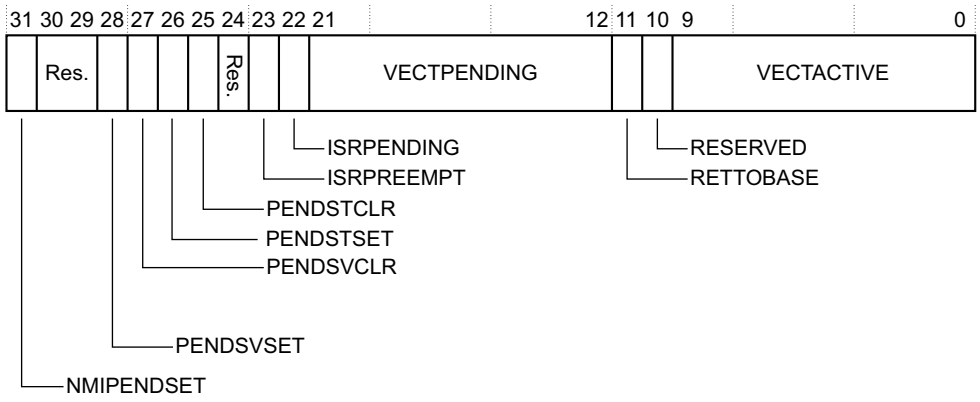


Figure 8-8 Interrupt Control State Register bit assignments

Table 8-14 describes the fields of the Interrupt Control State Register.

Table 8-14 Interrupt Control State Register bit assignments

Field	Name	Type	Definition
[31]	NMIPENDSET	Read/write	Set pending NMI bit: 1 = set pending NMI 0 = do not set pending NMI. NMIPENDSET pends and activates an NMI. Because NMI is the highest-priority interrupt, it takes effect as soon as it registers.
[30:29]	-	-	Reserved.
[28]	PENDSVSET	Read/write	Set pending pendSV bit: 1 = set pending pendSV 0 = do not set pending pendSV.
[27]	PENDSVCLR	Write-only	Clear pending pendSV bit: 1 = clear pending pendSV 0 = do not clear pending pendSV.
[26]	PENDSTSET	Read/write	Set a pending SysTick bit 1 = set pending SysTick 0 = do not set pending SysTick.
[25]	PENDSTCLR	Write-only	Clear pending SysTick bit: 1 = clear pending SysTick 0 = do not clear pending SysTick.

Table 8-14 Interrupt Control State Register bit assignments (continued)

Field	Name	Type	Definition
[23]	ISRPREEMPT	Read-only	Must only be used at debug time. It indicates that a pending interrupt is going to become active in the next running cycle. If C_MASKINTS is clear in the Debug Halting Control and Status Register, the interrupt is serviced.
[22]	ISRPENDING	Read-only	Interrupt pending flag. Excludes NMI and Faults: 1 = Interrupt pending 0 = Interrupt not pending.
[21:12]	VECTPENDING	Read-only	Pending ISR number field. VECTPENDING contains the interrupt number of the highest priority pending ISR.
[10]	-	-	Reserved.
[11]	RETTOBASE	Read-only	This bit is 1when a return-from-exception would return to Base level of activation, if no other exception is pending. If in Thread state, in a Handler more than one level of activation from Base, or in a Handler which is not marked as active (faults on return), this is 0.
[9:0]	VECTACTIVE	Read-only	Active ISR number field. VECTACTIVE contains the interrupt number of the currently running ISR, including NMI and Hard Fault. A shared handler can use VECTACTIVE to determine which interrupt invoked it. You can subtract 16 from the VECTACTIVE field to index into the Interrupt Clear/Set Enable, Interrupt Clear Pending/SetPending and Interrupt Priority Registers. **INTISR[0]** has vector number 16. Reset clears the VECTACTIVE field.

Vector Table Offset Register

Use the Vector Table Offset Register to determine:

- if the vector table is in RAM or code memory
- the vector table offset.

The register address, access type, and Reset state are:

Address `0xE000ED08`
Access Read/write
Reset state `0x00000000`

Figure 8-9 on page 8-21 shows the fields of the Vector Table Offset Register.

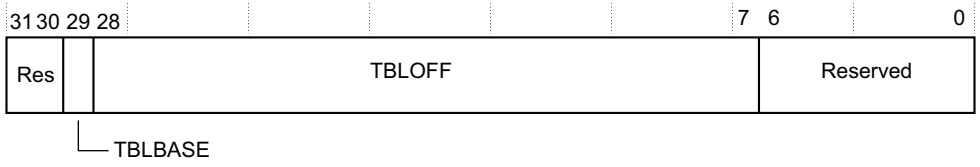


Figure 8-9 Vector Table Offset Register bit assignments

Table 8-15 describes the fields of the Vector Table Offset Register.

Table 8-15 Vector Table Offset Register bit assignments

Field	Name	Definition
[31:30]	-	Reserved
[29]	TBLBASE	Table Base is in Code (0) or RAM (1)
[28:7]	TBLOFF	Vector table base offset field. Contains the offset of the table base from the bottom of the SRAM or CODE space.
[6:0]	-	Reserved.

The Vector Table Offset Register positions the vector table in CODE or SRAM space. The default, on reset, is 0 (CODE space). When setting a position, the offset must be aligned based on the number of exceptions in the table. This means that the minimal alignment is 32 words which can be used for up to 16 interrupts. For more interrupts, you must adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, the alignment must be on a 64-word boundary because table size is 37 words, next power of two is 64.

Application Interrupt and Reset Control Register

Use the Application Interrupt and Reset Control Register to:

- determine data endianness
- clear all active state information for debug or to recover from a hard failure
- execute a system reset
- alter the priority grouping position (binary point).

The register address, access type, and Reset state are:

Address `0xE000ED0C`
Access Read/write
Reset state `0x00000000`

Figure 8-10 shows the fields of the Application Interrupt and Reset Control Register.

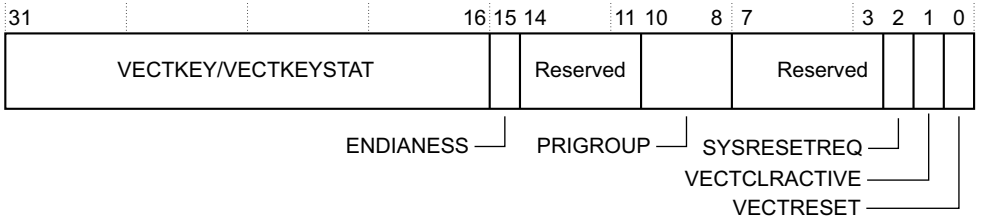


Figure 8-10 Application Interrupt and Reset Control Register bit assignments

Table 8-16 describes the fields of the Application Interrupt and Reset Control Register.

Table 8-16 Application Interrupt and Reset Control Register bit assignments

Field	Name	Definition
[31:16]	VECTKEY	Register key. Writing to this register requires `0x5FA` in the VECTKEY field. Otherwise the write value is ignored.
[31:16]	VECTKEYSTAT	Reads as `0xFA05`
[15]	ENDIANESS	Data endianness bit: 1 = big endian 0 = little endian ENDIANESS is sampled from the BIGEND input port during reset. ENDIANESS cannot be changed outside of reset.
[14:11]	-	Reserved
[10:8]	PRIGROUP	Interrupt priority grouping field: PRIGROUP — Split of pre-emption priority from subpriority 0 — 7.1 indicates seven bits of pre-emption priority, one bit of subpriority 1 — 6.2 indicates six bits of pre-emption priority, two bits of subpriority 2 — 5.3 indicates five bits of pre-emption priority, three bits of subpriority 3 — 4.4 indicates four bits of pre-emption priority, four bits of subpriority 4 — 3.5 indicates three bits of pre-emption priority, five bits of subpriority 5 — 2.6 indicates two bits of pre-emption priority, six bits of subpriority 6 — 1.7 indicates one bit of pre-emption priority, seven bits of subpriority 7 — 0.8 indicates no pre-emption priority, eight bits of subpriority

Table 8-16 Application Interrupt and Reset Control Register bit assignments (continued)

Field	Name	Definition
		PRIGROUP field is a binary point position indicator for creating subpriorities for exceptions that share the same pre-emption level. It divides the PRI_*n* field in the Interrupt Priority Register into a pre-emption level and a subpriority level. The binary point is a left-of value. This means that the PRIGROUP value represents a point starting at the left of the LSB. This is bit 0 of 7:0. The lowest value might not be 0 depending on the number of bits allocated for priorities, and implementation choices.
[7:3]	-	Reserved.
[2]	SYSRESETREQ	Causes a signal to be asserted to the outer system that indicates a reset is requested. Intended to force a large system reset of all major components except for debug. Setting this bit does not prevent Halting Debug from running.
[1]	VECTCLRACTIVE	Clear active vector bit: 1 = clear all state information for active NMI, fault, and interrupts 0 = do not clear. It is the responsibility of the application to reinitialize the stack. The VECTCLRACTIVE bit is for returning to a known state during debug. The VECTCLRACTIVE bit self-clears. IPSR is not cleared by this operation. So, if used by an application, it must only be used at the base level of activation, or within a system handler whose active bit can be set.
[0]	VECTRESET	System Reset bit. Resets the system, with the exception of debug components: 1 = reset system 0 = do not reset system. The VECTRESET bit self-clears. Reset clears the VECTRESET bit. For debugging, only write this bit when the core is halted.

System Control Register

Use the System Control Register for power-management functions:

- signal to the system when the Cortex-M3 processor can enter a low power state
- control how the processor enters and exits low power states.

The register address, access type, and Reset state are:

Address `0xE000ED10`
Access Read/write
Reset state `0x00000000`

Figure 8-11 shows the fields of the System Control Register.

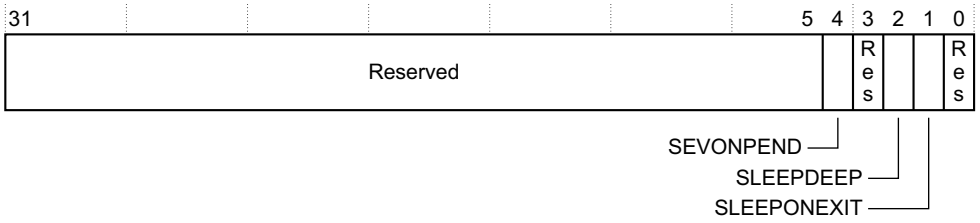


Figure 8-11 System Control Register bit assignments

Table 8-17 describes the fields of the System Control Register.

Table 8-17 System Control Register bit assignments

Field	Name	Definition
[31:5]	-	Reserved.
[4]	SEVONPEND	When enabled, this causes WFE to wake up when an interrupt moves from unpended to pended. Otherwise, WFE only wakes up from an event signal, external and SEV instruction generated. The event input, **RXEV**, is registered even when not waiting for an event, and so effects the next WFE.
[2]	SLEEPDEEP	Sleep deep bit: 1 = indicates to the system that Cortex-M3 clock can be stopped. Setting this bit causes the **SLEEPDEEP** port to be asserted when the processor can be stopped. 0 = not OK to turn off system clock. For more information about the use of **SLEEPDEEP**, see Chapter 7 *Power Management*.
[1]	SLEEPONEXIT	Sleep on exit when returning from Handler mode to Thread mode: 1 = sleep on ISR exit. 0 = do not sleep when returning to Thread mode. Enables interrupt driven applications to avoid returning to empty main application.
[0]	-	Reserved.

Configuration Control Register

Use the Configuration Control Register to:

- enable NMI, Hard Fault and FAULTMASK to ignore bus fault
- trap divide by zero, and unaligned accesses
- enable user access to the Software Trigger Exception Register
- control entry to Thread Mode.

The register address, access type, and Reset state are:

Address `0xE000ED14`

Access Read/write

Reset state `0x00000000`

Figure 8-12 shows the fields of the Configuration Control Register.

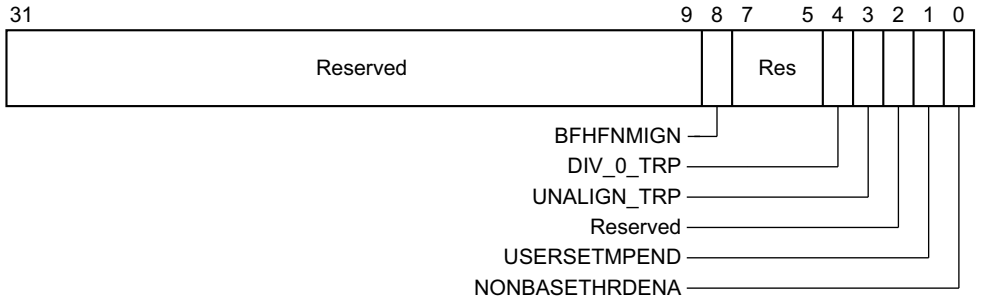


Figure 8-12 Configuration Control Register bit assignments

Table 8-18 describes the fields of the Configuration Control Register.

Table 8-18 Configuration Control Register bit assignments

Field	Name	Definition
[8]	BFHFNMIGN	When enabled, this causes handlers running at priority -1 and -2 (Hard Fault, NMI, and FAULTMASK escalated handlers) to ignore Data Bus faults caused by load and store instructions. When disabled, these bus faults cause a lock-up. This enable must be used with extreme caution. All data bus faults are ignored – it must only be used when the handler and its data are in absolutely safe memory. Its normal use is to probe system devices and bridges to detect control path problems and fix them.
[4]	DIV_0_TRP	Trap on Divide by 0. This enables faulting/halting when an attempt is made to divide by 0. The relevant Usage Fault Status Register bit is DIVBYZERO, see *Usage Fault Status Register* on page 8-33.

Table 8-18 Configuration Control Register bit assignments

Field	Name	Definition
[3]	UNALIGN_TRP	Trap for unaligned access. This enables faulting/halting on any unaligned half or full word access. Unaligned load-store multiples always fault. The relevant Usage Fault Status Register bit is UNALIGNED, see *Usage Fault Status Register* on page 8-33.
[1]	USERSETMPEND	If written as 1, enables user code to write the Software Trigger Interrupt register to trigger (pend) a Main exception, which is one associated with the Main stack pointer.
[0]	NONEBASETHRDENA	When 0, default, Thread mode can only be entered when returning from the last exception. When set to 1, Thread mode can be entered from any level in Handler mode by controlled return value.

System Handler Priority Registers

Use the three System Handler Priority Registers to prioritize the following system handlers:

- memory manage
- bus fault
- usage fault
- debug monitor
- SVC
- SysTick
- PendSV

System handlers are a special class of exception handler that can have their priority set to any of the priority levels. Most can be masked on (enabled) or off (disabled). When disabled, the fault is always treated as a Hard Fault.

The register addresses, access types, and Reset states are:

Address `0xE000ED18`, `0xE000ED1C` , `0xE000ED20`

Access Read/write

Reset state `0x00000000`

Figure 8-13 on page 8-27 shows the fields of the System Handler Priority Registers.

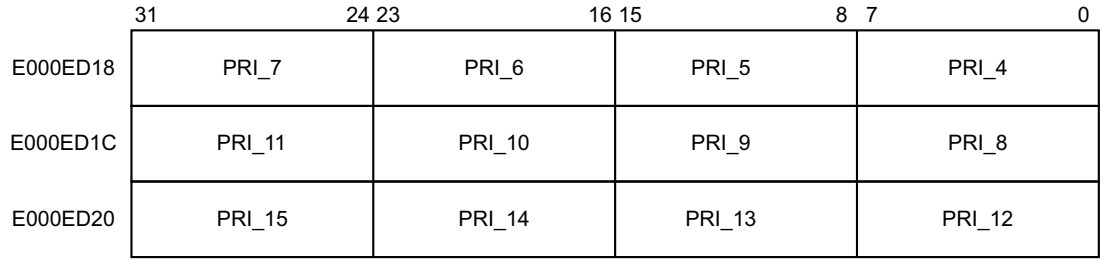


Figure 8-13 System Handler Priority Registers bit assignments

Table 8-19 describes the fields of the System Handler Priority Registers.

Table 8-19 System Handler Priority Registers bit assignments

Field	Name	Definition
[31:24]	PRI_N3	Priority of system handler 7, 11, and 15. Reserved, SVCall, and SysTick.
[23:16]	PRI_N2	Priority of system handler 6, 10, and 14. Usage Fault, reserved, and PendSV.
[15:8]	PRI_N1	Priority of system handler 5, 9, and 13, Bus Fault, reserved, and reserved.
[7:0]	PRI_N	Priority of system handler 4, 8 and 12. Mem Manage, reserved, and Debug Monitor.

System Handler Control and State Register

Use the System Handler Control and State Register to:

- enable or disable the system handlers
- determine the pending status of bus fault, mem manage fault, and SVC
- determine the active status of the system handlers

If a fault condition occurs while its fault handler is disabled, the fault escalates to a Hard Fault.

The register address, access type, and Reset state are:

Address `0xE000ED24`

Access Read/write

Reset state `0x00000000`

Figure 8-14 on page 8-28 shows the fields of the System Handler and State Control Register.

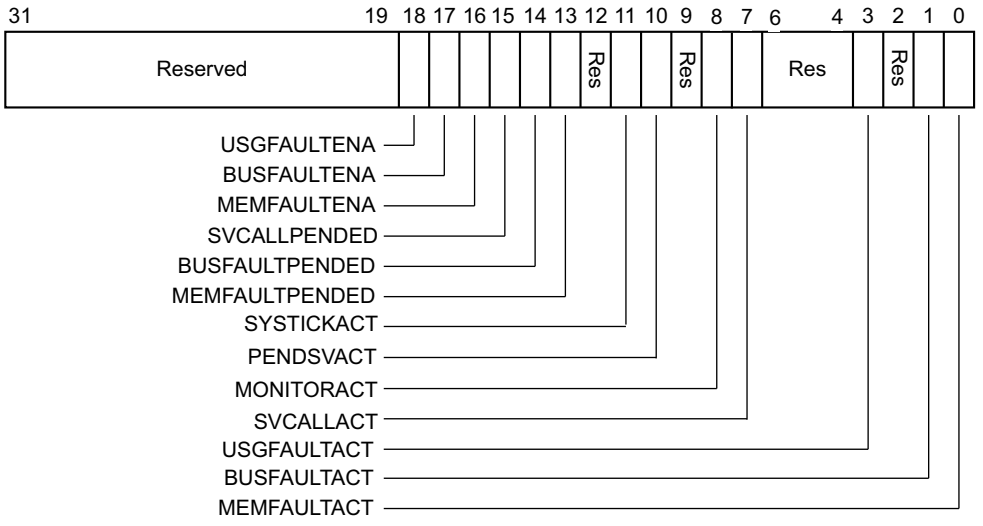


Figure 8-14 System Handler Control and State Register bit assignments

Table 8-20 describes the fields of the System Handler Control Register.

Table 8-20 System Handler Control and State Register bit assignment

Field	Name	Definition
[31:19]	-	Reserved
[18]	USGFAULTENA	Set to 0 to disable, else 1 for enabled.
[17]	BUSFAULTENA	Set to 0 to disable, else 1 for enabled.
[16]	MEMFAULTENA	Set to 0 to disable, else 1 for enabled.
[15]	SVCALLPENDED	Reads as 1 if SVCall is pended Started to invoke, but was replaced by a higher priority interrupt.
[14]	BUSFAULTPENDED	Reads as 1 if BusFault is pended Started to invoke, but was replaced by a higher priority interrupt.
[13]	MEMFAULTPENDED	Reads as 1 if MemManage is pended Started to invoke, but was replaced by a higher priority interrupt.
[12]	-	Reserved
[11]	SYSTICKACT	Reads as 1 if SysTick is active.
[10]	PENDSVACT	Reads as 1 if PendSV is active.

Table 8-20 System Handler Control and State Register bit assignment (continued)

Field	Name	Definition
[9]	-	Reserved
[8]	MONITORACT	Reads as 1 if the Monitor is active.
[7]	SVCALLACT	Reads as 1 if SVCall is active.
[6:4]	-	Reserved
[3]	USGFAULTACT	Reads as 1 if UsageFault is active.
[2]	-	Reserved
[1]	BUSFAULTACT	Reads as 1 if BusFault is active.
[0]	MEMFAULTACT	Reads as 1 if MemManage is active.

The active bits indicate if any of the system handlers are active, running now or stacked because of pre-emption. This information is used for debugging and is also used by the application handlers. The pend bits are only set when a fault that cannot be retried has been deferred because of late arrival of a higher priority interrupt.

Caution

The active bits can be written, cleared or set, but this must be used with extreme caution. Clearing and setting these does not repair stack contents nor clean up other data structures. Clearing and setting is intended to be used by context switchers to save a thread's context, even when in a fault handler. The most common case is to save the context of a thread which is in an SVCall handler or UsageFault handler, for undefined instruction and Coprocessor emulation. The model for doing this is to save the current state, switch out the stack containing the handler's context, load the state of the new thread, switch in the new thread's stacks, and then return to the thread. The active bit of the current handler must never be cleared, because the IPSR is not changed to reflect this. It must only be used to change stacked active handlers.

As indicated, the SVCALLPENDED and BUSFAULTPENDED bits are set when the corresponding handler is held off by a late arriving interrupt. These bits are not cleared until the underlying handler is actually invoked. That is, if a stack error or vector read error occurs before the SVCall or BusFault handler is started, the bits are not cleared. This enables the push-error or vector-read-error handler to choose to clear them or retry

Configurable Fault Status Registers

Use the three Configurable Fault Status Registers to obtain information about local faults. These registers include:

- *Memory Manage Fault Status Register*
- *Bus Fault Status Register* on page 8-31
- *Usage Fault Status Register* on page 8-33.

The flags in these registers indicate the causes of local faults. Multiple flags can be set if more than one fault occurs. These register are read/write-clear. This means that they can be read normally, but writing a 1 to any bit will clear that bit.

The register addresses, access types, and Reset states are:

Address `0xE000ED28` Memory Manage Fault Status Register
`0xE000ED29` Bus Fault Status Register
`0xE000ED2A` Usage Fault Status Register

Access Read/write clear

Reset state `0x00000000`

Figure 8-15 shows the fields of the configurable fault status registers.

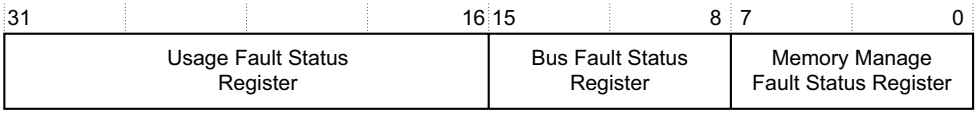


Figure 8-15 Local fault status registers bit assignments

Memory Manage Fault Status Register

The flags in the Memory Manage Fault Status Register indicate the cause of memory access faults.

The register address, access type, and Reset state are:

Address `0xE000ED28`

Access Read/write clear

Reset state `0x00000000`

Figure 8-16 on page 8-31 shows the fields of the Memory Manage Fault Status Register.

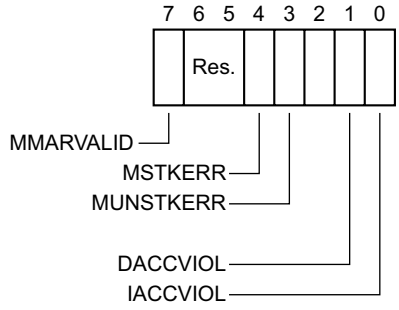


Figure 8-16 Memory Manage Fault Register bit assignments

Table 8-21 describes the fields of the Memory Manage Fault Status Register.

Table 8-21 Memory Manage Fault Status Register bit assignments

Field	Name	Definition
[7]	MMARVALID	*Memory Manage Address Register* (MMAR) address valid flag: 1 = valid fault address in MMAR. A memory manage fault can be cleared by a later arriving fault, such as a bus fault. 0 = no valid fault address in MMAR. If a MemManage fault occurs which is escalated to a Hard fault because of priority, the Hard Fault handler must clear this bit. This prevents problems on return to a stacked active MemManage handler whose MMAR value has been overwritten.
[4]	MSTKERR	Stacking from exception has caused one or more access violations. The SP is still adjusted and the values in the context area on the stack might be incorrect. The MMAR is not written.
[3]	MUNSTKERR	Unstack from exception return has caused one or more access violations. This is chained to the handler, so that the original return stack is still present. SP is not adjusted from failing return and new save is not performed. The MMAR is not written.
[1]	DACCVIOL	Data access violation flag. Attempting to load or store at a location that does not permit the operation sets the DACCVIOL flag. The return PC points to the faulting instruction. This error loads MMAR with the address of the attempted access.
[0]	IACCVIOL	Instruction access violation flag. Attempting to fetch an instruction from a location that does not permit execution sets the IACCVIOL flag. This occurs on any access to an XN region, even when the MPU is disabled or not present. The return PC points to the faulting instruction. The MMAR is not written.

Bus Fault Status Register

The flags in the Bus Fault Status Register indicate the cause of bus access faults.

The register address, access type, and Reset state are:

Address 0xE000ED29

Access Read/write clear

Reset state 0x00000000

Figure 8-17 shows the bit fields of the Bus Fault Status Register.

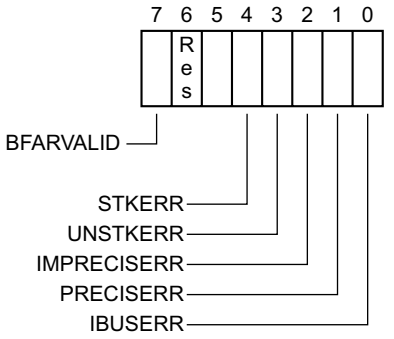


Figure 8-17 Bus Fault Status Register bit assignments

Table 8-22 describes the fields of the Bus Fault Status Register.

Table 8-22 Bus Fault Status Register bit assignments

Field	Name	Definition
[7]	BFARVALID	This bit is set if the *Bus Fault Address Register* (BFAR) contains a valid address. This is true after a bus fault where the address is known. This bit can be cleared by other faults, such as a Mem Manage fault occurring later. If a Bus fault occurs which is escalated to a Hard Fault because of priority, the Hard Fault handler must clear this bit. This prevents problems if returning to a stacked active Bus fault handler whose BFAR value has been overwritten.
[6:4]	-	Reserved.
[4]	STKERR	Stacking from exception has caused one or more bus faults. The SP is still adjusted and the values in the context area on the stack might be incorrect. The BFAR is not written.
[3]	UNSTKERR	Unstack from exception return has caused one or more bus faults. This is chained to the handler, so that the original return stack is still present. SP is not adjusted from failing return and new save is not performed. The BFAR is not written.

Table 8-22 Bus Fault Status Register bit assignments (continued)

Field	Name	Definition
[2]	IMPRECISERR	Imprecise data bus error. It is a BusFault, but the Return PC is not related to the causing instruction. This is not a synchronous fault. So, if detected when the priority of the current activation is higher than the Bus Fault, it only pends. Bus fault activates when returning to a lower priority activation. If a precise fault occurs before returning to a lower priority exception, the handler detects both IMPRECISERR set and one of the precise fault status bits set at the same time. The BFAR is not written.
[1]	PRECISERR	Precise data bus error return.
[0]	IBUSERR	Instruction bus error flag: 1 = instruction bus error 0 = no instruction bus error. The IBUSERR flag is set by a prefetch error. The fault stops on the instruction, so if the error occurs under a branch shadow, no fault occurs. The BFAR is not written.

Usage Fault Status Register

The flags in the Usage Fault Status Register indicate the following errors:

- illegal combination of EPSR and instruction
- illegal PC load
- illegal processor state
- instruction decode error
- attempt to use a coprocessor instruction
- illegal unaligned access.

The register address, access type, and Reset state are:

Address `0xE000ED2B`
Access Read/write clear
Reset state `0x00000000`

Figure 8-18 on page 8-34 shows the fields of the Usage Fault Status Register.

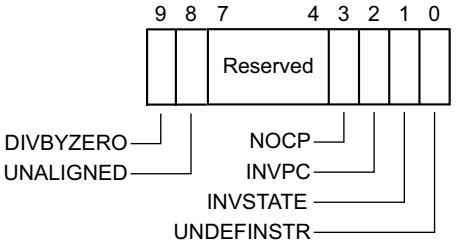


Figure 8-18 Usage Fault Status Register bit assignments

Table 8-23 describes the fields of the Usage Fault Status Register.

Table 8-23 Usage Fault Status Register bit assignments

Field	Name	Definition
[9]	DIVBYZERO	When DIV_0_TRP (see *Configuration Control Register* on page 8-24) is enabled and an SDIV or UDIV instruction is used with a divisor of 0, this fault occurs The instruction is executed and the return PC points to it. If DIV_0_TRP is not set, then the divide returns a Quotient of 0.
[8]	UNALIGNED	When UNALIGN_TRP is enabled (see *Configuration Control Register* on page 8-24), and there is an attempt to make an unaligned memory access, then this fault will occur. Unaligned LDM/STM/LDRD/STRD/LDC/STC instructions always fault irrespective of the setting of UNALIGN_TRP.
[7:4]	-	Reserved.
[3]	NOCP	Attempt to use a coprocessor instruction. The processor does not support coprocessor instructions.
[2]	INVPC	Attempt to load EXC_RETURN into PC illegally. Invalid instruction, invalid context, invalid value. The return PC points to the instruction which tried to set the PC.
[1]	INVSTATE	Invalid combination of EPSR and instruction, for reasons other than UNDEFINED instruction. Return PC points to faulting instruction, with the invalid state.
[0]	UNDEFINSTR	The UNDEFINSTR flag is set when the processor attempts to execute an undefined instruction. This is an instruction that could not be decoded. The return PC points to the undefined instruction.

Note

The fault bits are additive if more than one fault occurs before this register is cleared.

Hard Fault Status Register

Use the Hard Fault Status Register to obtain information about events that activate the Hard Fault handler.

The register address, access type, and Reset state are:

Address `0xE000ED2C`

Access Read/write clear

Reset state `0x00000000`

The HFSR is a write-clear register. This means that writing a 1 to a bit clears that bit. Figure 8-19 shows the fields of the Hard Fault Status Register.

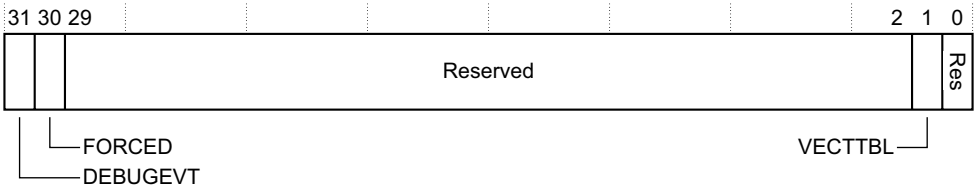


Figure 8-19 Hard Fault Status Register bit assignments

Table 8-24 describes the fields of the Hard Fault Status Register.

Table 8-24 Hard Fault Status Register bit assignments

Field	Name	Definition
[31]	DEBUGEVT	This bit is set if there is a fault related to debug. This is only possible when halting debug is not enabled. For monitor enabled debug, it only happens for BKPT when the current priority is higher than the monitor. When both halting and monitor debug are disabled, it only happens for debug events which are not ignored (minimally, BKPT). The Debug Fault Status Register is updated.
[30]	FORCED	Hard Fault activated because a Configurable Fault was received and cannot activate because of priority or because the Configurable Fault is disabled. The Hard Fault handler then has to read the other fault status registers to determine cause.
[29:2]	-	Reserved.
[1]	VECTTBL	This bit is set if there is a fault because of vector table read on exception processing (Bus Fault). This case is always a Hard Fault. The return PC points to the preempted instruction.

Debug Fault Status Register

Use the Debug Fault Status Register to monitor:

- external debug requests
- vector catches
- data watchpoint match
- BKPT instruction execution
- halt requests.

Multiple flags in the Debug Fault Status Register can be set when multiple fault conditions occur. The register is read/write clear. This means that it can be read normally. Writing a 1 to a bit clears that bit.

Note

These bits are not set unless the event is caught. This means that it causes a stop of some sort. If halting debug is enabled, these events stop the processor into debug. If debug is disabled and the debug monitor is enabled, then this becomes a debug monitor handler call, if priority allows. If debug and the monitor are both disabled, some of these events are Hard Faults, and the DBGEVT bit is set in the Hard Fault status register, and some are ignored.

The register address, access type, and Reset state are:

Address `0xEEE0ED30`

Access Read/write clear

Reset state `0x00000000`

Figure 8-20 shows the fields of the Debug Fault Status Register.

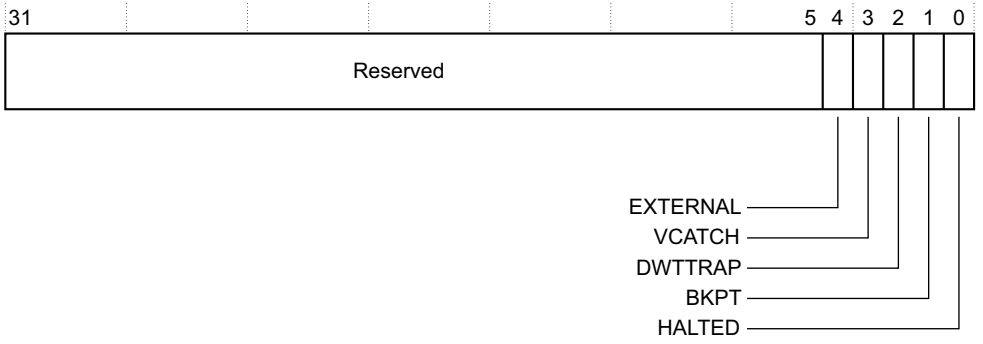


Figure 8-20 Debug Fault Status Register bit assignments

Table 8-25 describes the fields of the Debug Fault Status Register.

Table 8-25 Debug Fault Status Register bit assignments

Field	Name	Definition
[31:5]	-	Reserved
[4]	EXTERNAL	External debug request flag: 1 = **EDBGRQ** signal asserted 0 = **EDBGRQ** signal not asserted. The processor stops on next instruction boundary.
[3]	VCATCH	Vector catch flag: 1 = vector catch occurred 0 = no vector catch occurred. When the VCATCH flag is set, a flag in one of the local fault status registers is also set to indicate the type of fault.
[2]	DWTTRAP	*Data Watchpoint and Trace* (DWT) flag: 1 = DWT match 0 = no DWT match. The processor stops at the current instruction or at the next instruction.
[1]	BKPT	BKPT flag: 1 = BKPT instruction execution 0 = no BKPT instruction execution. The BKPT flag is set by a BKPT instruction in flash patch code, and also by normal code. Return PC points to breakpoint containing instruction.
[0]	HALTED	Halt request flag: 1 = halt requested by NVIC, including step. The processor is halted on the next instruction. 0 = no halt request.

Memory Manage Fault Address Register

Use the Memory Manage Fault Address Register to read the address of the location that caused a Memory Manage Fault.

The register address, access type, and Reset state are:

Address `0xE000ED34`

Access Read/write

Reset state Unpredictable

Table 8-26 describes the field of the Memory Manage Fault Address Register.

Table 8-26 Bit functions of the Memory Manage Fault Address Register

Field	Name	Definition
[31:0]	ADDRESS	Mem Manage fault address field. ADDRESS is the data address of a faulted load or store attempt. When an unaligned access faults, the address is the actual address which faulted. Because an access can be split into multiple parts, each aligned, this address can be any offset in the range of the requested size. Flags in the Memory Manage Fault Status Register indicate the cause of the fault. See *Memory Manage Fault Status Register* on page 8-30.

Bus Fault Address Register

Use the Bus Fault Address Register to read the address of the location that generated a Bus Fault.

The register address, access type, and Reset state are:

Address `0xEEE0ED38`
Access Read/write
Reset state Unpredictable

Table 8-27 describes the fields of the Bus Fault Address Register.

Table 8-27 Bit functions of the Bus Fault Address Register

Field	Name	Definition
[31:0]	ADDRESS	Bus fault address field. ADDRESS is the data address of a faulted load or store attempt. When an unaligned access faults, the address is the address requested by the instruction, even if that is not the address which faulted. Flags in the Bus Fault Status Register indicate the cause of the fault. See *Bus Fault Status Register* on page 8-31.

Software Trigger Interrupt Register

Use the Software Trigger Interrupt Register to pend an interrupt to trigger.

The register address, access type, and Reset state are:

Address `0xE000EF00`
Access Write-only
Reset state `0x00000000`

Figure 8-21 on page 8-39 shows the fields of the Software Trigger Interrupt Register.

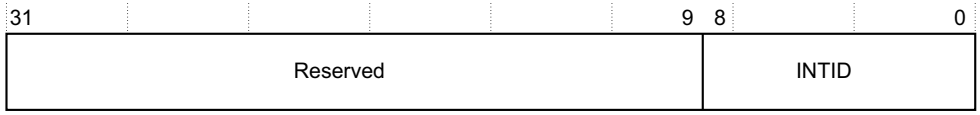


Figure 8-21 Software Trigger Interrupt Register bit assignments

Table 8-28 describes the fields of the Software Trigger Interrupt Register.

Table 8-28 Software Trigger Interrupt Register bit assignments

Field	Name	Definition
[31:9]	-	Reserved.
[8:0]	INTID	Interrupt ID field. Writing a value to the INTID field is the same as manually pending an interrupt by setting the corresponding interrupt bit in an Interrupt Set Pending Register.

8.3 Level versus pulse interrupts

The processor supports both level and pulse interrupts. A level interrupt is held asserted until it is cleared by the ISR accessing the device. A pulse interrupt is a variant of an edge model. The edge must be sampled on the rising edge of the Cortex-M3 clock, **HCLK**, instead of being asynchronous.

For level interrupts, if the signal is not deasserted before the return from the interrupt routine, the interrupt repends and re-activates. This is particularly useful for FIFO and buffer-based devices because it ensures that they drain either by a single ISR or by repeated invocations, with no extra work. This means that the device holds the signal in assert until the device is empty.

A pulse interrupt can be reasserted during the ISR so that the interrupt can be pended and active at the same time. The application design must ensure that a second pulse does not arrive before the first pulse is activated. The second pend has no affect because it is already pended. However, if the interrupt is asserted for at least one cycle, the NVIC latches the pend bit. When the ISR activates, the pend bit is cleared. If the interrupt asserts again while it is activated, it can latch the pend bit again.

Pulse interrupts are mostly used for external signals and for rate or repeat signals.

Chapter 9
Memory Protection Unit

This chapter describes the processor *Memory Protection Unit* (MPU). It contains the following sections:

- *About the MPU* on page 9-2
- *MPU programmer's model* on page 9-3
- *Interrupts and updating the MPU* on page 9-19
- *MPU access permissions* on page 9-13
- *MPU aborts* on page 9-15
- *Updating an MPU region* on page 9-16.

9.1 About the MPU

The *Memory Protection Unit* (MPU) is a component for memory protection. The processor supports the standard ARMv7 *Protected Memory System Architecture* (PMSA) model. The MPU provides full support for:

- protection regions
- overlapping protection regions
- access permissions
- exporting memory attributes to the system.

MPU mismatches and permission violations invoke the programmable-priority MemManage fault handler. For more information, see *Memory Manage Fault Address Register* on page 8-37.

You can use the MPU to:

- enforce privilege rules
- separate processes
- enforce access rules.

9.2 MPU programmer's model

This sections describes the registers that control the MPU. It contains the following:

- *Summary of the MPU registers*
- *Description of the MPU registers*.

9.2.1 Summary of the MPU registers

Table 9-1 provides a summary of the MPU registers.

Table 9-1 MPU registers

Name of register	Type	Address	Reset value	Page
MPU Type Register	Read Only	0xE000ED90	0x00000800	page 9-3
MPU Control Register	Read/Write	0xE000ED94	0x00000000	page 9-4
MPU Region Number register	Read/Write	0xE000ED98	-	page 9-6
MPU Region Base Address register	Read/Write	0xE000ED9C	-	page 9-7
MPU Region Attribute and Size register(s)	Read/Write	0xE000EDA0	-	page 9-8
MPU Alias 1 Region Base Address register	Alias of D9C	0xE000EDA4	-	page 9-11
MPU Alias 1 Region Attribute and Size register	Alias of DA0	0xE000EDA8	-	page 9-11
MPU Alias 2 Region Base Address register	Alias of D9C	0xE000EDAC	-	page 9-11
MPU Alias 2 Region Attribute and Size register	Alias of DA0	0xE000EDB0	-	page 9-11
MPU Alias 3 Region Base Address register	Alias of D9C	0xE000EDB4	-	page 9-11
MPU Alias 3 Region Attribute and Size register	Alias of DA0	0xE000EDB8	-	page 9-11

9.2.2 Description of the MPU registers

This section contains a description of the MPU registers.

MPU Type Register

Use the MPU Type Register to see how many regions the MPU supports. Read bits [15:8] to determine if an MPU is present.

The register address, access type, and Reset state are:

Address 0xE000ED90

Access Read-only

Reset state `0x00000800`

Figure 9-1 shows the fields of the MPU Type Register.

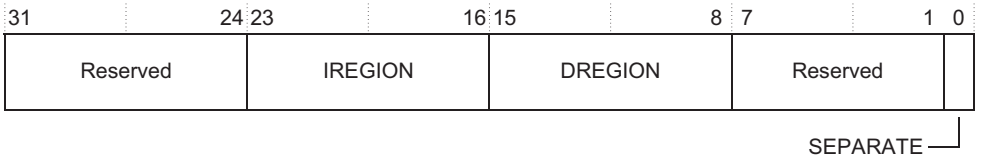


Figure 9-1 MPU Type Register bit assignments

Table 9-2 describes the fields of the MPU Type Register.

Table 9-2 MPU Type Register bit assignments

Field	Name	Definition
[31:24]	-	Reserved.
[23:16]	IREGION	Because the processor core uses only a unified MPU, IREGION always contains `0x00`.
[15:8]	DREGION	Number of supported MPU regions field. DREGION contains `0x08` if the implementation contains an MPU indicating eight MPU regions, otherwise it contains `0x00`.
[7:0]	-	Reserved.
[0]	SEPARATE	Because the processor core uses only a unified MPU, SEPARATE is always 0.

MPU Control Register

Use the MPU Control Register to:

- enable the MPU
- enable the default memory map (background region)
- enable the MPU when in Hard Fault, NMI, and FAULTMASK escalated handlers.

When the MPU is enabled, at least one region of the memory map must be enabled for the MPU to function unless the PRIVDEFENA bit is set. If the PRIVDEFENA bit is set and no regions are enabled, then only privileged code can operate.

When the MPU is disabled, the default address map is used, as if no MPU is present.

When the MPU is enabled, only the system partition and vector table loads are always accessible. Other areas are accessible based on regions and whether PRIVDEFENA is enabled.

Unless HFNMIENA is set, the MPU is not enabled when the exception priority is –1 or –2. These priorities are only possible when in Hard fault, NMI, or when FAULTMASK is enabled. The HFNMIENA bit is used to enable the MPU when operating with these two priorities.

The register address, access type, and Reset state are:

Address `0xE000ED94`

Access Read/write

Reset state `0x00000000`

Figure 9-2 shows the fields of the MPU Control Register.

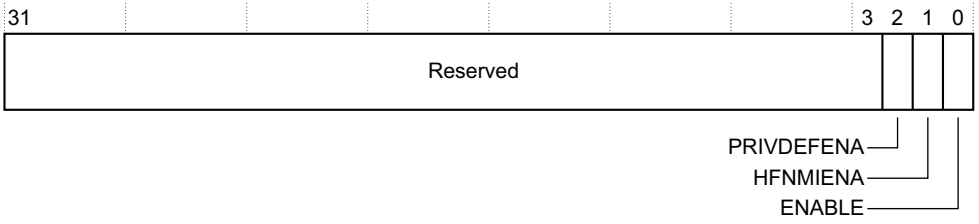


Figure 9-2 MPU Control Register bit assignments

Table 9-3 describes the fields of the MPU Control Register.

Table 9-3 MPU Control Register bit assignments

Field	Name	Definition
[31:2]	-	Reserved.
[2]	PRIVDEFENA	This bit enables the default memory map for privileged access, as a background region, when the MPU is enabled. The background region acts as if it was region number 1 before any settable regions. Any region which is set up overlays this default map, and overrides it. If this bit = 0, the default memory map is disabled, and memory not covered by a region faults. When the MPU is enabled and PRIVDEFENA is enabled, the default memory map is as described in Chapter 4 *Memory Map*. This applies to memory type, XN, cache and shareable rules. However, this only applies to privileged mode (fetch and data access). User mode code faults unless a region has been set up for its code and data. When the MPU is disabled, the default map acts on both privileged and user mode code. XN and SO rules always apply to the System partition whether this enable is set or not. If the MPU is disabled, this bit is ignored. Reset clears the PRIVDEFENA bit.
[1]	HFNMIENA	This bit enables the MPU when in Hard Fault, NMI, and FAULTMASK escalated handlers. If this bit = 1 and the ENABLE bit = 1, the MPU is enabled when in these handlers. If this bit = 0, the MPU is disabled when in these handlers, regardless of the value of ENABLE. If this bit =1 and ENABLE = 0, behavior is unpredictable. Reset clears the HFNMIENA bit.
[0]	ENABLE	MPU enable bit: 1 = enable MPU 0 = disable MPU. Reset clears the ENABLE bit.

MPU Region Number Register

Use the MPU Region Number Register to select which protection region is accessed. Then write to the MPU Region Base Address Register or the MPU Attributes and Size Register to configure the characteristics of the protection region.

The register address, access type, and Reset state are:

Address `0xE000ED98`

Access Read/write

Reset state Unpredictable

Figure 9-3 on page 9-7 shows the fields of the MPU Region Number Register.

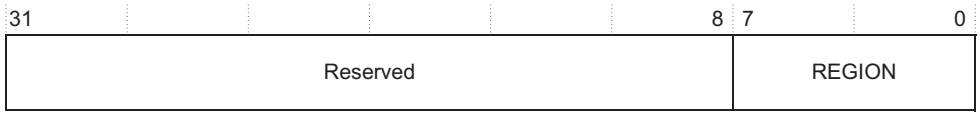


Figure 9-3 MPU Region Number Register bit assignments

Table 9-4 describes the fields of the MPU Region Number Register.

Table 9-4 MPU Region Number Register bit assignments

Field	Name	Definition
[31:8]	-	Reserved.
[7:0]	REGION	Region select field. Selects the region to operate on when using the Region Attribute and Size Register and the Region Base Address Register. It must be written first except when the address VALID + REGION fields are written, which overwrites this.

MPU Region Base Address Register

Use the MPU *Region Base Address Register* to write the base address of a region. The Region Base Address Register also contains a REGION field that you can use to override the REGION field in the MPU Region Number Register, if the VALID bit is set.

The Region Base Address register sets the base for the region. It is aligned by the size. So, a 64-KB sized region must be aligned on a multiple of 64KB, for example, `0x00010000`, `0x00020000`, and so on.

The region always reads back as the current MPU region number. Valid always reads back as 0. Writing with VALID = 1 and REGION = n changes the region number to n. This is a short-hand way to write the MPU Region Number Register.

This register is unpredictable if accessed other than as a word.

The register address, access type, and Reset state are:

Address `0xE000ED9C`
Access Read/write
Reset state Unpredictable

Figure 9-4 on page 9-8 shows the fields of the MPU Region Base Address Register.

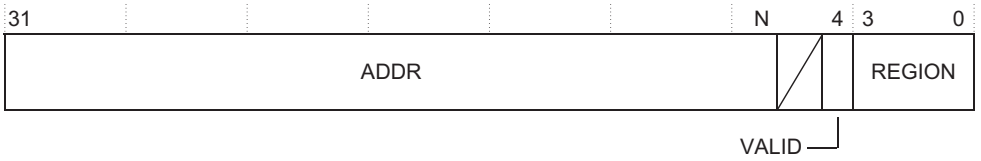


Figure 9-4 MPU Region Base Address Register bit assignments

Table 9-5 describes the fields of the MPU Region Base Address Register.

Table 9-5 MPU Region Base Address Register bit assignments

Field	Name	Definition
[31:N]	ADDR	Region base address field. The value of N depends on the region size, so that the base address is aligned according to an even multiple of size. The power of 2 size specified by the SZENABLE field of the MPU Region Attribute and Size Register defines how many bits of base address are used.
[4]	VALID	MPU Region Number valid bit: 1 = MPU Region Number Register is overwritten by bits 3:0 (the REGION value). 0 = MPU Region Number Register remains unchanged and is interpreted.
[3:0]	REGION	MPU region override field.

MPU Region Attribute and Size Register

Use the MPU Region Attribute and Size Register to control the MPU access permissions. The register is made up of two part registers, each of halfword size. These can be accessed using the individual size, or they can both be simultaneously accessed using a word operation.

The sub-region disable bits are unpredictable for region sizes of 32 bytes, 64 bytes, and 128 bytes.

The register address, access type, and Reset state are:

Address `0xE000EDA0`

Access Read/write

Reset state Unpredictable

Figure 9-5 on page 9-9 shows the fields of the MPU Region Attribute and Size Register.

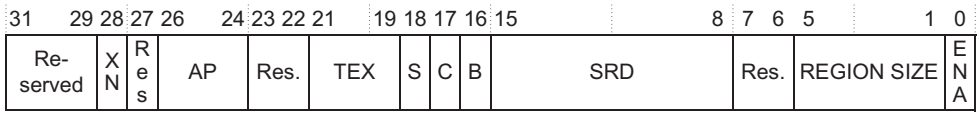


Figure 9-5 MPU Region Attribute and Size Register bit assignments

Table 9-6 describes the fields of the MPU Region Attribute and Size Register. For more information, see *MPU access permissions* on page 9-13.

Table 9-6 MPU Region Attribute and Size Register bit assignments

Field	Name	Definition
[31:29]	-	Reserved.
[28]	XN	Instruction access disable bit: 1 = disable instruction fetches 0 = enable instruction fetches.
[27]	-	Reserved.
[26:24]	AP	Data access permission field:

Value	Privileged permissions	User permissions
b000	no access	no access
b001	read/write	no access
b010	read/write	read-only
b011	read/write	read/write
b100	reserved	reserved
b101	read-only	no access
b110	read-only	read-only
b111	read-only.	read-only.

[23:22]	-	Reserved.
[21:19]	TEX	Type extension field.
[18]	S	Shareable bit: 1 = shareable 0 = not shareable.
[17]	C	Cacheable bit: 1 = cacheable 0 = not cacheable.

Table 9-6 MPU Region Attribute and Size Register bit assignments (continued)

Field	Name	Definition
[16]	B	Bufferable bit: 1 = bufferable 0 = not bufferable.
[15:8]	SRD	Sub-region disable field. Setting an SRD bit disables the corresponding sub-region. Regions are split into eight equal-sized sub-regions. Sub-regions are not supported for region sizes of 128 bytes and below. For more information, see *Sub-Regions* on page 9-12.
[7:6]	-	Reserved.
[5:1]	REGION SIZE	MPU Protection Region Size Field. See Table 9-7 on page 9-10.
[0]	SZENABLE	Region enable bit.

For information about access permission, see *MPU access permissions* on page 9-13.

Table 9-7 MPU protection region size field

Region	Size
b00000	Reserved
b00001	Reserved
b00010	Reserved
b00011	Reserved
b00100	32B
b00101	64B
b00110	128B
b00111	256B
b01000	512B
b01001	1KB
b01010	2KB
b01011	4KB
b01100	8KB
b01101	16KB

Table 9-7 MPU protection region size field

Region	Size
b01110	32KB
b01111	64KB
b10000	128KB
b10001	256KB
b10010	512KB
b10011	1MB
b10100	2MB
b10101	4MB
b10110	8MB
b10111	16MB
b11000	32MB
b11001	64MB
b11010	128MB
b11011	256MB
b11100	512MB
b11101	1GB
b11110	2GB
b11111	4GB

9.2.3 Accessing the MPU using the alias registers

You can optimize the loading speed of the MPU registers using register aliasing. There are three sets of NVIC alias registers. These are described in *NVIC register descriptions* on page 8-7.

The aliases access the registers in exactly the same way, and they exist to enable the use of sequential writes (STM) to update between one and four regions. This is used when disable/change/enable is not required.

You cannot use these aliases to read the contents of the regions because the region number must be written.

An example code sequence for updating four regions is

```
; R1 = 4 region pairs from process control block (8 words)
MOV     R0, #NVIC_BASE
ADD     R0, #MPU_REG_CTRL
LDM     R1, [R2-R9] ; load region information for 4 regions
STM     R0, [R2-R9] ; update all 4 regions at once
```

Note

You can normally use the `memcpy()` function in a C/C++ compiler for this sequence. However, you must verify that the compiler uses word transfers.

9.2.4 Sub-Regions

The eight *Sub-Region Disable* (SRD) bits of the Region Attribute and Size Register are used to divide a region into eight equal sized units based on the region size. This enables selectively disabling some of the 1/8th sub-regions. The least significant bit affects the first 1/8th sub-region, and the most significant bits affects the last 1/8th sub-region. A disabled sub-region enables any other region overlapping that range to be matched instead. If no other region overlaps the sub-region, the default behavior is used, no match – a fault. Sub-regions cannot be used with the three smallest regions of size: 32, 64, and 128. If these sub-regions are used, the results are UNPREDICTABLE.

Example of SRD use

Two regions with the same base address overlap. One region is 64KB, and the other is 512KB. The bottom 64KB of the 512KB region is disabled so that the attributes from the 64KB apply. This is achieved by setting SRD for the 512KB region to b11111110.

9.3 MPU access permissions

This section describes the MPU access permissions. The access permission bits, TEX, C, B, AP, and XN, of the Region Access Control Register (see *MPU Region Attribute and Size Register* on page 9-8) control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, then a permission fault is raised.

Table 9-8 describes the TEX, C, and B encoding.

Table 9-8 TEX, C, B encoding

TEX	C	B	Description	Memory type	Region shareability
b000	0	0	Strongly ordered	Strongly ordered	Shareable
b000	0	1	Shared device	Device	Shareable
b000	1	0	Outer and inner write-through No write allocate	Normal	S
b000	1	1	Outer and inner write-back No write allocate	Normal	S
b001	0	0	Outer and inner noncacheable	Normal	S
b001	0	1	Reserved	Reserved	Reserved
b001	1	0	Implementation-defined		
b001	1	1	Outer and inner write-back Write and read allocate	Normal	S
b010	1	X	Nonshared device	Device	Not shareable
b010	0	1	Reserved	Reserved	Reserved
b010	1	X	Reserved	Reserved	Reserved
b1BB	A	A	Cached memory BB = outer policy. AA = inner policy	Normal	S

Note

In Table 9-8, ‘S’ is the S bit[2] from the MPU Region Attributes and Size Register.

Table 9-9 describes the cache policy for memory attribute encoding.

Table 9-9 Cache policy for memory attribute encoding

Memory attribute encoding (AA and BB)	Cache policy
00	Non-cacheable
01	Write back, write and read allocate
10	Write through, no write allocate
11	Write back, no write allocate

Table 9-10 describes the AP encoding.

Table 9-10 AP encoding

AP[2:0]	Privileged permissions	User permissions	Descriptions
000	No access	No access	All accesses generate a permission fault
001	Read/write	No access	Privileged access only
010	Read/write	Read only	Writes in user mode generate a permission fault
011	Read/write	Read/write	Full access
100	Unpredictable	Unpredictable	Reserved
101	Read only	No access	Privileged read only
110	Read only	Read only	Privileged/user read only
111	Read only	Read only	Privileged/user read only

Table 9-11 describes the XN encoding.

Table 9-11 XN encoding

XN	Description
0	All instruction fetches enabled
1	No instruction fetches enabled

9.4 MPU aborts

For information about MPU aborts, see *Memory Manage Fault Address Register* on page 8-37.

9.5 Updating an MPU region

There are three registers consisting of three memory mapped words that are used to program the MPU regions. These are part registers that can be programmed and accessed individually. This means that existing ARMv6, ARMv7 and CP15 code can be ported. This replaces MRC and MCR with LDRx and STRx operations.

It is also possible to access these registers as three words, and program them using only two words. Aliases are provided to enable programming a set of regions simultaneously using an STM instruction.

9.5.1 Updating an MPU region using CP15 equivalent code

Using CP15 equivalent code:

```
; R1 = region number

; R2 = size/enable

; R3 = attributes

; R4 = address

MOV     R0,#NVIC_BASE

ADD     R0,#MPU_REG_CTRL

STR     R1,[R0,#0]       ; region number

STR     R4,[R0,#4]       ; address

STRH    R2,[R0,#8]       ; size and enable

STRH    R3,[R0,#10]      ; attributes
```

Note

If interrupts could pre-empt during this period, they could be affected by this region. This means that the region must be disabled, written, and then enabled. This is usually not necessary for a context switcher, but would be necessary if updated elsewhere.

```
; R1 = region number

; R2 = size/enable

; R3 = attributes

; R4 = address

MOV     R0,#NVIC_BASE
```



```
ADD     R0,#MPU_REG_CTRL
STR     R1,[R0,#0]       ; region number
BIC     R2,R2, #1        ; disable
STRH    R2,[R0,#8]       ; size and enable
STR     R4,[R0,#4]       ; address
STRH    R3,[R0,#10]      ; attributes
ORR     R2,#1            ; enable
STRH    R2,[R0,#8]       ; size and enable
```

DMB/DSB is not necessary because the Private Peripheral Bus is a strongly ordered memory area. However, a DSB is necessary before the effect of the MPU takes place, such as the end of a context switcher.

An ISB is necessary if the code used to program the MPU region or regions is entered using a branch or call. If the code is entered using a return from exception, or by taking an exception, then an ISB is not necessary.

9.5.2 Updating an MPU region using two or three words

It is possible to program directly using two or three words, depending on how the information is divided:

```
; R1 = region number
; R2 = address
; R3 = size, attributes in one
MOV     R0,#NVIC_BASE
ADD     R0,#MPU_REG_CTRL
STR     R1,[R0,#0]       ; region number
STR     R2,[R0,#4]       ; address
STR     R3,[R0,#8]       ; size, attributes
```

This can be optimized using an STM:

```
; R1 = region number
; R2 = address
; R3 = size, attributes in one
```

```
MOV     R0,#NVIC_BASE

ADD     R0,#MPU_REG_CTRL

STM     R0,{R1-R3}      ; region number, address, size, and attributes
```

This can be done in two words for pre-packed information. This means that the base address register contains the region number in addition to a region-valid bit. This is useful when the data is statically packed, for example in a boot list or a PCB.

```
; R1 = address and region number in one

; R2 = size and attributes in one

MOV     R0,#NVIC_BASE

ADD     R0,#MPU_REG_CTRL

STR     R1,[R0,#4]      ; address and region number

STR     R2,[R0,#8]      ; size and attributes
```

This can be optimized using an STM:

```
; R1 = address and region number in one

; R2 = size and attributes in one

MOV     R0,#NVIC_BASE

ADD     R0,#MPU_REG_CTRL

STM     R0,{R1-R2}      ; address, region number, size
```

For information about interrupts and updating the MPU, see *Interrupts and updating the MPU* on page 9-19.

9.6 Interrupts and updating the MPU

An MPU region can contain critical data. This is because it takes more than one bus transaction to update. This is normally two words. As a result, it is not “thread safe”. That is, an interrupt can split the two words, leaving the region with incoherent information. There are two different issues:

- An interrupt can come in that would also update the MPU. This is not only a read-modify-write issue, it also affects cases where the interrupt routine is guaranteed not to modify the same region. This is because the programming relies on the region number being written into a register so that it knows which region to update. So, interrupts must be disabled around each update routine in this case.
- An interrupt can come in that would use the region being updated or would be affected because only the base or size fields had been updated. If the new size field is changed, but the base is not, the base+new_size might overlap into an area normally handled by another region. In this case, the disable-modify-enable approach is required.

But, for standard OS context switch code, which would be changing user regions, there is no risk, as these regions would be preset to be user privilege and a user area address. This means that even an interrupt would cause no side effect. Therefore the disable/enable code is not required nor is interrupt disable.

The most common approach is to only program the MPU from two places: boot code and context switcher. If those are the only two places, and the context switcher is only updating user regions, then disable is not required because the context switcher is already a critical region and the boot code runs with interrupts disabled.

Chapter 10
Core Debug

This chapter describes how to debug and test the processor. It contains the following sections:

- *About core debug* on page 10-2
- *Core debug registers* on page 10-3
- *Core debug access example* on page 10-12
- *Using application registers in core debug* on page 10-13.

10.1 About core debug

Core debug is accessed through the core debug registers. Debug access to these registers is by means of the AHB-AP port, see *AHB Access Port* on page 11-35. The processor can access these registers directly over the internal *Private Peripheral Bus* (PPB).

Table 10-1 shows the core debug registers.

Table 10-1 Core debug registers

Address	Type	Reset Value	Description
0xE000EDF0	Read/Write	0x00000000[a]	Debug Halting Control and Status Register
0xE000EDF4	Write-only	-	Debug Core Register Selector Register
0xE000EDF8	Read/Write	-	Debug Core Register Data Register
0xE000EDFC	Read/Write	0x00000000[b]	Debug Exception and Monitor Control Register.

a. Bits 5, 3, 2, 1, 0 are reset by **PORESETn**. Bit[1] is also reset by **SYSRESETn** and writing a 1 to the **VECTRESET** bit of the Application Interrupt and Reset Control Register.
b. Bits 16,17,18,19 are also reset by **SYSRESETn** and writing a 1 to the **VECTRESET** bit of the Application Interrupt and Reset Control Register.

Also used is the Debug Fault Status Register see *Debug Fault Status Register* on page 8-36 for more information

10.1.1 Halt mode debugging

The debugger can halt the core by setting the **C_DEBUGEN** and **C_HALT** bits of the Debug Halting Control and Status Register. The core will acknowledge once halted by setting the **S_HALT** bit of the Debug Halting Control and Status Register.

The core can be single stepped by halting the core, setting the **C_STEP** bit to 1, and then clearing the **C_HALT** bit to 0. The core will acknowledge completion of the step and re-halt by setting the **S_HALT** bit of the Debug Halting Control and Status Register.

10.1.2 Exiting core debug

The core can exit Halting debug by clearing the **C_DEBUGEN** bit in the Debug Halting and Status Register.

10.2 Core debug registers

The registers that provide debug operations are:

- *Debug Halting Control and Status Register*
- *Core debug registers*
- *Debug Core Register Data Register* on page 10-8
- *Debug Exception and Monitor Control Register* on page 10-8.

10.2.1 Debug Halting Control and Status Register

The purpose of the *Debug Halting Control and Status Register* (DHCSR) is to:

- provide status information about the state of the processor
- enable core debug
- halt and step the processor.

The DHCSR:

- is a 32-bit read/write register
- address is `0xE000EDF0`.

Note

The DHCSR is only reset from a system reset, including power on. Bit 16 of DHCSR is UNPREDICTABLE on reset.

Figure 10-1 on page 10-4 shows the arrangement of bits in the register.

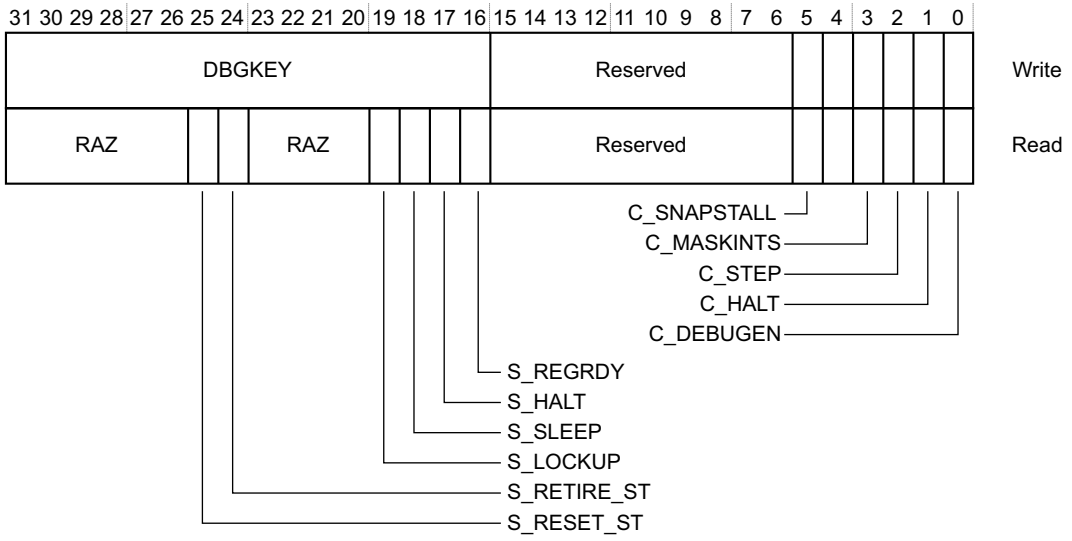


Figure 10-1 Debug Halting Control and Status Register format

Table 10-2 shows the bit functions of the Debug ID Register.

Table 10-2 Debug Halting Control and Status Register

Bit range	R/W	Field name	Function
[31:16]	W	**DBGKEY**	Debug Key. `0xA05F` must be written anytime this register is written. Reads back as status bits [25:16]. If not written as Key, the write operation is ignored and no bits are written into the register.
[31:26]	-	-	Reserved, RAZ.
[25]	R	**S_RESET_ST**	Indicates that the core has been reset, or is now being reset, since the last time this bit was read. This a sticky bit that clears on read. So, reading twice and getting 1 then 0 means it was reset in the past. Reading twice and getting 1 both times means that it is being reset now (held in reset still).
[24]	R	**S_RETIRE_ST**	Indicates that an instruction has completed since last read. This is a sticky bit that clears on read. This is used to determine if the core is stalled on a load/store or fetch.
[23:20]	-	-	Reserved, RAZ.
[19]	R	**S_LOCKUP**	Reads as one if the core is running (not halted) and a lockup condition is present.

Table 10-2 Debug Halting Control and Status Register (continued)

Bit range	R/W	Field name	Function
[18]	R	**S_SLEEP**	Indicates that the core is sleeping (WFI, WFE or SLEEP-ON-EXIT). Must use **C_HALT** to gain control or wait for interrupt to wake-up. For more information on SLEEP-ON-EXIT see Table 7-1 on page 7-3.
[17]	R	**S_HALT**	Indicates core status on Halt.
[16]	R	**S_REGRDY**	Register Read/Write on the Debug Core Register Selector register is available. Last transfer is complete.
[15:6]	-	-	Reserved.
[5]	R/W	**C_SNAPSTALL**	If the core is stalled on a load/store operation the stall ceases and the instruction is forced to complete. This enables Halting debug to gain control of the core. It can only be set if: **C_DEBUGEN** = 1 **C_HALT** = 1 **S_RETIRE_ST** is read as 0 by the core. This indicates that no instruction has advanced. This prevents misuse. The bus state is Unpredictable when this is used. Core stalls on load/store operations can be detected using **S_RETIRE**.
[4]	-	-	Reserved.
[3]	R/W	**C_MASKINTS**	Mask interrupts when stepping or running in halted debug. Does not affect NMI, which is not maskable. Must only be modified when the processor is halted (S_HALT == 1).
[2]	R/W	**C_STEP**	Steps the core in halted debug. When **C_DEBUGEN** = 0, this bit has no effect. Must only be modified when the processor is halted (S_HALT == 1).
[1]	R/W	**C_HALT**	Halts the core. This bit is set automatically when the core Halts. For example Breakpoint. This bit clears on core reset. This bit can only be written if **C_DEBUGEN** is 1, otherwise it is ignored. When setting this bit to 1, **C_DEBUGEN** must also be written to 1 in the same value (value[1:0] is 2'b11). The core can halt itself, but only if **C_DEBUGEN** is already 1 and only if it writes with b11).
[0]	R/W	**C_DEBUGEN**	Enables debug. This can only be written by AHB-AP and not by the core. It is ignored when written by the core, which cannot set or clear it. The core must write a 1 to it when writing **C_HALT** to halt itself.

If not enabled for Halting mode, **C_DEBUGEN** = 1, all other fields are disabled.

This register is not reset on a system reset. It is reset by a power-on reset. However, the **C_HALT** bit always clears on a system reset.

To halt on a reset the following bits must be enabled:

- bit [0], **VC_CORERESET**, of the Debug Exception and Monitor Control Register
- bit [0],**C_DEBUGEN**, of the Debug Halting Control and Status Register.

Note

Writes to this register in any size other than word are Unpredictable. It is acceptable to read in any size, and this can be used to avoid or intentionally change a sticky bit.

10.2.2 Debug Core Selector Register

The purpose of the *Debug Core Selector Register* (DCSR) is to select the processor register to transfer data to or from.

The DCSR:

- is a 17-bit write-only register
- address is `0xE000EDF4`.

Figure 10-2 shows the arrangement of bits in the register.

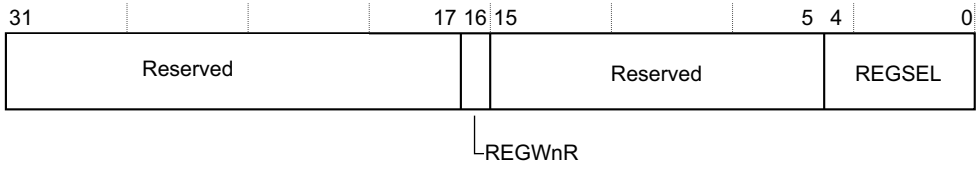


Figure 10-2 Debug Core Selector Register format

Table 10-3 shows the bit functions of the Debug Core Selector Register.

Table 10-3 Debug Core Selector Register

Bit range	R/W	Field name	Function
[31:17]	-	-	Reserved
[16]	W	**REGWnR**	Write = 1 Read = 0
[15:5]	-	-	-
[4:0]	W	**REGSEL**	0b00000 = R0 0b00001 = R1 … 0b01111 = R15 0b10000 = xPSR/ Flags 0b10001 = MSP (Main SP) 0b10010 = PSP (Process SP) 0b10011 = RAZ/WI 0b10100 = CONTROL/FAULTMASK/BASEPRI/PRIMASK (packed into 4 bytes of words. CONTROL is MSB (31:24) 0b1xxxx = Reserved

This write-only register generates a handshake to the core to transfer data to or from Debug Core Register Data Register and the selected register. Until this core transaction is complete, bit [16], **S_REGRDY**, of the DHCSR is 0.

Note

- Writes to this register in any size but word are Unpredictable.
- PSR registers are fully accessible this way, whereas some read as 0 when using MRS instructions.
- All bits can be written, but some combinations cause a fault when execution is resumed.
- IT might be written and behaves as though in an IT block.
- ICI can be written, though invalid values or when not used on an LDM/STM causes a fault, as would on return from exception. Changing ICI from a value to 0 causes the underlying LDM/STM to start, not continue.

10.2.3 Debug Core Register Data Register

The purpose of the *Debug Core Register Data Register* (DCRDR) is to hold data for reading and writing registers to and from the processor.

The DCRDR:

- is a 32-bit read/write register
- address `0xE000EDF8.`

This is the data value written to the register selected by the Debug Register Selector Register.

When the processor receives a request from the Debug Core Register Selector, this register is read or written by the processor using a normal load-store unit operation.

If core register transfers are not being performed, this register can be used by software-based debug monitors for communication in non-halting debug. For example, OS RSD and Real View Monitor. This enables flags and bits to be used to acknowledge state and indicate if commands have been accepted to, replied to, or accepted and replied to.

10.2.4 Debug Exception and Monitor Control Register

The purpose of the *Debug Exception and Monitor Control Register* (DEMCR) is for:

- Vector catching. That is, to cause debug entry when a specified vector is committed for execution.
- Debug monitor control.

The DEMCR:

- is a 32-bit read/write register
- address `0xE000EDFC`

Figure 10-2 on page 10-6 shows the arrangement of bits in the register.

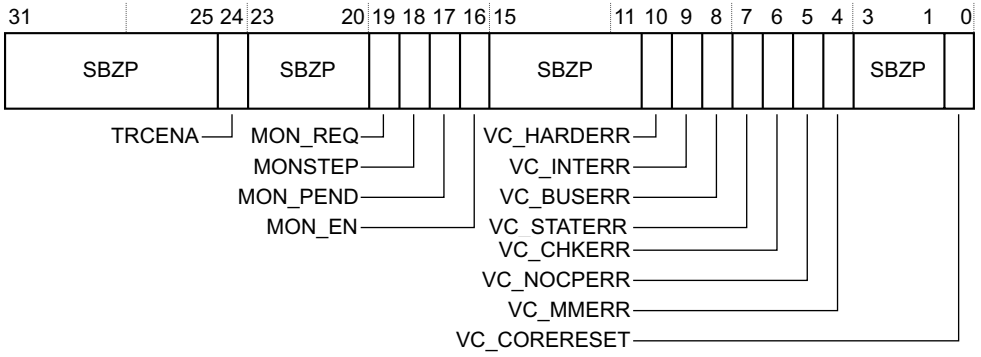


Figure 10-3 Debug Exception and Monitor Control Register format

Table 10-4 shows the bit functions of the Debug Exception and Monitor Control Register.

Table 10-4 Debug Exception and Monitor Control Register

Bits	R/W	Name	Function
[31:25]	-	-	Reserved, SBZP
[24]	R/W	**TRCENA**	This bit must be set to 1 to enable use of the trace and debug blocks: • *Data Watchpoint and Trace* (DWT) • *Instruction Trace Macrocell* (ITM) • *Embedded Trace Macrocell* (ETM) • *Trace Port Interface Unit* (TPIU). This enables control of power usage unless tracing is required. This can be enabled by the application, for ITM use, or by a debugger.
[23:20]	-	-	Reserved, SBZP
[19]	R/W	**MON_REQ**[a]	This enables the monitor to identify how it wakes up: 1 = woken up by MON_PEND 0 = woken up by debug exception.
[18]	R/W	**MONSTEP**[a]	When **MON_EN** = 1, this is used to step the core. When **MON_EN** = 0, this bit is ignored. This is the equivalent to **C_STEP**. Interrupts are only stepped according to the priority of the monitor and settings of PRIMASK, FAULTMASK, or BASEPRI.

Table 10-4 Debug Exception and Monitor Control Register (continued)

Bits	R/W	Name	Function
[17]	R/W	**MON_PEND**[a]	Pend the monitor to activate when priority permits. This can be used to wake up the monitor through the AHB-AP port. This is the equivalent to **C_HALT** for Monitor debug. This register does not reset on a system reset. It is only reset by a power-on reset. The debug monitor must be enabled by software in the reset handler or later, or by the DAP.
[16]	R/W	**MON_EN**[a]	Enable the debug monitor. When enabled, the System handler priority register controls its priority level. If disabled, then all debug events go to Hard fault. **C_DEBUGEN** in the Debug Halting Control and Statue register overrides this bit. Vector catching is semi-synchronous. When a matching event is seen, a Halt is requested. Because the processor can only halt on an instruction boundary, it must wait until the next instruction boundary. As a result, it stops on the first instruction of the exception handler. However, two special cases exist when a vector catch has triggered: • If a fault is taken during vectoring, vector read or stack push error, the halt occurs on the corresponding fault handler, for the vector error or stack push. • If a late arriving interrupt comes in during vectoring, it is not taken. That is, an implementation that supports the late arrival optimization must suppress it in this case.
[15:11]	-	-	Reserved, SBZP
[10]	R/W	**VC_HARDERR**[b]	Debug trap on Hard Fault (see section 8.3).
[9]	R/W	**VC_INTERR**[b]	Debug Trap on interrupt/exception service errors (see section 8.3). These are a subset of other faults and catches before BUSERR or HARDERR.
[8]	R/W	**VC_BUSERR**[b]	Debug Trap on normal Bus error (see section 8.3).
[7]	R/W	**VC_STATERR**[b]	Debug trap on Usage Fault state errors (see section 8.3).
[6]	R/W	**VC_CHKERR**[b]	Debug trap on Usage Fault enabled checking errors (see section 8.3).
[5]	R/W	**VC_NOCPERR**[b]	Debug trap on Usage Fault access to Coprocessor which is not present or marked as not present in CAR register.
[4]	R/W	**VC_MMERR**[b]	Debug trap on Memory Management faults (see section 8.3).
[3:1]	-	-	Reserved, SBZP
[0]	R/W	**VC_CORERESET**[b]	Reset Vector Catch. Halt running system if Core reset occurs.

a. This bit clears on a Core Reset.
b. Only usable when **C_DEBUGEN** = 1.

This register is used to manage exception behavior under debug.

Vector catching is only available to halting debug. The upper halfword is for monitor controls and the lower halfword is for halting exception support.

This register is not reset on a system reset.

This register is reset by a power-on reset. Bits [19:16] are always cleared on a core reset. The debug monitor is enabled by software in the reset handler or later, or by the AHB-AP port.

Vector catching is semi-synchronous. When a matching event is seen, a Halt is requested. Because the processor can only halt on an instruction boundary, it must wait until the next instruction boundary. As a result, it stops on the first instruction of the exception handler. However, two special cases exist when a vector catch has triggered:

1. If a fault is taken during a vector read or stack push error the halt occurs on the corresponding fault handler for the vector error or stack push.

2. If a late arriving interrupt detected during a vector read or stack push error it is not taken. That is, an implementation which supports the late arrival optimization must suppress it in this case.

10.3 Core debug access example

If you want to halt the processor and write a value into one of the registers, perform the following sequence:

1. Write `0xA05F0003` to the Debug Halting Control and Status register. This enables debug and halts the core.
2. Wait for the **S_HALT** bit of the Debug Halting and Status Register to be set. This indicates that the core is halted.
3. Write the value that you want to be written to the Debug Core Register Data Register.
4. Write the register number that you want to write to into the Debug Core Register Selector Register.

10.4 Using application registers in core debug

You can also use the application registers for status access and to effect change on the system.

If you intend to use the application registers for core debug be aware that:

- There are read-modify-write issues if both AHB-AP and the application are modifying these registers.
- For the write registers like PENDSET and PENDCLR, there are read-modify-write issues because these are not read first.
- For registers containing priority and other read-write registers, the register can change between the read and the write when performing a read-modify-write operation. In some cases the registers enable byte access to alleviate this situation, and the debugger must be aware of these issues when the processor is running.

Table 10-5 shows the application registers and the register bits that are most useful for use in core debug. For a complete list of the application registers see the *ARMv7-M Architecture Reference Manual*.

Table 10-5 Application registers for use in core debug

Register	Bits or fields for use in core debug
Interrupt Control State	**ISRPREEMPT ISRPENDING VECTPENDING**.
Vector Table Offset	To find vector table
Application Interrupt/Reset Control	**VECTCLRACTIVE ENDIANESS**
Configuration Control	**DIV_0_TRP UNALIGN_TRP**.
System Handler Control and State	**ACTIVE PENDED**

Chapter 11
System Debug

This chapter describes the processor system debug. It contains the following sections:

- *About system debug* on page 11-2
- *System Debug Access* on page 11-3
- *System debug programmer's model* on page 11-5
- *Flash Patch and Breakpoint* on page 11-6
- *Data Watchpoint and Trace* on page 11-12
- *Instrumentation Trace Macrocell* on page 11-26
- *AHB Access Port* on page 11-35.

11.1 About system debug

The processor contains several system debug components that facilitate:

- low-cost debug
- trace and profiling
- breakpoints
- watchpoints
- code patching.

The system debug components are:

- *Flash Patch and Breakpoint* (FPB) unit to implement breakpoints and code patches.
- *Data Watchpoint and Trigger* (DWT) unit to implement watchpoints, trigger resources, and system profiling.
- *Instrumentation Trace Macrocell* (ITM) for application-driven trace source that supports *printf* style debugging.
- *Embedded Trace Macrocell* (ETM) for instruction trace. The processor is supported in versions with and without the ETM.

All the debug components exist on the internal *Private Peripheral Bus* (PPB) and can be accessed using privileged code.

Note

- For a description of the Core debug, see Chapter 10 *Core Debug*.

11.2 System Debug Access

Debug control and data access occurs through the AHB-AP interface. This interface is driven by either the SW-DP or JTAG-DP components. See Chapter 12 *Debug Port* for information on the SW-DP and JTAG-DP components. Access includes:

- The internal *Private Peripheral Bus* (PPB). Through this bus, the debugger can access Cortex-M3 components, including:
  - *Nested Vectored Interrupt Controller* (NVIC). Debug access to the processor core is made through the NVIC. For details, see Chapter 10 *Core Debug*.
  - *Data Watchpoint and Trace* (DWT) unit.
  - *Flash Patch and Breakpoint* (FPB) unit.
  - *Instrumentation Trace Macrocell* (ITM).
  - *Memory Protection Unit* (MPU).
- The External Private Peripheral Bus. Through this bus, debug access can be made to:
  - *Embedded Trace Macrocell* (ETM). A low-cost trace macrocell that supports instruction trace only. See Chapter 15 *Embedded Trace Macrocell* for more information.
  - *Trace Port Interface Unit* (TPIU). This component acts as a bridge between the Cortex-M3 trace data (from the ITM, and ETM if present) and an off-chip Trace Port Analyzer. See Chapter 13 *Trace Port Interface Unit* for more information.
  - ROM table.
- The DCode bus. Through this bus, debug can access memory located in code space.
- The System bus. Through this bus, memory and peripherals located in system bus space can be accessed.

Figure 11-1 on page 11-4 shows the structure of the system debug access, and shows how each of the system components and external buses can be accessed using the AHB-AP.

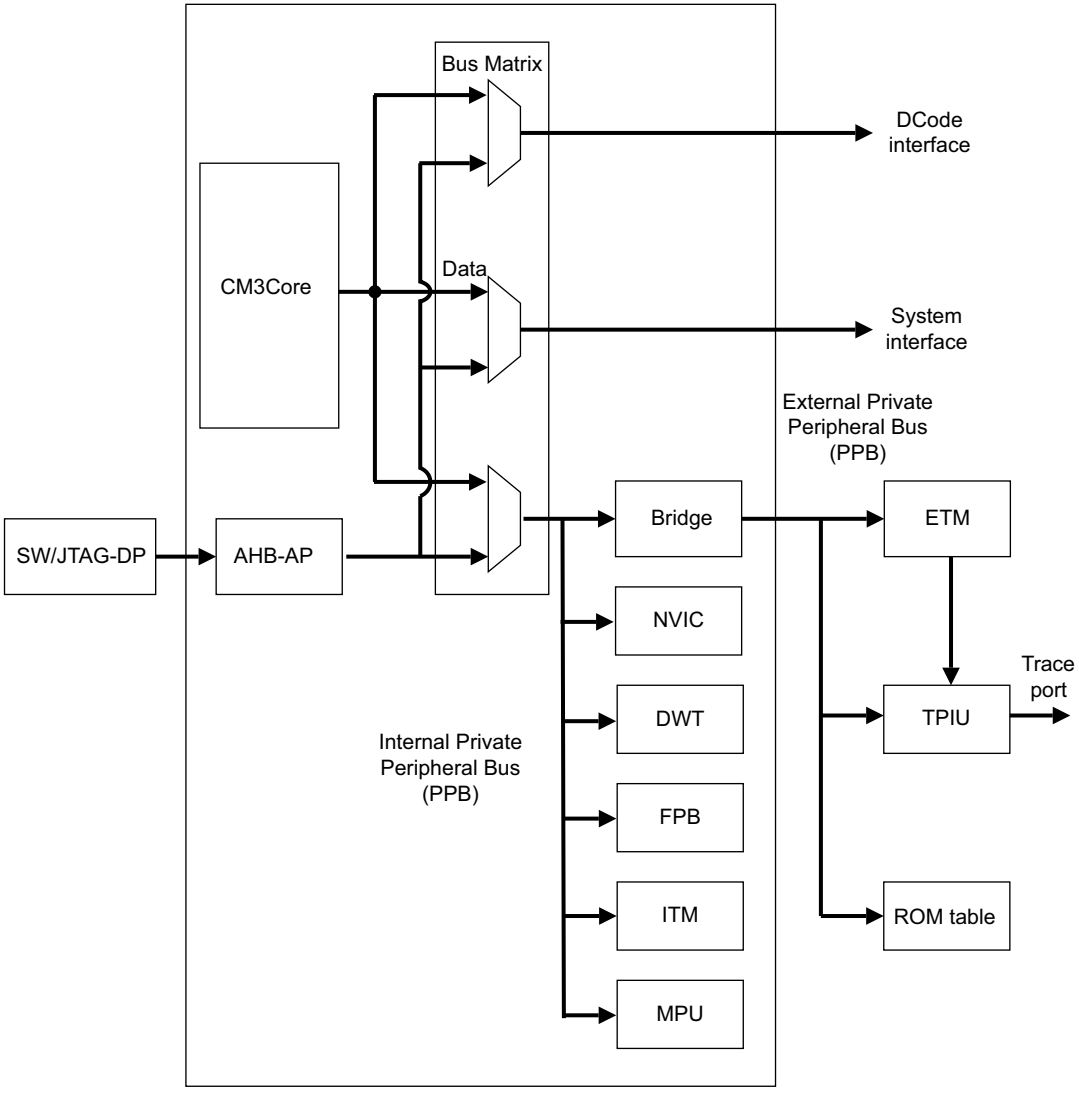


Figure 11-1 System debug access block diagram

11.3 System debug programmer's model

This section lists and describes the debug registers for all the system debug components. It contains:

- *Flash Patch and Breakpoint* on page 11-6
- *Data Watchpoint and Trace* on page 11-12
- *Instrumentation Trace Macrocell* on page 11-26
- *AHB Access Port* on page 11-35.

Note

- For a description of the Core debug registers, see *Core debug registers* on page 10-3.
- For a description of the JTAG-DP and SW-DP registers see Chapter 12 *Debug Port*.
- For a description of the TPIU, see Chapter 13 *Trace Port Interface Unit*.

11.4 Flash Patch and Breakpoint

The *Flash Patch and Breakpoint* unit (FPB):

- implements hardware breakpoints
- patches code and data from code space to system space.

The FPB unit contains:

- Two literal comparators, for matching against literal loads from Code space and remapping to a corresponding area in System space.
- Six instruction comparators, for matching against instruction fetches from Code space and remapping to a corresponding area in System space. Alternatively, the comparators can be individually configured to return a breakpoint instruction (BKPT) to the processor core on a match, so providing hardware breakpoint capability.

The FPB contains a global enable, but also individual enables for the eight comparators. If the comparison for an entry matches, the address is remapped to the address set in the remap register plus an offset corresponding to the comparator which matched, or is remapped to a BKPT instruction if that feature is enabled. The comparison happens on the fly, but the result of the comparison is too late to stop the original instruction fetch or literal load taking place from the Code space. The processor ignores this transaction however, and only the remapped transaction is used.

If an MPU is present, the MPU lookups are performed for the original address, not the remapped address.

Note

Unaligned literal accesses are not remapped. The original access to the DCode bus takes place in this case.

Note

Load exclusives are UNPREDICTABLE to the FPB. The address is remapped but the access does not take place as an exclusive load.

Note

Remapping to the bit-band alias directly accesses the alias address, and does not remap to the bit-band region.

11.4.1 FPB Programmer's Model

Table 11-1 lists the flash patch registers.

Table 11-1 Flash patch register summary

Name	Type	Address	Description
FP_CTRL	Read/write	0xE0002000	See *Flash Patch Control Register* on page 11-8
FP_REMAP	Read/write	0xE0002004	See *Flash Patch Remap Register* on page 11-9
FP_COMP0	Read/write	0xE0002008	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP1	Read/write	0xE000200C	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP2	Read/write	0xE0002010	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP3	Read/write	0xE0002014	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP4	Read/write	0xE0002018	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP5	Read/write	0xE000201C	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP6	Read/write	0xE0002020	See *Flash Patch Comparator Registers* on page 11-10
FP_COMP7	Read/write	0xE0002024	See *Flash Patch Comparator Registers* on page 11-10
PERIPID4	Read-only	0xE0002FD0	Value 0x04
PERIPID5	Read-only	0xE0002FD4	Value 0x00
PERIPID6	Read-only	0xE0002FD8	Value 0x00
PERIPID7	Read-only	0xE0002FDC	Value 0x00
PERIPID0	Read-only	0xE0002FE0	Value 0x03
PERIPID1	Read-only	0xE0002FE4	Value 0xB0
PERIPID2	Read-only	0xE0002FE8	Value 0x0B
PERIPID3	Read-only	0xE0002FEC	Value 0x00
PCELLID0	Read-only	0xE0002FF0	Value 0x0D
PCELLID1	Read-only	0xE0002FF4	Value 0xE0
PCELLID2	Read-only	0xE0002FF8	Value 0x05
PCELLID3	Read-only	0xE0002FFC	Value 0xB1

Flash Patch Control Register

Use the Flash Patch Control Register to enable the flash patch block.

The register address, access type, and Reset state are:

Address `0xE0002000`

Access Read/write

Reset state Bit[0] (ENABLE) is reset to 1'b0.

Figure 11-2 shows the fields of the Flash Patch Control Register.

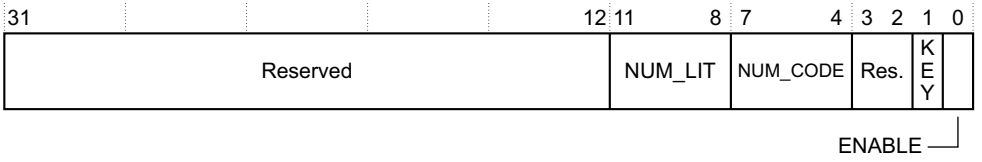


Figure 11-2 Flash Patch Control Register bit assignments

Table 11-2 describes the fields of the Flash Patch Control Register.

Table 11-2 Flash Patch Control Register bit assignments

Field	Name	Definition
[31:12]	-	Reserved. Read As Zero. Write Ignored.
[11:8]	NUM_LIT	Number of literal slots field. This read only field contains b0010 to indicate that there are two literal slots.
[7:4]	NUM_CODE	Number of code slots field. This read only field contains b0110 to indicate that there are six code slots.
[3:2]	-	Reserved.
[1]	KEY	Key field. To write to the Flash Patch Control Register, you must write a 1 to this write-only bit.
[0]	ENABLE	Flash patch unit enable bit: 1 = flash patch unit enabled 0 = flash patch unit disabled. Reset clears the ENABLE bit.

Flash Patch Remap Register

Use the Flash Patch Remap Register to provide the location in System space where a matched address is remapped. The REMAP address is 8-word aligned, with one word allocated to each of the eight FPB comparators.

A comparison match remaps to:

```
{3'b001, REMAP, COMP[2:0], HADDR[1:0]}
```

where:

- 3'b001 hardwires the remapped access to system space
- REMAP is the 24-bit, 8-word aligned remap address
- COMP is the matching comparator. See Table 11-3.

Table 11-3 COMP mapping

COMP[2:0]	Comparator	Description
000	FP_COMP0	Instruction comparator
001	FP_COMP1	Instruction comparator
010	FP_COMP2	Instruction comparator
011	FP_COMP3	Instruction comparator
100	FP_COMP4	Instruction comparator
101	FP_COMP5	Instruction comparator
110	FP_COMP6	Literal comparator
111	FP_COMP7	Literal comparator

- HADDR[1:0] is the two LSBs of the original address. HADDR[1:0] is always 2'b00 for instruction fetches.

The register address, access type, and Reset state are:

Address `0xE0002004`
Access Read/write
Reset state This register is not reset

Figure 11-3 shows the fields of the Flash Patch Remap Register.

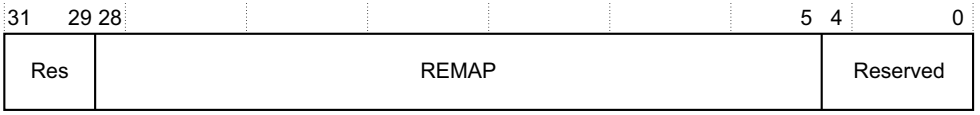


Figure 11-3 Flash Patch Remap Register bit assignments

Table 11-4 describes the fields of the Flash Patch Remap Register.

Table 11-4 Flash Patch Remap Register bit assignments

Field	Name	Definition
[31:29]	-	Reserved. Read as b001. Hardwires the remap to the system space.
[28:5]	REMAP	Remap base address field.
[4:0]	-	Reserved. Read As Zero. Write Ignored.

Flash Patch Comparator Registers

Use the Flash Patch Comparator Registers to store the values to compare with the PC address.

The register address, access type, and Reset state are:

Access Read/write

Address `0xE0002008`, `0xE000200C`, `0xE0002010`, `0xE0002014`, `0xE0002018`, `0xE000201C`, `0xE0002020`, `0xE0002024`

Reset state Bit[0] (ENABLE) is reset to 1'b0.

Figure 11-4 shows the fields of the Flash Patch Comparator Registers.

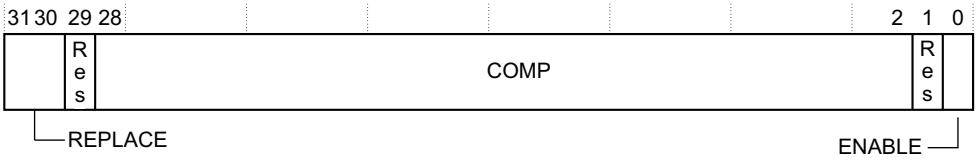


Figure 11-4 Flash Patch Comparator Registers bit assignments

Table 11-5 describes the fields of the Flash Patch Comparator Registers.

Table 11-5 Flash Patch Comparator Registers bit assignments

Field	Name	Definition
[31:30]	REPLACE	This is used to select what happens when the COMP address is matched. It is interpreted as: b00 = remap to remap address. See FP_REMAP. b01 = set BKPT on lower halfword, upper is unaffected. b10 = set BKPT on upper halfword, lower is unaffected. b11 = set BKPT on both lower and upper halfwords. Settings other than b00 are only valid for instruction comparators. Literal comparators ignore non-b00 settings. Address remapping only takes place for the b00 setting.
[29]	-	Reserved
[28:2]	COMP	Comparison address.
[1]	-	Reserved. Read As Zero. Write Ignore.
[0]	ENABLE	Compare and remap enable for Flash Patch Comparator Register *n*: 1 = Flash Patch Comparator Register *n* compare and remap enabled 0 = Flash Patch Comparator Register *n* compare and remap disabled The ENABLE bit of FP_CTRL must also be set to enable comparisons. Reset clears the ENABLE bit.

11.5 Data Watchpoint and Trace

The *Data Watchpoint and Trace* (DWT) unit performs the following debug functionality:

- It contains four comparators each of which can be configured as a hardware watchpoint, an ETM trigger, a PC sampler event trigger, or a data address sampler event trigger. The first comparator, DWT_COMP0, can also compare against the clock cycle counter (CYCCNT).
- The DWT contains several counters which count:
  - clock cycles (CYCCNT)
  - folded instructions
  - LSU operations
  - sleep cycles
  - CPI (all instruction cycles except for the first cycle)
  - interrupt overhead

  Note

  An event is emitted each time a counter overflows.
- It can be configured to emit PC samples at defined intervals, and to emit interrupt event information.

11.5.1 Summary and description of the DWT registers

Table 11-6 lists the DWT registers.

Table 11-6 DWT register summary

Name	Type	Address	Reset value	Description
DWT_CTRL	Read/write	0xE0001000	0x00000000	See *DWT Control Register* on page 11-14
DWT_CYCCNT	Read/write	0xE0001004	0x00000000	See *DWT Current PC Sampler Cycle Count Register* on page 11-17
DWT_CPICNT	Read/write	0xE0001008	-	See *DWT CPI Count Register* on page 11-18
DWT_EXCCNT	Read/write	0xE000100C	-	See *DWT Exception Overhead Count Register* on page 11-19
DWT_SLEEPCNT	Read/write	0xE0001010	-	See *DWT Sleep Count Register* on page 11-20

Table 11-6 DWT register summary (continued)

Name	Type	Address	Reset value	Description
DWT_LSUCNT	Read/write	0xE0001014	-	See *DWT LSU Count Register* on page 11-20
DWT_FOLDCNT	Read/write	0xE0001018	-	See *DWT Fold Count Register* on page 11-21
DWT_COMP0	Read/write	0xE0001020	-	See *DWT Comparator Register* on page 11-22
DWT_MASK0	Read/write	0xE0001024	-	See *DWT Mask Registers 0-3* on page 11-22
DWT_FUNCTION0	Read/write	0xE0001028	0x00000000	See *DWT Function Registers 0-3* on page 11-23
DWT_COMP1	Read/write	0xE0001030	-	See *DWT Comparator Register* on page 11-22
DWT_MASK1	Read/write	0xE0001034	-	See *DWT Mask Registers 0-3* on page 11-22
DWT_FUNCTION1	Read/write	0xE0001038	0x00000000	See *DWT Function Registers 0-3* on page 11-23
DWT_COMP2	Read/write	0xE0001040	-	See *DWT Comparator Register* on page 11-22
DWT_MASK2	Read/write	0xE0001044	-	See *DWT Mask Registers 0-3* on page 11-22
DWT_FUNCTION2	Read/write	0xE0001048	0x00000000	See *DWT Function Registers 0-3* on page 11-23
DWT_COMP3	Read/write	0xE0001050	-	See *DWT Comparator Register* on page 11-22
DWT_MASK3	Read/write	0xE0001054	-	See *DWT Mask Registers 0-3* on page 11-22
DWT_FUNCTION3	Read/write	0xE0001058	0x00000000	See *DWT Function Registers 0-3* on page 11-23
PERIPHID4	Read-only	0xE0001FD0	0x04	Value 0x04
PERIPHID5	Read-only	0xE0001FD4	0x00	Value 0x00
PERIPHID6	Read-only	0xE0001FD8	0x00	Value 0x00
PERIPHID7	Read-only	0xE0001FDC	0x00	Value 0x00
PERIPHID0	Read-only	0xE0001FE0	0x02	Value 0x02
PERIPHID1	Read-only	0xE0001FE4	0xB0	Value 0xB0
PERIPHID2	Read-only	0xE0001FE8	0x0B0	Value 0x0B
PERIPHID3	Read-only	0xE0001FEC	0x00	Value 0x00
PCELLID0	Read-only	0xE0001FF0	0x0D	Value 0x0D

Table 11-6 DWT register summary (continued)

Name	Type	Address	Reset value	Description
PCELLID1	Read-only	0xE0001FF4	0xE0	Value 0xE0
PCELLID2	Read-only	0xE0001FF8	0x05	Value 0x05
PCELLID3	Read-only	0xE0001FFC	0xB1	Value 0xB1

DWT Control Register

Use the DWT Control Register to enable the DWT block.

The register address, access type, and Reset state are:

Address 0xE0001000
Access Read/write
Reset state 0x40000000

Figure 11-5 shows the fields of the DWT Control Register.

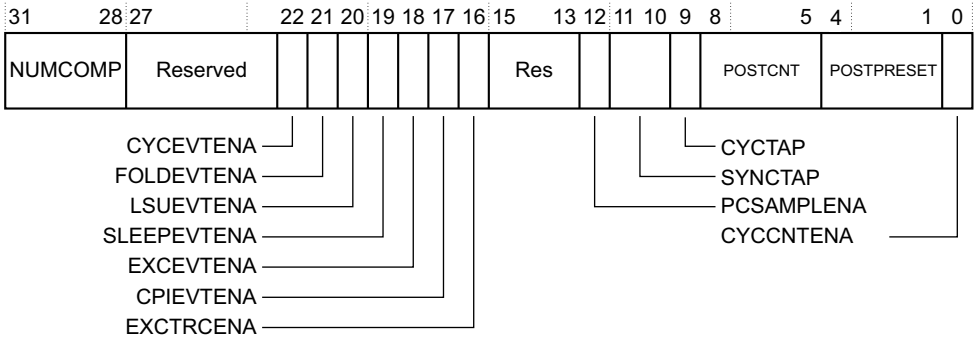


Figure 11-5 DWT Control Register bit assignments

Table 11-7 describes the fields of the DWT Control Register.

Table 11-7 DWT Control Register bit assignments

Field	Name	Definition
[31:28]	NUMCOMP	Number of comparators field. This read-only field contains b0100 to indicate four comparators.
[27:23]	-	Reserved. Read As Zero. Write ignore.
[22]	CYCEVTEN	Enables Cycle count event. Emits an event when the POSTCNT counter triggers it. See CYCTAP (bit [9]) and POSTPRESET (bits [4:1]) for details. 1 = Cycle count events enabled 0 = Cycle count events disabled. This event is only be emitted if PCSAMPLENA (bit [12]) is disabled. PCSAMPLENA overrides the setting of this bit. Reset clears the CYCEVTENA bit.
[21]	FOLDEVTENA	Enables Folded instruction count event. Emits an event when DWT_FOLDCNT overflows (every 256 cycles of folded instructions). A folded instruction is one which does not incur even one cycle to execute (e.g. an IT instruction is folded away and so does not use up one cycle). 1 = Folded instruction count events enabled 0 = Folded instruction count events disabled Reset clears the FOLDEVTENA bit.
[20]	LSUEVTENA	Enables LSU count event. Emits an event when DWT_LSUCNT overflows (every 256 cycles of LSU operation). LSU counts include all LSU costs after the initial cycle for the instruction. 1 = LSU count events enabled 0 = LSU count events disabled Reset clears the LSUEVTENA bit.
[19]	SLEEPEVTENA	Enables Sleep count event. Emits an event when DWT_SLEEPCNT overflows (every 256 cycles that the processor is sleeping). 1 = Sleep count events enabled 0 = Sleep count events disabled Reset clears the SLEEPEVTENA bit.
[18]	EXCEVTENA	Enables Interrupt overhead event. Emits an event when DWT_EXCCNT overflows (every 256 cycles of interrupt overhead). 1 = Interrupt overhead event enabled 0 = Interrupt overhead event disabled Reset clears the EXCEVTENA bit.

Table 11-7 DWT Control Register bit assignments (continued)

Field	Name	Definition
[17]	CPIEVTENA	Enables CPI count event. Emits an event when DWT_CPICNT overflows (every 256 cycles of multi-cycle instructions). 1 = CPI counter events enabled 0 = CPI counter events disabled. Reset clears the CPIEVTENA bit.
[16]	EXCTRCENA	Enables Interrupt event tracing: 1 = interrupt event trace enabled 0 = interrupt event trace disabled. Reset clears the EXCEVTENA bit.
[15:13]	-	Reserved
[12]	PCSAMPLEENA	Enables PC Sampling event. A PC sample event is emitted when the POSTCNT counter triggers it. See CYCTAP (bit [9]) and POSTPRESET (bits [4:1]) for details. Enabling this bit overrides CYCEVTENA (bit [20]). 1 = PC Sampling event enabled 0 = PC Sampling event disabled Reset clears the PCSAMPLENA bit.
[11:10]	SYNCTAP	This is used to feed a synchronization pulse to the ITM SYNCEN control. The value selected here picks the rate (should be about 1/second or less) by selecting a “tap” on the DWT_CYCCNT register. To use synchronization (heartbeat and hot-connect synchronization), CYCCNTENA must be set to 1, SYNCTAP must be set to one of its values, and SYNCEN must be set to 1. 0b00 Disabled. No synch counting 0b01 Tap at CYCCNT bit 24 0b10 Tap at CYCCNT bit 26 0b11 Tap at CYCCNT bit 28
[9]	CYCTAP	Selects a tap on the DWT_CYCCNT register. These are spaced at bits [6] and [10]. CYCTAP = 0 selects bit [6] to tap, and CYCTAP = 1 selects bit [10] to tap. When the selected bit in the CYCCNT register changes from 0 to 1 or 1 to 0, it emits into the POSTCNT (bits [8:5]) post-scalar counter. That counter will count-down. On a bit change when post-scalar is 0, it triggers an event for PC sampling or CYCEVTCNT.

Table 11-7 DWT Control Register bit assignments (continued)

Field	Name	Definition
[8:5]	POSTCNT	Post-scalar counter for CYCTAP. When the selected tapped bit changes from 0 to 1 or 1 to 0, the post scalar counter is down-counted when not 0. If 0, it triggers an event for PCSAMPLENA or CYCEVTENA use. It also reloads with the value from POSTPRESET (bits [4:1]).
[4:1]	POSTPRESET	Reload value for POSTCNT (bits [8:5]) post-scalar counter. If this value is 0, events are triggered on each tap change (a power of 2, such as 1<<6 or 1<<10). If this field has a non-0 value, this forms a count-down value (to be reloaded into POSTCNT each time it reaches 0). For example, a value 1 in this register means an event is formed every other tap change.
[0]	CYCCNTENA	Enable the DWT_CYCCNT counter. If not enabled, the counter does not count and so no event is generated for PS sampling or CYCCNTENA. The CYCCNT counter should be initialized to 0 by the debugger in normal use.

Note

The TRCENA bit of the Debug Exception and Monitor Control register must be set before the DWT can be used. See *Debug Exception and Monitor Control Register* on page 10-8.

Note

The DWT is enabled independently from the ITM. If the DWT is enabled to emit events then the ITM must also be enabled.

DWT Current PC Sampler Cycle Count Register

Use the DWT Current PC Sampler Cycle Count Register to count the number of core cycles. This count can be used to measure elapsed execution time.

The register address, access type, and Reset state are:

Address `0xE0001004`
Access Read-only
Reset state `0x00000000`

Table 11-8 describes the fields of the DWT Current PC Sampler Cycle Count Register.

Table 11-8 DWT Current PC Sampler Cycle Count Register bit assignments

Field	Name	Definition
[31:0]	CYCCNT	Current PC Sampler Cycle Counter count value. When enabled, this counter counts the number of core cycles, except when the core is halted. DWT_CYCCNT is a free running counter, counting upwards. It wraps around to 0 on overflow. The debugger should initialize this to 0 when first enabling.

This is a free-running counter. The counter has three functions:

- When PCSAMPLENA is set, the PC is sampled and emitted when the selected tapped bit changes value (0 to 1 or 1 to 0) and any post-scalar value counts to 0.
- When CYCEVTENA is set (and PCSAMPLENA is clear), an event is emitted when the selected tapped bit changes value (0 to 1 or 1 to 0) and any post-scalar value counts to 0.
- Can be used by applications and debuggers to measure elapsed execution time. By subtracting a start and an end time, an application can measure time between in-core clocks (other than when Halted in debug). This is valid to 2^{32} core clock cycles (for example, almost 82 seconds at 50MHz).

DWT CPI Count Register

Use the DWT CPI Count Register to count the total number of instruction cycles beyond the first cycle.

The register address, access type, and Reset state are:

Address `0xE0001008`

Access Read-write

Reset state -

Figure 11-6 shows the fields of the DWT CPI Count Register.

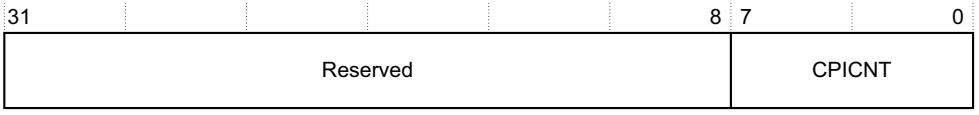


Figure 11-6 DWT CPI Count Register bit assignments

Table 11-9 describes the fields of the DWT CPI Count Register.

Table 11-9 DWT CPI Count Register bit assignments

Field	Name	Definition
[31:8]	-	Reserved.
[7:0]	CPICNT	Current CPI counter value. Increments on the additional cycles (the first cycle is not counted) required to execute all instructions except those recorded by DWT_LSUCNT. This counter also increments on all instruction fetch stalls. If CPIEVTENA is set, an event is emitted when the counter overflows. Clears to 0 on enabling.

DWT Exception Overhead Count Register

Use the DWT Exception Overhead Count Register to count the total cycles spent in interrupt processing.

The register address, access type, and Reset state are:

Address `0xE000100C`

Access Read-write

Reset state -

Figure 11-7 shows the fields of the DTW Exception Overhead Count Register.

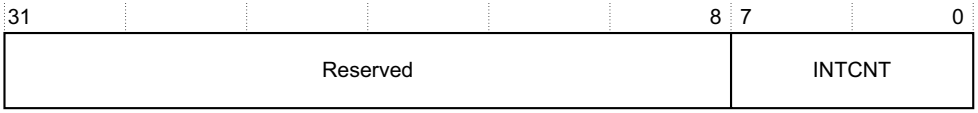


Figure 11-7 DWT Exception Overhead Count Register bit assignments

Table 11-10 describes the fields of the DWT Exception Overhead Count Register.

Table 11-10 DWT Exception Overhead Count Register bit assignments

Field	Name	Definition
[31:8]	-	Reserved.
[7:0]	EXCCNT	Current interrupt overhead counter value. Counts the total cycles spent in interrupt processing (for example entry stacking, return unstacking, pre-emption). An event is emitted on counter overflow (every 256 cycles). This counter initializes to 0 when enabled. Clears to 0 on enabling.

DWT Sleep Count Register

Use the DWT Sleep Count Register to count the total number of cycles during which the processor is sleeping.

The register address, access type, and Reset state are:

Address `0xE0001010`

Access Read-write

Reset state -

Figure 11-8 shows the fields of the DTW Sleep Count Register.

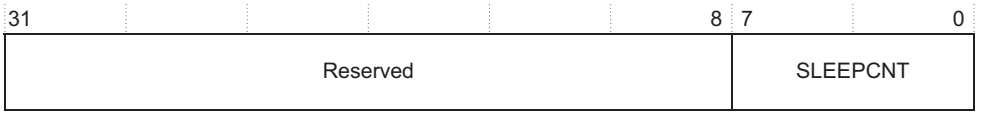


Figure 11-8 DWT Sleep Count Register bit assignments

Table 11-11 describes the fields of the DWT Sleep Count Register.

Table 11-11 DWT Sleep Count Register bit assignments

Field	Name	Definition
[31:8]	-	Reserved.
[7:0]	SLEEPCNT	Sleep counter. Counts the number of cycles during which the processor is sleeping. An event is emitted on counter overflow (every 256 cycles). This counter initializes to 0 when enabled.

Note

SLEEPCNT is clocked using **FCLK**. It is possible that the frequency of **FCLK** may be reduced while the processor is sleeping to minimize power consumption. This means that sleep duration must be calculated with the frequency of **FCLK** during sleep.

DWT LSU Count Register

Use the DWT LSU Count Register to count the total number of cycles during which the processor is processing an LSU operation beyond the first cycle.

The register address, access type, and Reset state are:

Address `0xE0001014`

Access Read/write

Reset state -

Figure 11-9 describes the fields of the DWT LSU Count Register.

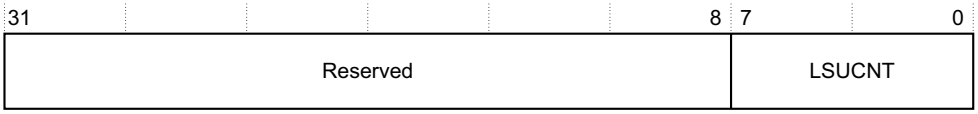


Figure 11-9 DWT LSU Count Register bit assignments

Table 11-12 describes the fields of the DWT LSU Count Register.

Table 11-12 DWT LSU Count Register bit assignments

Field	Name	Definition
[31:8]	-	Reserved.
[7:0]	LSUCNT	LSU counter. This counts the total number of cycles that the processor is processing an LSU operation. The initial execution cost of the instruction is not counted. For example, an LDR which takes two cycles to complete increments this counter one cycle. Equivalently, an LDR which stalls for two cycles (and so takes four cycles), increments this counter three times. An event is emitted on counter overflow (every 256 cycles). Clears to 0 on enabling.

DWT Fold Count Register

Use the DWT Fold Count Register to count the total number of folded instructions. This counts 1 for each instruction which takes 0 cycles.

The register address, access type, and Reset state are:

Address `0xE0001018`

Access Read/write

Reset state -

Figure 11-10 describes the fields of the DWT Fold Count Register.

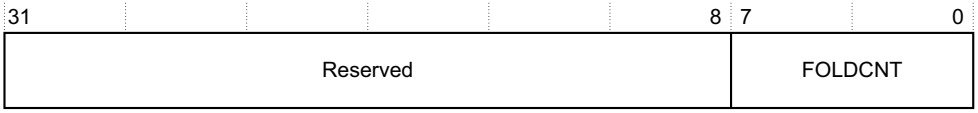


Figure 11-10 DWT Fold Count Register bit assignments

Table 11-13 describes the fields of the DWT Fold Count Register.

Table 11-13 DWT Fold Count Register bit assignments

Field	Name	Definition
[31:8]	-	Reserved.
[7:0]	FOLDCNT	This counts the total number folded instructions. This counter initializes to 0 when enabled.

DWT Comparator Register

Use the DWT Comparator Registers 0-3 to write the values that trigger watchpoint events.

The register address, access type, and Reset state are:

Address `0xE0001020`, `0xE0001030`, `0xE0001040`, `0xE0001050`

Access Read/write

Reset state -

Table 11-14 describes the field of DWT Comparator Registers 0-3.

Table 11-14 DWT Comparator Registers 0-3 bit assignments

Field	Name	Definition
[31:0]	COMP	Data value to compare against. If PCMATCH is set, the PC is compared with this value, otherwise the data address is compared. DWT_COMP0 can also compare against the value of the PC Sampler Counter (DWT_CYCCNT).

DWT Mask Registers 0-3

Use the DWT Mask Registers 0-3 to apply a mask to data addresses when matching against COMP.

The register address, access type, and Reset state are:

Address `0xE0001024`, `0xE0001034`, `0xE0001044`, `0xE0001054`

Access Read/write

Reset state -

Figure 11-11 on page 11-23 shows the fields of DWT Mask Registers 0-3.

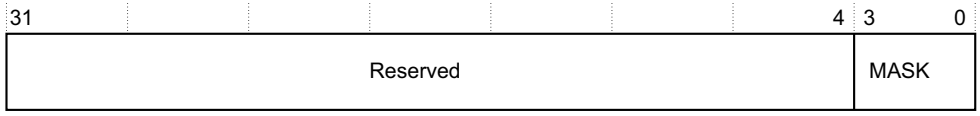


Figure 11-11 DWT Mask Registers 0-3 bit assignments

Table 11-15 describes the field of DWT Mask Registers 0-3.

Table 11-15 DWT Mask Registers 0-3 bit assignments

Field	Name	Definition
[31:4]	-	Reserved.
[3:0]	MASK	Mask on data address when matching against COMP. This is the size of the ignore mask. So, ~0<<MASK forms the mask against the address to use. That is, DWT matching is performed as: (ADDR & (~0 << MASK)) == COMP However, the actual comparison is slightly more complex to enable matching an address wherever it appears on a bus. So, if COMP is 3, this matches a word access of 0, because 3 would be within the word.

DWT Function Registers 0-3

Use the DWT Function Registers 0-3 to control the operation of the comparator. Each comparator can:

- Match against either the PC or the data address. This is controlled by CYCMATCH. This function is only available for comparator 0 (DWT_COMP0).
- Cause data or PC couples to be emitted, trigger the ETM, or generate a watchpoint depending on the operation defined by FUNCTION.

The register address, access type, and Reset state are:

Address `0xE0001028, 0xE0001038, 0xE0001048, 0xE0001058`

Access Read/write

Address `0x00000000`

Figure 11-12 on page 11-24 shows the fields of DWT Function Registers 0-3.

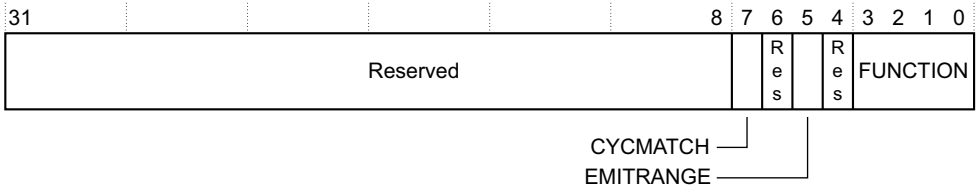


Figure 11-12 DWT Function Registers 0-3 bit assignments

Table 11-16 describes the fields of DWT Function Registers 0-3.

Table 11-16 Bit functions of DWT Function Registers 0-3

Field	Name	Definition
[31:8]	-	Reserved
[7]	CYCMATCH	Only available in comparator 0. When set, this comparator compares against the PC Sampler Counter.
[6]	-	Reserved
[5]	EMITRANGE	Emit range field. Reserved to permit emitting offset when range match occurs. Reset clears the EMITRANGE bit. PC sampling is not supported when EMITRANGE is enabled. EMITRANGE only applies for: FUNCTION=b0001, b0010, and b0011.
[4]	-	Reserved
[3:0]	FUNCTION	See Table 11-17 on page 11-24 for FUNCTION settings.

Table 11-17 Settings for DWT Function Registers

Value	Function
b0000	Disabled
b0001	EMITRANGE=0, sample and emit PC through ITM EMITRANGE=1, emit address offset through ITM
b0010	EMITRANGE=0, emit data through ITM on read and write. EMITRANGE=1, emit data and address offset through ITM on read or write.
b0011	EMITRANGE=0, sample PC and data value through ITM on read or write. EMITRANGE=1, emit address offset and data value through ITM on read or write.
b0100	Watchpoint on PC match.

Table 11-17 Settings for DWT Function Registers (continued)

Value	Function
b0101	Watchpoint on read.
b0110	Watchpoint on write.
b0111	Watchpoint on read or write.
b1000	ETM trigger on PC match
b1001	ETM trigger on read
b1010	ETM trigger on write
b1011	ETM trigger on read or write
b1100	Reserved
b1101	Reserved
b1110	Reserved
b1111	Reserved

Note

- If the ETM is not fitted, then ETM trigger is not possible.
- Data value is only sampled for accesses which do not fault (MPU or bus fault). The PC is sampled irrespective of any faults. The PC is only sampled for the first address of a burst.
- PC match is not recommended for watchpoints because it stops after the instruction - it is mainly used for guards and for ETM triggering.

11.6 Instrumentation Trace Macrocell

The *Instrumentation Trace Macrocell* (ITM) is a an application driven trace source that supports *printf* style debugging to trace OS and application events, and emits diagnostic system information. Trace information is emitted from the ITM as packets. Packets can be generated from three sources. If multiple sources generate packets at the same time, ITM arbitrates the order in which packets are output. The three sources in decreasing order of priority are:

- Software trace. Software can write directly to ITM stimulus registers. This causes packets to be emitted.
- Hardware trace. These packets are generated by the DWT, and emitted by the ITM.
- Time stamping. Timestamps are emitted relative to packets. ITM contains a 21-bit counter to generate the timestamp. The counter is clocked using either the Cortex-M3 clock, or the bitclock rate of the SWV output.

11.6.1 Summary and description of the ITM registers

Note

TRCENA of the Debug Exception and Monitor Control Register (see *Debug Exception and Monitor Control Register* on page 10-8) must be enabled before you program or use the ITM.

Table 11-18 lists the ITM registers.

Table 11-18 ITM register summary

Name	Type	Address	Reset value	Description
Stimulus Ports 0-31	Read/write	0xE0000000-0xE000007C	-	See *ITM Stimulus Ports 0-31* on page 11-28
Trace Enable	Read/write	0xE0000E00	0x00000000	See *ITM Trace Enable Register* on page 11-28
Trace Privilege	Read/write	0xE0000E40	0x00000000	See *ITM Trace Privilege Register* on page 11-29
Control Register	Read/write	0xE0000E80	0x00000000	See *ITM Control Register* on page 11-30

Table 11-18 ITM register summary (continued)

Name	Type	Address	Reset value	Description
Integration Write	Write-only	0xE0000EF8	0x00000000	See *ITM Integration Write Register* on page 11-31
Integration Read	Read-only	0xE0000EFC	0x00000000	See *ITM Integration Read Register* on page 11-32
Integration Mode Control	Read/write	0xE0000F00	0x00000000	See *ITM Integration Mode Control Register* on page 11-32
Lock Access Register	Write-only	0xE0000FB0	0x00000000	See *ITM Lock Access Register* on page 11-33
Lock Status Register	Read-only	0xE0000FB4	0x00000003	See *ITM Lock Status Register* on page 11-33
PERIPHID4	Read-only	0xE0000FD0	0x00000004	Value 0x04
PERIPHID5	Read-only	0xE0000FD4	0x00000000	Value 0x00
PERIPHID6	Read-only	0xE0000FD8	0x00000000	Value 0x00
PERIPHID7	Read-only	0xE0000FDC	0x00000000	Value 0x00
PERIPHID0	Read-only	0xE0000FE0	0x00000002	Value 0x01
PERIPHID1	Read-only	0xE0000FE4	0x000000B0	Value 0xB0
PERIPHID2	Read-only	0xE0000FE8	0x0000000B	Value 0x0B
PERIPHID3	Read-only	0xE0000FEC	0x00000000	Value 0x00
PCELLID0	Read-only	0xE0000FF0	0x0000000D	Value 0x0D
PCELLID1	Read-only	0xE0000FF4	0x000000E0	Value 0xE0
PCELLID2	Read-only	0xE0000FF8	0x00000005	Value 0x05
PCELLID3	Read-only	0xE0000FFC	0x000000B1	Value 0xB1

Note

ITM registers are fully accessible in Privileged mode. In User mode, all registers can be read, but only the Stimulus Registers and Trace Enable Registers can be written, and only when the corresponding Trace Privilege Register bit is set. Invalid User mode writes to the ITM registers is discarded.

ITM Stimulus Ports 0-31

Each of the 32 stimulus ports has its own address. A write to one of these locations causes data to be written into the FIFO if the corresponding bit in the Trace Enable Register is set. Reading from any of the stimulus ports returns the FIFO status (0 = Full, 1 = not Full) in bit 0.

The polled FIFO interface does not provide an atomic read-modify-write, so the Cortex-M3 exclusive monitor must be used if a polled *printf* is used concurrently with ITM usage by interrupts or other threads. The following polled code guarantees stimulus is not lost by polled access to the ITM:

```
; Cortex-M3 exclusive monitor cleared by interrupt

; R0 = FIFO-full/exclusive status

; R1 = base of ITM stimulus ports

; R2 = value to write

retry

        LDREX R0, [R1, #??]      ; read FIFO status and request excl lock

        CBZEQ R0, retry          ; FIFO not ready, try again

        STREX R0, R2, [R1, #??]  ; store if FIFO !Full and excl lock

        CZBNE R0, retry          ; excl lock failed, try again
```

ITM Trace Enable Register

Use the Trace Enable Register to generate trace data by writing to the corresponding stimulus port.

The register address, access type, and Reset state are:

Access Read/write

Address `0xE0000E00`

Reset `0x00000000`

Table 11-19 describes the fields of the ITM Trace Enable Register.

Table 11-19 Bit functions of the ITM Trace Enable Register

Field	Name	Definition
[31:0]	STIMULUS MASK	Bit mask to enable tracing on ITM stimulus ports. One bit per stimulus port.

Note

Privileged writes are accepted to this register if ITMEN is set. User writes are accepted to this register if ITMEN is set and the appropriate privilege mask is cleared. Privileged access to the stimulus ports enables an RTOS kernel to guarantee instrumentation slots or bandwidth as required.

ITM Trace Privilege Register

Use the ITM Trace Privilege Register to enable an operating system to control which stimulus ports are accessible by user code.

Note

You can only write to this register in Privileged mode.

The register address, access type, and Reset state are:

Access Read/write

Address `0xE0000E40`

Reset `0x00000000`

Figure 11-13 shows the ITM Trace Privilege Register bit assignments.

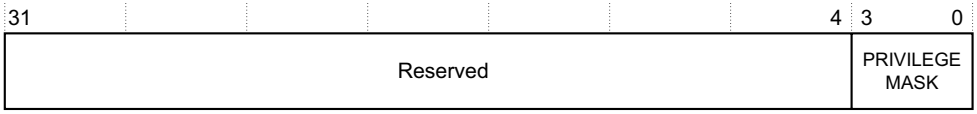


Figure 11-13 ITM Trace Privilege Register bit assignments

Table 11-20 describes the fields of the ITM Trace Privilege Register.

Table 11-20 Bit functions of the ITM Trace Privilege Register

Field	Name	Definition
[31:4]	-	Reserved
[3:0]	Privilege Mask	Bit mask to enable tracing on ITM stimulus ports: bit[0] = stimulus ports [7:0] bit[1] = stimulus ports [15:8] bit[2] = stimulus ports [23:16] bit[3] = stimulus ports [31:24].

ITM Control Register

Use this register to configure and control ITM transfers.

Note

You can only write to this register in privilege mode.

The register address, access type, and Reset state are:

Access Read/write

Address `0xE0000E80`

Reset `0x00000000`

Figure 11-14 shows the ITM Control Register bit assignments.

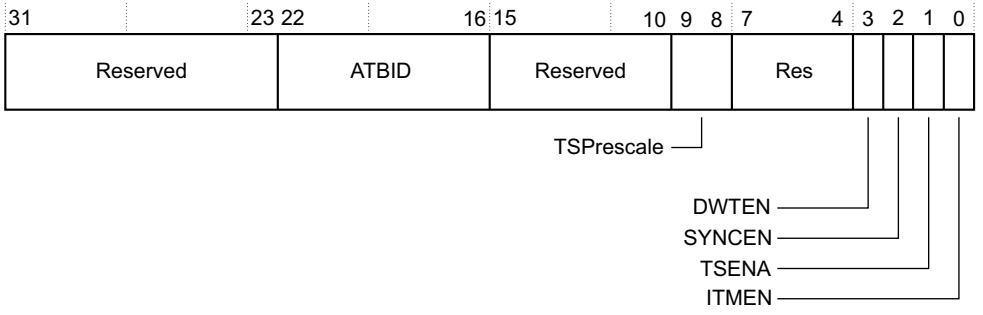


Figure 11-14 ITM Control Register bit assignments

Table 11-21 describes the fields of the ITM Control Register.

Table 11-21 Bit functions of the ITM Control Register

Field	Name	Definition
[31:23]	-	0b000000000
[22:16]	ATBID	ATB ID for CoreSight system
[15:10]	-	0b000000
[9:8]	TSPrescale	Timestamp prescaler 0b00 = no prescaling 0b01 = divide by 4 0b10 = divide by 16 0b11 = divide by 64

Table 11-21 Bit functions of the ITM Control Register

Field	Name	Definition
[7:4]	-	Reserved
[3]	DWTEN	Enable the DWT stimulus
[2]	SYNCEN	Enable sync packets for TPIU
[1]	TSENA	Enable differential timestamps. Differential timestamps are emitted when a packet is written to the FIFO with a non-zero timestamp counter, and when the timestamp counter overflows. Timestamps are emitted during idle times after a fixed number of cycles. This provides a time reference for packets and inter-packet gaps.
[0]	ITMEN	Enable ITM. This is the master enable, and must be set before ITM Stimulus and Trace Enable registers can be written.

Note

DWT is not enabled in the ITM block. However, DWT stimulus entry into the FIFO is controlled by DWTEN. If DWT requires timestamping, the TSSEN bit must be set.

ITM Integration Write Register

Use this register to determine the behavior of the ATVALIDM bit.

Figure 11-15 shows the ITM Integration Write Register bit assignments.

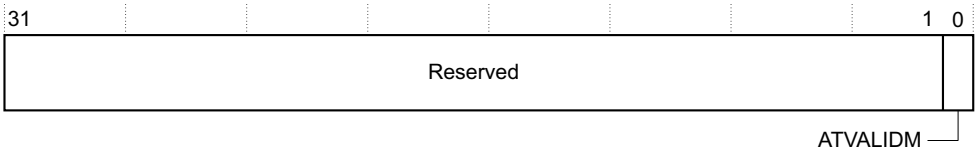


Figure 11-15 ITM Integration Write Register bit assignments

Table 11-22 describes the fields of the ITM Integration Write Register.

Table 11-22 Bit functions of the ITM Integration Write Register

Field	Name	Definition
[31:1]	-	Reserved
[0]	ATVALIDM	When the integration mode is set: 0 = ATVALIDM clear 1 = ATVALIDM set.

Note

ATVALIDM is driven by bit 0 when mode is set.

ITM Integration Read Register

Use this register to read the value on ATREADYM

Figure 11-16 shows the ITM Integration Read Register bit assignments.

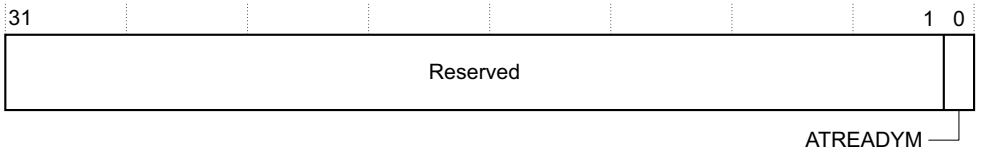


Figure 11-16 ITM Integration Read Register bit assignments

Table 11-23 describes the fields of the ITM Integration Read Register.

Table 11-23 Bit functions of the ITM Integration Read Register

Field	Name	Definition
[31:1]	-	Reserved
[0]	ATREADYM	Value on ATREADYM

ITM Integration Mode Control Register

Use this register to enable write accesses to the Control Register.

Figure 11-17 on page 11-33 shows the ITM Integration Mode Control Register bit assignments.

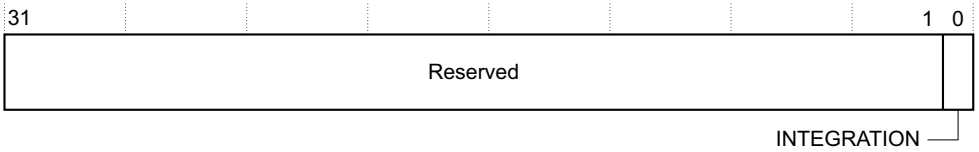


Figure 11-17 ITM Integration Mode Control bit assignments

Table 11-24 describes the fields of the ITM Integration Mode Control Register

Table 11-24 Bit functions of the ITM Integration Mode Control Register

Field	Name	Definition
[31:1]	-	Reserved
[0]	INTEGRATION	0 = ATVALIDM normal 1 = ATVALIDM driven from Integration Write Register.

ITM Lock Access Register

Use this register to prevent write accesses to the Control Register.

Table 11-25 describes the fields of the ITM Lock Access Register

Table 11-25 Bit functions of the ITM Lock Access Register

Field	Name	Definition
[31:0]	Lock Access	A privileged write of `0xC5ACCE55` enables further write access to Control Register `0xE00::0xFFC`. An invalid write removes write access.

ITM Lock Status Register

Use this register to enable write accesses to the Control Register.

Figure 11-18 on page 11-34 shows the ITM Lock Status Register bit assignments.

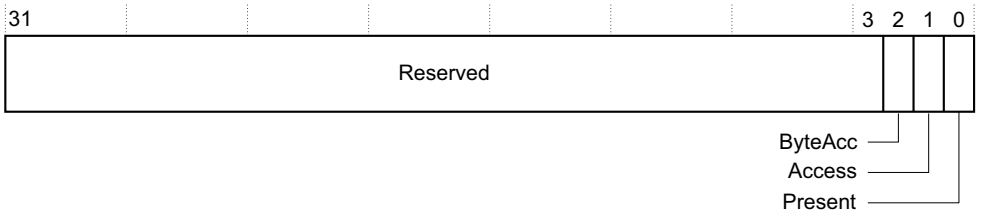


Figure 11-18 ITM Lock Status Register bit assignments

Table 11-26 describes the fields of the ITM Lock Status Register

Table 11-26 Bit functions of the ITM Lock Status Register

Field	Name	Definition
[31:3]	-	Reserved
[2]	ByteAcc	You cannot implement 8-bit lock accesses
[1]	Access	Write access to component is blocked. All writes are ignored, reads are permitted.
[0]	Present	Indicates that a lock mechanism exists for this component

11.7 AHB Access Port

The Advanced High-performance Bus Access Port (AHB-AP) is a debug access port into Cortex-M3, and provides access to all memory and registers in the system, including processor registers (through the NVIC). System access is independent of the processor status. AHB-AP is accessed from either a SW-DP or a JTAG-DP.

AHB-AP is a master into the Bus Matrix. Transactions are made using the AHB-AP Programmer's Model, see *Summary and description of the AHB-AP registers*, which generates AHBLite transactions into the Bus Matrix.

11.7.1 AHB-AP transaction types

AHB-AP does not do back-to-back transactions on the bus, and so all transactions are non-sequential. AHB-AP can perform unaligned and bit-band transactions. These are handled by the Bus Matrix. AHB-AP transactions are not subject to MPU lookups. AHB-AP transactions bypass the FPB, and so no remapping of AHB-AP transactions by FPB is possible.

AHB-AP supports a sideband signal, HABORT, which is driven by JTAG/SW-DP initiated transaction aborts. HABORT is driven into the BusMatrix, which resets the BusMatrix state, so that the Private Peripheral Bus can be accessed by AHB-AP for "last ditch" debugging (read/stop/reset the core).

AHB-AP transactions are little endian.

11.7.2 Summary and description of the AHB-AP registers

Table 11-27 lists the AHB-AP registers.

Table 11-27 AHB-AP register summary

Name	Type	Address	Reset value	Description
Control and Status Word (CSW)	R/W	`0x00`	See Register	See *AHB-AP Control and Status Word Register* on page 11-36
Transfer Address (TAR)	R/W	`0x04`	`0x00000000`	See *AHB-AP Transfer Address Register* on page 11-38
Data Read/write (DRW)	R/W	`0x0C`	-	See *AHB-AP Data Read/Write Register* on page 11-39
Banked Data 0 (BD0)	R/W	`0x10`	-	See *AHB-AP Banked Data Registers 0-3* on page 11-39

Table 11-27 AHB-AP register summary

Name	Type	Address	Reset value	Description
Banked Data 1 (BD1)	R/W	0x14	-	See *AHB-AP Banked Data Registers 0-3* on page 11-39
Banked Data 2 (BD2)	R/W	0x18	-	See *AHB-AP Banked Data Registers 0-3* on page 11-39
Banked Data 3 (BD3)	R/W	0x1C	-	See *AHB-AP Banked Data Registers 0-3* on page 11-39
Debug ROM Address	RO	0xF8	0xE000E000	See *AHB-AP Debug ROM Address Register* on page 11-39
Identification Register (IDR)	RO	0xFC	0x04770011	See *AHB-AP ID Register* on page 11-40

AHB-AP Control and Status Word Register

Use this register to configure and control transfers through the AHB interface.

Figure 11-19 shows the fields of the AHB-AP Control and Status Word Register.

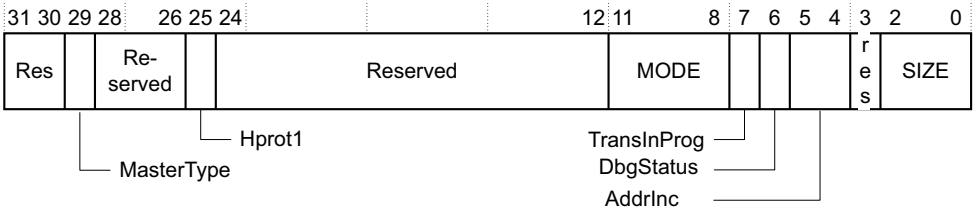


Figure 11-19 AHB-AP Control and Status Word Register

Table 11-28 describes the fields of the AHB-AP Control and Status Word Register.

Table 11-28 Bit functions of the AHB-AP Control and Status Word Register

Field	Name	Definition
[31:30]	-	Reserved. Read as b010
[29]	MasterType[a]	0 = Core 1 = Debug This bit cannot be cleared if COREACCEN = 0. Read back to confirm if accepted. It cannot be changed if transaction is outstanding. (Debugger should first check TransinProg.) Reset value = 0b1
[28:26]	-	Reserved, 0b000.
[25]	Hprot1	User/Privilege control - HPROT[1] Reset value = 0b1
[24]	-	Reserved, 0b1.
[23:12]	-	Reserved, `0x000`
[11:8]	Mode	Mode of operation bits: b0000 = normal download/upload mode b0001-b1111 are reserved. Reset value = 0b0000.
[7]	TransINProg	Transfer in progress. This field indicates if a transfer is in progress on the APB master port.
[6]	DbgStatus	Indicates the status of the DBGEN port. If DbgStatus is LOW, no AHB transfers carried out. 1 = AHB transfers permitted. 0 = AHB transfers not permitted.

Table 11-28 Bit functions of the AHB-AP Control and Status Word Register (continued)

Field	Name	Definition
[5:4]	AddrInc	Auto address increment and pack mode on Read or Write data access. Only increments if the current transaction completes with no error. Auto address incrementing and packed transfers are not performed on access to Banked Data registers `0x10 - 0x1C`. The status of these bits is ignored in these cases. Increments and wraps within a 4KB address boundary, for example for word incrementing from `0x1000 to 0x1FFC`. If the start is at `0x14A0`, then the counter increments to `0x1FFC`, wraps to `0x1000`, then continues incrementing to `0x149C`. 0b00 = auto increment off. 0b01 = increment single. Single transfer from corresponding byte lane. 0b10 = increment packed. 0b11 = reserved. No transfer. Size of address increment is defined by the Size field[2:0] Reset value: 0b00
[3]	-	Reserved.
[2:0]	SIZE	Size of access field: b000 = 8 bits b001 = 16 bits b010 = 32 bits b011-111 are reserved. Reset value: b000.

a. When clear, this bit will prevent the debugger from setting the C_DEBUGEN bit in the Debug Halting Control and Status Register, and so prevent the debugger from being able to halt the core.

AHB-AP Transfer Address Register

Use this register to program the address of the current transfer.

Table 11-29 describes the fields of the AHB-AP Transfer Address Register.

Table 11-29 AHB-AP Transfer Address Register bit functions

Field	Name	Definition
[31:0]	ADDRESS	Current transfer address. Reset value = `0x00000000`

AHB-AP Data Read/Write Register

Use this register to read and write data for the current transfer.

Table 11-30 describes the fields of the AHB-AP Data Read/Write Register.

Table 11-30 Bit functions of the AHB-AP Data Read/Write Register

Field	Name	Definition
[31:0]	DATA	Write mode: Data value to write for the current transfer. Read mode: Data value to read for the current transfer. Reset value = 0x00000000

AHB-AP Banked Data Registers 0-3

Use these register to directly map AHB-AP accesses to AHB transfers without rewriting the *AHB-AP Transfer Address Register* (TAR).

Table 11-31 describes the field of the AHB-AP Banked Data Registers.

Table 11-31 Bit functions of the AHB-AP Banked Data Register

Field	Name	Definition
[31:0]	DATA	BD0-BD3 provide a mechanism for directly mapping through DAP accesses to AHB transfers without having to rewrite the Transfer Address Register (TAR) within a four location boundary, so BD0 reads/write from TAR, BD1 from TAR+4 and so on. If **DAPADDR[7:4]** == 0x0001, so accessing AHB-AP registers in the range 0x10-0x1C, then the derived **HADDR[31:0]** is as follows: Write mode: Data value to write for the current transfer to external address TAR[31:4] + **DAPADDR[3:0]**. Read mode: Data value read from the current transfer from external address TAR[31:4] + **DAPADDR[3:0]**. Auto address incrementing is not performed on DAP accesses to BD0-BD3. Banked transfers are only supported for word transfers. Non-word banked transfer size is currently ignored, assumed word access. Reset value - `0x00000000`

AHB-AP Debug ROM Address Register

This register specifies the base address of the debug interface. It is read-only.

Table 11-32 describes the fields of the AHB-AP Debug ROM Address Register.

Table 11-32 Bit functions of the AHB-AP Debug ROM Address Register

Field	Name	Definition
[31:0]	Debug ROM address	Base address of debug interface.

AHB-AP ID Register

This register defines the external interface on the access port.

Figure 11-20 shows the fields of the AHB-AP ID Register.

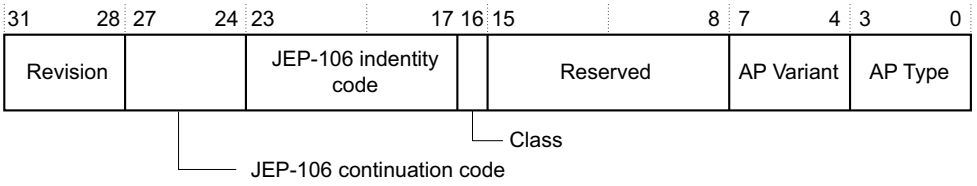


Figure 11-20 AHB-AP ID Register

Table 11-33 describes the fields of the AHB-AP ID Register.

Table 11-33 Bit functions of the AHB-AP ID Register

Field	Name	Definition
[31:28]	Revision	This field is zero for the first implementation of an AP design, and is updated for each major revision of the design.
[27:24]	JEP-106 continuation code	For an ARM-designed AP, this field has value 0b0100, `0x4`.
[23:17]	JEP-106 identity code	For an ARM-designed AP, this field has value 0b0111011, `0x3B`.
[16]	Class	0b1: This AP is a Memory Access Port
[15:8]	-	Reserved. SBZ.
[7:4]	AP Variant	`0x1`: Cortex-M3 variant
[3:0]	AP Type	`0x1`: AMBA AHB bus

Chapter 12
Debug Port

This chapter describes the processor debug port. It contains:

- *About the Debug Port* on page 12-2
- *JTAG-DP* on page 12-3
- *SW-DP* on page 12-20
- *Common Debug Port (DP) features* on page 12-41
- *Debug Port Programmer's Model* on page 12-47.

12.1 About the Debug Port

The processor contains an AHB-AP interface for debug accesses. This interface is accessed external to the processor by means of a *Debug Port* (DP) component. The Cortex-M3 system supports two possible DP implementations:

- The *JTAG Debug Port* (JTAG-DP). The JTAG-DP is based on the IEEE 1149.1 Test Access Port (TAP) and Boundary Scan Architecture, widely referred to as JTAG, and provides a JTAG interface to the AHB-AP port. For more information, see *JTAG-DP* on page 12-3,
- The Serial Wire Debug Port (SW-DP). SW-DP provides a two-pin (clock + data) interface to the AHB-AP port. For more information, see *AHB Access Port* on page 11-35.

These alternative DP implementations provide different mechanisms for debug access to Cortex-M3. Your implementation might contain either, or both, of these components.

Note

- Only one DP can be used at once, and switching between the two debug ports should only be performed when neither DP is in use.
- Your implementation might contain an additional implementor-specific DP in parallel to SW_DP or JTAG-DP. See your implementor for details.

For more information on the AHB-AP, see *AHB Access Port* on page 11-35.

The DP and AP together are referred to as the *Debug Access Port* (DAP).

12.2 JTAG-DP

JTAG-DP contains a DP state machine (JTAG) that controls the JTAG-DP operation, including controlling the scan chain interface that provides the external physical interface to the JTAG-DP. It is based closely on the JTAG TAP State Machine, see *IEEE Std 1149.1-1990*. This chapter describes both the JTAG-DP and its scan chain interface.

Figure 12-1 on page 12-4 shows an ARM Debug Interface with a JTAG-DP, including the operation of the scan chain interface.

This section contains the following:

- *The scan chain interface*
- *IR scan chain and IR instructions* on page 12-7
- *DR scan chain and DR registers* on page 12-10.

12.2.1 The scan chain interface

The JTAG-DP comprises:

- a DP State Machine (JTAG)
- an *Instruction Register* (IR) and associated IR scan chain, used to control the behavior of the JTAG and the currently-selected data register
- a number of *Data Registers* (DRs) and associated DR scan chains, that interface to the registers in the JTAG-DP.

Physical connection to the JTAG-DP

Table 12-1 lists the JTAG-DP JTAG connections.

Table 12-1 JTAG-DP signal connections

Port	Direction	Description
TDI	Input	Debug Data In
TDO	Output	Debug Data Out
TCK	Input	Debug Clock
TMS	Input	Debug Mode Select
nTRST	Input	Debug Reset

Note

The **nTRST** input to JTAG-DP can be driven from Power On Reset, and does not have to be connected to a dedicated pin.

The recommended physical connection to the JTAG-DP is shown in Figure 12-1.

Figure 12-1 JTAG-DP physical connection

The DAP State Machine (JTAG)

Figure 12-2 on page 12-5 shows the JTAG state machine.

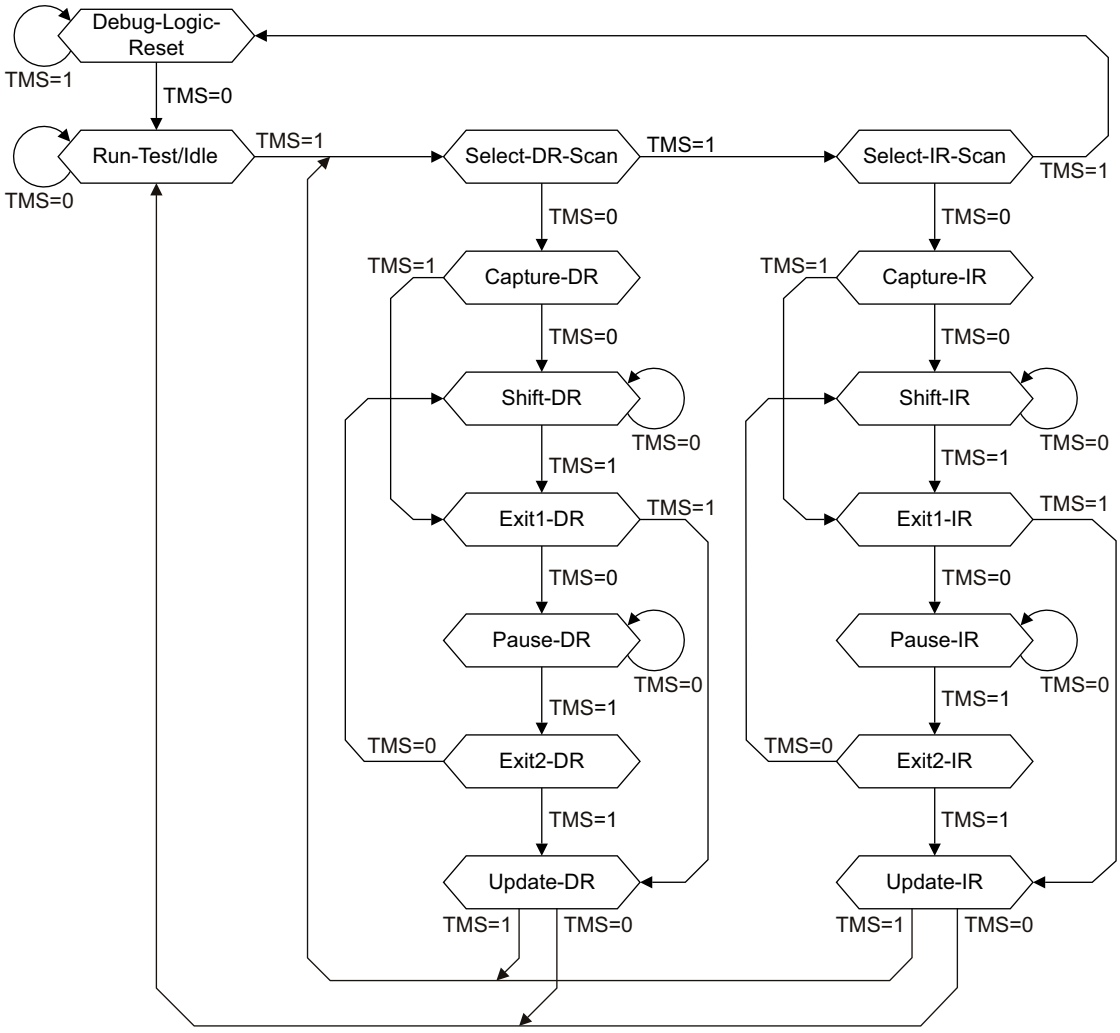


Figure 12-2 The DAP State Machine (JTAG)

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

Basic operation of the JTAG-DP

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

Referring to the DAP State Machine (JTAG) shown in Figure 12-2 on page 12-5:

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

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

  Note

  This is a change from the behavior of previous versions of the ARM Debug Interface based on the IEEE JTAG standard. From ARM Debug Interface v5, debuggers do not have to gate the DAP clock to obtain a true rest state.

The behavior of the IR and DR scan chains is described in more detail in *IR scan chain and IR instructions* on page 12-7 and *DR scan chain and DR registers* on page 12-10.

The **nTRST** signal only resets the JTAG state machine logic. **nTRST** asynchronously takes the JTAG state machine logic to the Debug-Logic-Reset state. As shown in Figure 12-2 on page 12-5, the Debug-Logic-Reset state can also always be entered synchronously from any state by a sequence of five **TCK** cycles with **TMS** high. However, depending on the initial state of the JTAG, this might take the state machine through one of the Update states, with the resulting side effects.

Within the DAP:

- the DP registers are only reset on a power-on reset.

12.2.2 IR scan chain and IR instructions

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

The JTAG Instruction Register (IR)

Purpose Holds the current DAP Controller instruction.

Length 4 bits.

Operating mode

When in Shift-IR state, the shift section of the IR is selected as the serial path between **TDI** and **TDO**. At the Capture-IR state, the binary value b0001 is loaded into this shift section. This is shifted out, least significant bit first, during Shift-IR. As this happens, a new instruction is shifted in, least significant bit first. At the Update-IR state, the value in the shift section is loaded into the IR so it becomes the current instruction.

On debug logic reset, IDCODE becomes the current instruction, see *The JTAG Device ID Code Register (IDCODE)* on page 12-10.

Order Figure 12-3 shows the bit order of the Instruction Register.

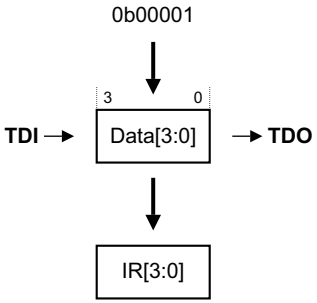


Figure 12-3 JTAG Instruction Register bit order

This register is mandatory in the IEEE 1149.1 standard.

The IR instructions

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

Unused IR instruction values select the Bypass register, described in *The JTAG Bypass Register (BYPASS)* on page 12-10.

Table 12-2 Standard IR instructions

IR instruction value	JTAG-DP register	DR scan width	See section
b0xxx	-	-	*Implementation-defined extensions to the IR instruction set*
b1000	ABORT	35	*The JTAG-DP Abort Register (ABORT)* on page 12-18
b1001	-	-	-
b1010	DPACC	35	*The JTAG DP/AP Access Registers (DPACC/APACC)* on page 12-11
b1011	APACC	35	
b110x	-	-	-
b1110	IDCODE	32	*The JTAG Device ID Code Register (IDCODE)* on page 12-10
b1111	BYPASS	1	*The JTAG Bypass Register (BYPASS)* on page 12-10

Implementation-defined extensions to the IR instruction set

The eight IR instructions b0000 to b0111 are reserved for Implementation-defined extensions to the JTAG-DP. These registers might be used for accessing a boundary scan register, for IEEE 1149.1 compliance. The required instructions are listed in Table 12-3 on page 12-9.

Note

- ARM Limited recommends that separate JTAG TAPs are used for boundary scan and debug.
- If the IR register is set to an IR instruction value that is not implemented, or reserved, then the Bypass Register is selected.

Table 12-3 Recommended implementation-defined IR instructions for IEEE 1149.1-compliance

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

a. Reserved, if the instruction is not implemented.

If you require a boundary scan implementation then you must implement the instructions that are shown as required by IEEE 1149.1 in a wrapper to JTAG-DP. The other IR instruction values listed in Table 12-3 are Reserved encodings that should be used if that function is implemented in the boundary scan. If implemented, these instructions must behave as required by the IEEE 1149.1 specification. If not implemented, they select the Bypass register.

The IEEE 1149.1 specification also requires the IDCODE and BYPASS instructions. However these are included in Table 12-2 on page 12-8.

The CLAMPZ instruction

CLAMPZ is not an IEEE 1149.1 instruction.

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

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

The JTAG Bypass Register (BYPASS)

Purpose Bypasses the device, by providing a direct path between **TDI** and **TDO**.

Length 1 bit.

Operating mode

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

- in the Shift-DR state, data is transferred from **TDI** to **TDO** with a delay of one TCK cycle
- in the Capture-DR state, a logic 0 is loaded into this register
- nothing happens at the Update-DR state.

Order Figure 12-4 shows the operation of the Bypass Register.

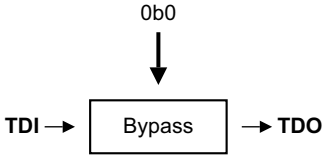


Figure 12-4 JTAG Bypass Register operation

This register is mandatory in the IEEE 1149.1 standard.

12.2.3 DR scan chain and DR registers

There are five physical DR registers:

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

There is a scan chain associated with each of these registers. As described in *IR scan chain and IR instructions* on page 12-7, the value in the IR register determines which of these scan chains is connected to the **TDI** and **TDO** signals.

The JTAG Device ID Code Register (IDCODE)

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

This is the JTAG-DP implementation of the Identification Code Register, see *The Identification Code Register, IDCODE* on page 12-52.

Length 32 bits.

Operating mode

When the IDCODE instruction is the current instruction in the IR, the shift section of the Device ID Code Register is selected as the serial path between **TDI** and **TDO**:

- in the Capture-DR state, the 32-bit device ID code is loaded into this shift section
- in the Shift-DR state, this data is shifted out, least significant bit first
- the shifted-in data is ignored at the Update-DR state.

Order Figure 12-5 shows the bit order of the Device ID Code Register.

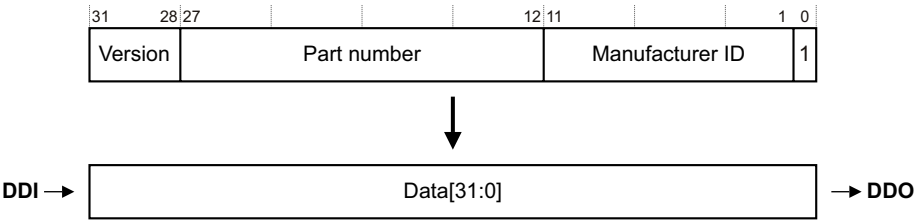


Figure 12-5 JTAG Device ID Code Register bit order

The JTAG DP/AP Access Registers (DPACC/APACC)

The DPACC and APACC scan chains have the same format.

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

The DPACC is used to access the CTRL/STAT, SELECT and RDBUFF registers, see *JTAG-DP register map* on page 12-47.

The APACC is used to access all of the AP registers, see *Summary and description of the AHB-AP registers* on page 11-35 for details of accessing AHB-AP registers, and *JTAG-DP Registers* on page 12-47 for details of accessing JTAG-AP registers.

Length 35 bits.

Operating mode

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

- In the Capture-DR state, the result of the previous transaction, if any, is returned, together with a 3-bit ACK response. Only two ACK responses are implemented, and these are summarized in Table 12-4.

Table 12-4 DPACC and APACC ACK responses

Response	ACK[2:0] encoding	See:
OK/FAULT	b010	*The OK/FAULT response to a DPACC or APACC access* on page 12-13
WAIT	b001	*The WAIT response to a DPACC or APACC access* on page 12-15

All other ACC encodings are Reserved.

- In the Shift-DR state, this data is shifted out, least significant bit first. As shown in Figure 12-6 on page 12-13, the first three bits of data shifted out are ACK[2:0], and therefore you can check the ACK response without shifting out all of the returned data, see *The WAIT response to a DPACC or APACC access* on page 12-15.

  As the returned data is shifted out to **TDO**, new data is shifted in from **TDI**. This is described in *The OK/FAULT response to a DPACC or APACC access* on page 12-13.
- Operation in the Update-DR depends on whether the ACK[2:0] response was OK/FAULT or WAIT. The two cases are described in:
  - *Update-DR operation following an OK/FAULT response* on page 12-13
  - *Update-DR operation following a WAIT response* on page 12-15.

Order Figure 12-6 shows the bit order of the DP and AP Access Registers.

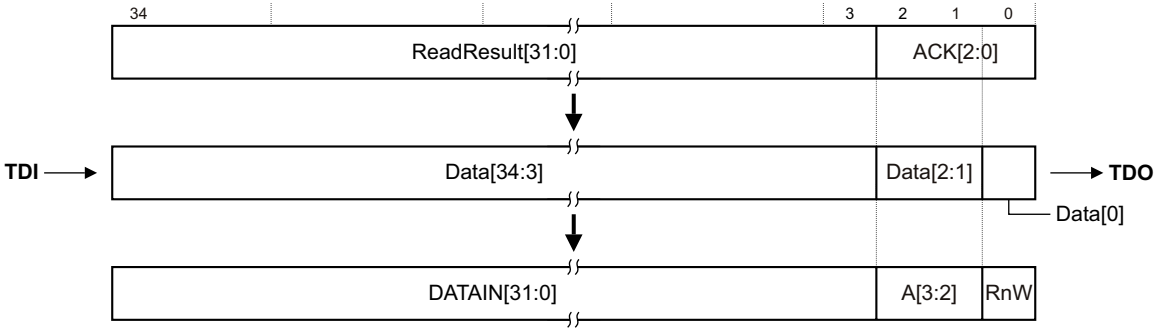


Figure 12-6 Bit order of JTAG DP and AP Access Registers

The OK/FAULT response to a DPACC or APACC access

If the response indicated by ACK[2:0] is OK/FAULT, the previous transaction has completed. The response code does not show whether the transaction completed successfully or was faulted. You must read the CTRL/STAT register to find whether the transaction was successful, see *The Control/Status Register, CTRL/STAT* on page 12-53:

- If the previous transaction was a read that completed successfully, then the captured ReadResult[31:0] is the requested register value. This result is shifted out as Data[34:3].
- If the previous transaction was a write, or a read that did not complete successfully, then the captured ReadResult[31:0] is Unpredictable, and if Data[34:3] is shifted out it must be discarded.

Update-DR operation following an OK/FAULT response

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

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

In either case:

- If RnW is shifted in as 0, the request is to write the value in DATAIN[31:0] to the addressed register.

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

The required register is addressed:

- In the case of a DPACC access, to read a DP register, by the value shifted into A[3:2]. See *JTAG-DP register map* on page 12-47 for the addressing details.
- In the case of a APACC access, to read an AP register, by the combination of:
  - the value shifted into A[3:2]
  - the current value of the SELECT register in the DP, see *The AP Select Register, SELECT* on page 12-57.

  For details of the register addressing, see:
  - Table 11-27 on page 11-35, if you want to access an AHB-AP register

Register accesses can be pipelined, because a single DPACC or APACC scan can return the result of the previous read operation at the same time as requesting another register access. At the end of a sequence of pipelined register reads, you can read the DP RDBUFF Register to return the result of the final register read. Reading the DP RDBUFF Register is benign, that is, it has no effect on the operation of the JTAG, see *The Read Buffer, RDBUFF* on page 12-59. The section *Target response summary* on page 12-16 gives more information about how one DPACC or APACC scan returns the result from the previous scan.

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

- If any sticky flag is set in the DP CTRL/STAT Register, the transaction is discarded. The next scan returns an OK/FAULT response immediately. For more information see *Sticky flags and DP error responses* on page 12-41, and *The Control/Status Register, CTRL/STAT* on page 12-53.
- If pushed compare or pushed verify operations are enabled then the scanned-in value of RnW *must* be 0, otherwise behavior is Unpredictable. On Update-DR, a read request is issued, and the returned value compared against DATAIN[31:0]. The STICKYCMP flag in the DP CTRL/STAT register is updated based on this comparison. For more information see *Pushed compare and pushed verify operations* on page 12-44. Pushed operations are enabled using the TRNMODE field of the DP CTRL/STAT register, see *The Control/Status Register, CTRL/STAT* on page 12-53 for more information.

- The AP access does not complete until the AP signals it as completed. For example, if you access a Memory Access Port (AHB-AP), the access might cause an access to a memory system connected to the AHB-AP. In this case, the access does not complete until the memory system signals to the AHB-AP that the memory access has completed.

The WAIT response to a DPACC or APACC access

A WAIT response indicates that the previous transaction has not completed. The host should retry the DPACC or APACC access.

Note

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

Normally, if software detects a WAIT response, it retries the same transfer. This enables the protocol to process data as quickly as possible. However, if the software has retried a transfer a number of times, allowing enough time for a slow interconnect and memory system to respond, it might write to the ABORT register, to cancel the operation. This signals to the active AP that it should terminate the transfer it is currently attempting, and permits access to other parts of the debug system. An AP might not be able to terminate a transfer on its ASIC interface. However, on receiving an ABORT, the AP should free its JTAG interface.

Update-DR operation following a WAIT response

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

Note

You can detect a WAIT response without shifting through the entire DP or AP Access Register, see the response details in Table 12-4 on page 12-12.

Sticky overrun behavior on DPACC and APACC accesses

At the Capture-DR state, if the previous transaction has not completed a WAIT response is generated. When this happens, if the Overrun Detect flag is set, the Sticky Overrun flag, STICKYORUN, is set. See *The Control/Status Register, CTRL/STAT* on page 12-53 for more information about the Overrun Detect and Sticky Overrun flags.

While the previous transaction remains not completed, subsequent scans also receive a WAIT response.

Once the previous transaction has completed, further APACC transactions are abandoned and scans respond immediately with an OK/FAULT response. However, DP registers can be accessed. In particular the CTRL/STAT register can be accessed, to confirm that the Sticky Overrun flag is set, and to clear the flag after gathering any required information about the overrun condition. See *Overrun detection* on page 12-42 for more information.

Minimum response times

As explained in *The OK/FAULT response to a DPACC or APACC access* on page 12-13, a DP or AP register access is initiated at the Update-DR state of one DPACC or APACC access, and the result of the access is returned at the Capture-DR state of the following DPACC or APACC access. However, the second access generates a WAIT response if the requested register access has not completed.

The JTAG clock, **TCK**, is asynchronous to the internal clock of the system being debugged, and the time required for an access to complete includes clock cycles in both domains. However, the timing between the Update-DR state and the Capture-DR state only includes **TCK** cycles. Referring to Figure 12-2 on page 12-5, there are two paths from the Update-DR state, where the register access is initiated, to the Capture-DR state, where the response is captured:

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

If the second path is followed, the state machine can spend any number of **TCK** cycles spinning in the Run-Test/Idle state. This means it is possible to vary the number of **TCK** cycles between the Update-DR and Capture-DR states.

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

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

Target response summary

As described in *The OK/FAULT response to a DPACC or APACC access* on page 12-13 and *Minimum response times*, a DP or AP register access is initiated at the Update-DR state of one DPACC or APACC access, and the result of the access is returned at the

Capture-DR state of the following DPACC or APACC access. Table 12-5 summarizes the target responses, at the Capture-DR state, for every possible DPACC and APACC access in the previous scan.

Note

The target responses shown in Table 12-5 are independent of the operation being performed in the current DPACC or APACC scan. In the table, *Read result* is the data shifted out as Data[34:3], and ACK is decoded from the data shifted out as Data[2:0].

Table 12-5 JTAG target response summary

Previous scan, at Update-DR state[a]				Current scan, at Capture-DR state			Notes
R/W	IR	ADDR [b]	Sticky[c]	AP state[d]	Read result	ACK	
X	X	bXX	X	Busy	UNP[e]	WAIT	Can cause Sticky Overrun flag to be set.[f]
R	DPACC	b01	X	Not Busy	CTRL/STAT	OK/FAULT	Returns CTRL/STAT value.
		b10			SELECT		Returns SELECT value.
		b00 or b11			0x00000000		No readable DP registers at addresses b00 and b11.
W	DPACC	b01	X	Not Busy	UNP[e]	OK/FAULT	Write to CTRL/STAT.
		b10					Write to SELECT.
		b00 or b11					Write ignored.
R	APACC	bXX	No	Ready	See Notes	OK/FAULT	See footnote[g].
				Error	UNP[e]		Sticky Error flag is set.
W	APACC	bXX	No	Ready	UNP[e]	OK/FAULT	See footnote[h].
				Error	UNP[e]		Sticky Error flag is set.
X	APACC	bXX	Yes	X	UNP[e]	OK/FAULT	Previous transaction was discarded.

a. The Previous scan is the most recent scan for which the ACK response at the Capture-DR state was OK/FAULT. Updates made following a WAIT response are discarded.
b. ADDR[3:2] in the DPACC or APACC access.

c. The Sticky column indicates whether any Sticky flag is set in the DP CTRL/STAT register, see *The Control/Status Register, CTRL/STAT* on page 12-53.
d. The state of the AP when the current scan reaches the Capture-DR state, or the response from the AP at that time.
e. UNP = Unpredictable.
f. If the Overrun Detect flag is set then this access/response sequence causes the Sticky Overrun flag to be set. See *The Control/Status Register, CTRL/STAT* on page 12-53.
g. If Pushed Verify or Pushed Compare is enabled, the behavior is Unpredictable. Otherwise, returns the value of the AP Register addressed on the previous scan.
h. If Pushed Verify or Pushed Compare is enabled, the previous transaction performed the required pushed operation, which might have set the Sticky Compare flag, see *Pushed compare and pushed verify operations* on page 12-44. Otherwise, the data captured at the previous scan has been written to the AP register requested.

Host response summary

The ACK column, for the *Current scan, at Capture-DR* state section of Table 12-5 on page 12-17, shows the responses the host might receive after initiating a DPACC or APACC access.

Table 12-6 Summary of JTAG host responses

JTAG access type	ACK from target	Suggested host action in response to ACK
Read	OK/FAULT	Capture read data.
Write	OK/FAULT	No more action required.
Read or Write	WAIT	Repeat the same access until either an OK/FAULT ACK is received or the wait timeout is reached. If necessary, use the DAP ABORT register to enable access to the AP.
Read or Write	Invalid ACK	Assume a target or line error has occurred and treat as a fatal error.

The JTAG-DP Abort Register (ABORT)

Purpose Access the DP Abort Register, to force a DAP abort.

This is the JTAG-DP implementation of the Abort Register, see *The Abort Register, ABORT* on page 12-49.

Length 35 bits.

Operating mode

When the ABORT instruction is the current instruction in the IR, the serial path between **TDI** and **TDO** is connected to a 35-bit scan chain that is used to access the Abort Register.

The debugger must scan the value `0x0000008` into this scan chain. This value:

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

Caution

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

Order Figure 12-7 shows the bit order of the ABORT scan chain.

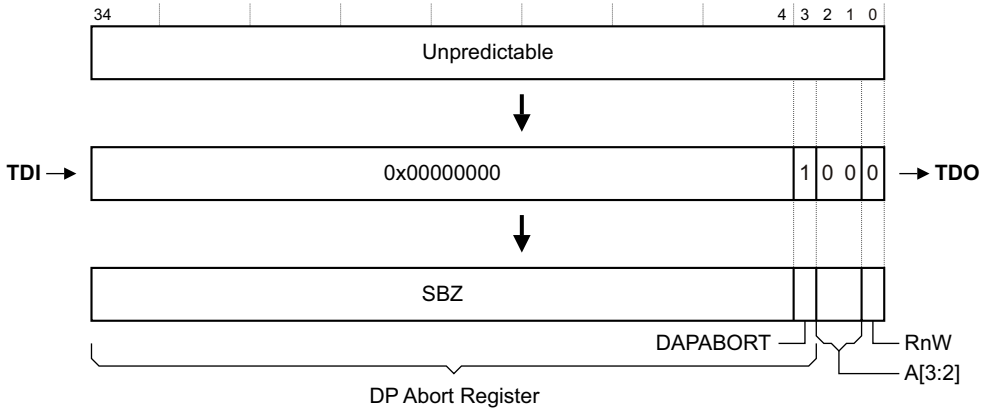


Figure 12-7 JTAG-DP ABORT scan chain bit order

12.3 SW-DP

This section gives an architectural description of the ARM Serial Wire Debug (SW-DP) interface. In particular, it describes the Serial Wire Debug (SWD) protocol, and how this protocol provides access to the DP registers. These registers are described in detail in *Debug Port Programmer's Model* on page 12-47.

The SW-DP operates with a synchronous serial interface. This uses a single bi-directional data signal, and a clock signal.

Each sequence of operations on the wire consists of two or three phases:

Packet request

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

Acknowledge response

The target sends an acknowledge response to the host.

Data transfer phase

This phase is only present if a data read or data write request has been followed by a valid (OK) acknowledge response, or if the ORUNDETECT flag is set in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 12-53.

The data transfer can be target to host, following a read request, or host to target, following a write request.

Note

If the Overrun Detect bit is set in the DP CTRL/STAT Register then a data transfer phase is required on all responses, including WAIT and FAULT. For details of the CTRL/STAT Register see *The Control/Status Register, CTRL/STAT* on page 12-53.

12.3.1 Clocking

The SW-DP clock, **DBGCLK**, can be asynchronous to the Cortex-M3 clock. **DBGCLK** can be stopped when the debug port is idle. **DBGCLK** must be clocked for two additional cycles after the final data bit has been transmitted over the wire.

12.3.2 Overview of debug interface

This section gives an overview of the physical interface used by SW-DP.

Line interface

SW-DP uses a serial wire for both host and target sourced signals. The host emulator drives the protocol timing - only the host emulator generates packet headers.

SW-DP operates in synchronous mode, and requires a clock pin and a data pin.

Synchronous mode uses a clock reference signal, which can be sourced from an on-chip source and exported, or provided by the host device. This clock is then used by the host as a reference for generation and sampling of data so that the target is not required to perform any oversampling.

Both the target and host are capable of driving the bus HIGH and LOW, or tristating it. The ports must be able to tolerate short periods of contention to allow for loss of synchronization.

Line pullup

Both the host and target are able to drive the line HIGH or LOW, so it is important to ensure that contention does not occur by providing undriven time slots as part of the handover. So that the line can be assumed to be in a known state when neither is driving the line, a 100kOhm pullup is required at the target, but this can only be relied on to maintain the state of the wire. If the wire is driven LOW and released, the pullup resistor eventually brings the line to the HIGH state, but this takes many bit periods.

The pullup is intended to prevent false detection of signals when no host device is connected. It must be of a high value to reduce IDLE state current consumption from the target when the host actively pulls down the line.

Note

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

Line turn-round

To avoid contention, a turnaround period is required when the device driving the wire changes.

Idle and reset

Between transfers, the host must either drive the line low to the IDLE state, or continue immediately with the start bit of a new transfer. The host is also free to leave the line HIGH, either driven or tristated, after a packet. This reduces the static current drain, but if this approach is used with a free running clock, a minimum of 50 clock cycles must be used, followed by a READ-ID as a new re-connection sequence.

There is no explicit reset signal for the protocol. A reset is detected by either host or target when the expected protocol is not observed. It is important that both ends of the link become reset before the protocol can be restarted with a training sequence. Re-synchronization following the detection of protocol errors or after reset is achieved by providing 50 clock cycles with the wire HIGH, or tristate, followed by a read ID request. If the SW-DP detects that it has lost synchronization, for example no stop bit is seen when expected, it leaves the line undriven and waits for the host to either re-try with a new header, or signals a reset by not driving the line itself. If the SW-DP detects two bad data sequences in a row, it locks out until a reset sequence of 50 clock cycles with DBGDI high is seen.

If the host does not see an expected response from SW-DP, it must allow time for SW-DP to return a data payload. The host can then retry with a read to the SW-DP ID code register. If this is unsuccessful, the host must attempt a reset.

12.3.3 Overview of protocol operation

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

The sequences of operations illustrated here are:

- *Successful write operation (OK response)* on page 12-24
- *Successful read operation (OK response)* on page 12-25
- *WAIT response to Read or Write operation request* on page 12-25
- *FAULT response to Read or Write operation request* on page 12-26
- *Protocol error sequence* on page 12-27.

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

Key to illustrations of operations

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

Start A single start bit, with value 1.

DPnAP A single bit, indicating whether the Data Port or the Access Port Access Register is to be accessed. This bit is 0 for an APACC access, or 1 for a DPACC access.

RnW A single bit, indicating whether the access is a read or a write. This bit is 0 for an write access, or 1 for a read access.

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

- For an APACC access, the register being addressed depends on the ADDR[3:2] value and the value held in the SELECT register. For details of the addressing see:
  - Table 11-27 on page 11-35, if you want to access a AHB-AP register.

  For details of the SELECT register see *The AP Select Register, SELECT* on page 12-57.
- For a DPACC access, the A[3:2] value determines which register is accessed, see Table 12-13 on page 12-47.

Note

The A[3:2] value is transmitted LSB-first on the wire.

Parity A single parity bit for the preceding packet.

Note

- Parity bits are used to protect the header part of the packet, and the data part separately. The ACK bits are redundantly encoded already, and do not use any additional protection.
- To calculate the parity bit for the header, count the number of bits set out of **DP/AP**, **RnW**, and **ADDR[0:1]**. If this is odd (one or three) then the parity bit should be set, to make the total number of 1s even. To calculate the parity bit for a data field, count the number of bits set in just the 32 data bits. If this is odd, then the parity bit should be set, to make the total number of 1s even.

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

Park A single bit. The host does not drive the line for this bit, and the line is pulled high by the SWD interface hardware. The target reads this bit as 1.

Trn Turnaround. This is a period during which the line is not driven and the state of the line is Undefined. The length of the turnaround period is controlled by the TURNROUND field in the Wire Control Register, see *The Wire Control Register, WCR (SW-DP only)* on page 12-60. The default setting is a turnaround period of one clock cycle.

Note

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

There are additional turnaround periods in the asynchronous SWD protocol.

Ack A three-bit target-to-host response. These bits appear on the wire LSB-first.

WDATA[0:31]

32 bits of write data, from host to target.

Note

The WDATA[0:31] value is transmitted LSB-first on the wire.

RDATA[0:31]

32 bits of read data, from target to host.

Note

The RDATA[0:31] value is transmitted LSB-first on the wire.

Successful write operation (OK response)

A successful write operation consists of three phases:

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

By default, there are single-cycle turnaround periods between each of these phases. See the description of **Trn** in *Key to illustrations of operations* on page 12-22 for more information.

A successful write operation is shown in Figure 12-8 on page 12-25.

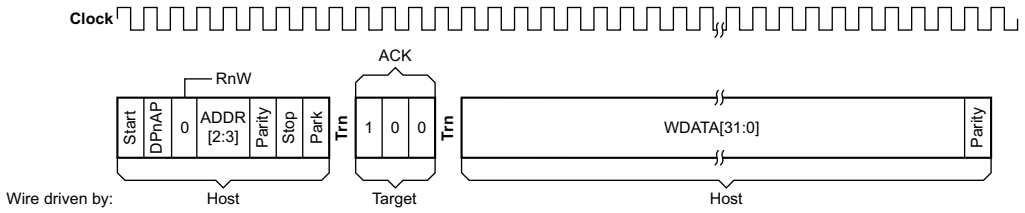


Figure 12-8 Serial Wire Debug successful write operation

Successful read operation (OK response)

A successful read operation consists of three phases:

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

By default, there are single-cycle turnaround periods between the first and second of these phases, and after the third phase. See the description of **Trn** in *Key to illustrations of operations* on page 12-22 for more information. However, there is no turnaround period between the second and third phases.

A successful read operation is shown in Figure 12-9.

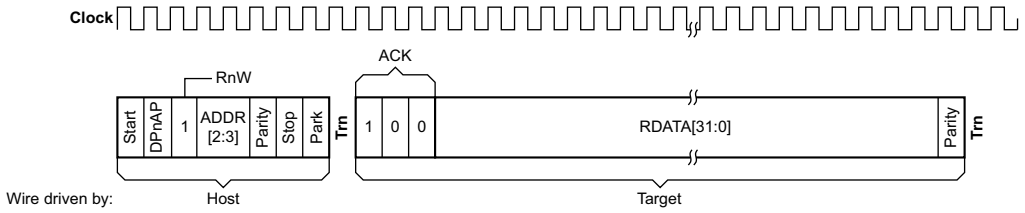


Figure 12-9 Serial Wire Debug successful read operation

WAIT response to Read or Write operation request

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

- an eight-bit read or write packet request, from the host to the target
- a three-bit WAIT acknowledge response, from the target to the host.

By default, there are single-cycle turnaround periods between these two phases, and after the second phase. See the description of **Trn** in *Key to illustrations of operations* on page 12-22 for more information.

A WAIT response to a read or write packet request is shown in Figure 12-10 on page 12-26.

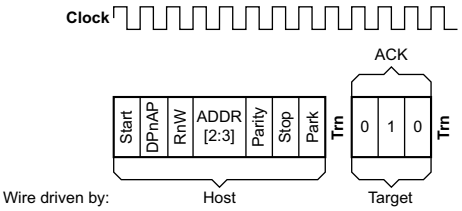


Figure 12-10 Serial Wire Debug WAIT response to a packet request

Note

If Overrun Detection is enabled then a data phase is required on a WAIT response. For more information see *Sticky overrun behavior* on page 12-30.

FAULT response to Read or Write operation request

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

- an eight-bit read or write packet request, from the host to the target
- a three-bit FAULT acknowledge response, from the target to the host.

By default, there are single-cycle turnaround periods between these two phases, and after the second phase. See the description of **Trn** in *Key to illustrations of operations* on page 12-22 for more information.

A FAULT response to a read or write packet request is shown in Figure 12-11.

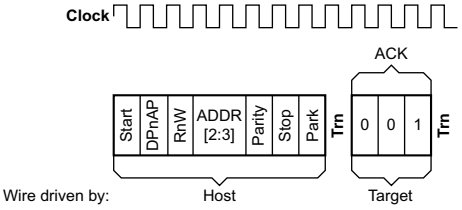


Figure 12-11 Serial Wire Debug FAULT response to a packet request

Note

If Overrun Detection is enabled then a data phase is required on a FAULT response. For more information see *Sticky overrun behavior* on page 12-30.

Protocol error sequence

A protocol error occurs when a host issues a packet request but the target fails to return any acknowledge response. This is illustrated in Figure 12-12.

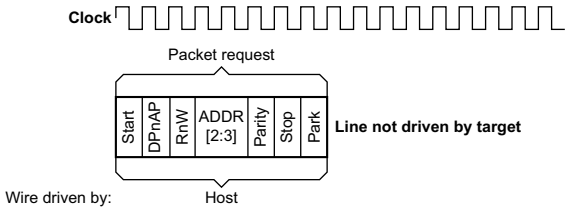


Figure 12-12 Serial Wire Debug protocol error after a packet request

12.3.4 Protocol description

This section provides additional information on the DAP Serial Wire Debug operations that were introduced in *Overview of protocol operation* on page 12-22.

Connection sequence

A connection sequence is used to ensure that hot-plugging does not result in unintentional transfers. To ensure that the SW-DP is correctly synchronized to the header that is used to signal a connection, a wire reset sequence of 50 clock cycles must be sent first. After the host has transmitted a reset sequence to the SW-DP, a valid header request for a read of the ID register is accepted, and the normal response and data returned. Having the host make an ID Code Register read to exit the training state ensures a high level of confidence that correct packet frame alignment has been achieved.

The OK response

When it receives a packet request from the debug host, the SW-DP must respond immediately. It issues an OK response, indicated by an acknowledge phase of b001, if it is ready for the data phase of the transfer, if one is required.

Note

- As shown in *Overview of protocol operation* on page 12-22, there is always a turnaround between the end of the packet request from the host and the start of the acknowledgement from the SW-DP target. The default turnaround is exactly one serial clock cycle, but see the description of **Trn** in *Key to illustrations of operations* on page 12-22 for more information.

  There is a turnaround whenever there is a change in the direction of data transfer over the serial SWD connection. If an operation that is described as immediate involves a change in the data transfer direction then the operation must start immediately after the turnaround.

- All SWD transfers are made LSB-first. Therefore, the OK response of b001 appears on the wire as 1, followed by 0, followed by 0, as shown in Figure 12-8 on page 12-25 and Figure 12-9 on page 12-25.

If the host requested a write access it must start the write transfer immediately after receiving the acknowledgement from the target. This behavior is the same whether the write is to the DP or to an AP. However, the SW-DP can buffer AP writes, as described in *Access Port write buffering* on page 12-32.

If the host requested a read access to the DP then the SW-DP sends the read data immediately after the acknowledgement. Because there is no change in the data transfer direction between the acknowledgement and the read data there is no turnaround between these phases. This is shown in Figure 12-9 on page 12-25.

Read accesses to the AP are *posted*. This means that the result of the access is returned on the next transfer. If the next access you have to make is not another AP read then you must insert a read of the DP RDBUFF Register to obtain the posted result, see *The Read Buffer, RDBUFF* on page 12-59.

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

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

Operation and use of the READOK flag

The SW-DP CTRL/STAT register includes a READOK flag, bit [6]. This register is described in *The Control/Status Register, CTRL/STAT* on page 12-53.

The READOK flag is updated on every AP read access, and on every RDBUFF read request. When the SW-DP initiates the AP access it clears the READOK flag to 0, and when the SW-DP target gives an OK response to the read request it sets the READOK flag to 1.

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

- If the flag is set to 1 then the read was performed correctly. The host can use a RESEND request to obtain the read result, see *The Read Resend Register, RESEND (SW-DP only)* on page 12-62.
- If the flag is set to 0 then the read was not successful. The host must retry the original AP or RDBUFF read request.

The WAIT response

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

- a read of the IDCODE register, see *The Identification Code Register, IDCODE* on page 12-52
- a read of the CTRL/STAT register, see *The Control/Status Register, CTRL/STAT* on page 12-53
- a write to the ABORT register, see *The Abort Register, ABORT* on page 12-49.

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

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

Note

When overrun detection is enabled a WAIT response must include a data phase. See *Sticky overrun behavior* on page 12-30 for more information.

Normally, when a debugger receives a WAIT response it retries the same operation. This enables it to process data as quickly as possible. However, if several retries have been attempted, and time allowed for a slow interconnection and memory system to respond, if appropriate, the debugger might write to the ABORT register. This signals to the active AP that it must terminate the transfer that it is currently attempting. An AP implementation might be unable to terminate a transfer on its ASIC interface. However, on receiving an ABORT request the AP must free up the Serial Wire Debug interface.

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

The FAULT response

SW-DP does not issue a FAULT response to an access to the IDCODE, CTRL/STAT or ABORT registers. For any other access, the SW-DP issues a FAULT response if any sticky flag is set in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 12-53. See *Sticky overrun behavior* for more information about the sticky overrun flag.

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

The sticky error flags are cleared by writing bits in the ABORT register, see *The Abort Register, ABORT* on page 12-49.

Sticky overrun behavior

If SW-DP receives a transaction request when the previous transaction has not completed it generates a WAIT response. If overrun detection is enabled in the CTRL/STAT Register, the STICKYORUN flag is set to 1 in that register. For more information see *The Control/Status Register, CTRL/STAT* on page 12-53. Subsequent transactions generate FAULT responses, because a sticky flag is set.

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

- if the transaction is a read the data in the data phase is Unpredictable
- if the transaction is a write the data phase is ignored.

Figure 12-13 on page 12-31 shows the WAIT or FAULT response to a read operation when overrun detection is enabled, and Figure 12-14 on page 12-31 shows the response to a write operation when overrun detection is enabled.

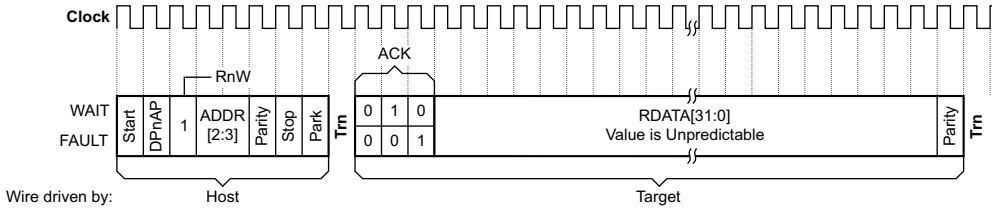


Figure 12-13 Serial Wire WAIT or FAULT response to a read operation when overrun detection is enabled

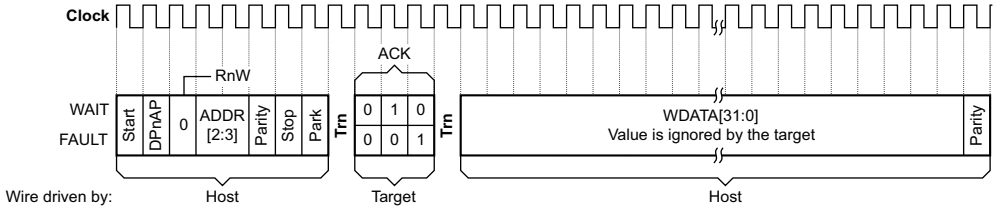


Figure 12-14 Serial Wire WAIT or FAULT response to a write operation when overrun detection is enabled

Protocol Error responses

If SW-DP detects a parity error in the packet request it does not reply to the request.

When the host receives no reply to its request, it must back off, in case the SW-DP mis-aligned within the packet frame, and could be about to return read data. The back-off period should correspond to the period required for a read transaction to complete. After this, it can issue a new transfer request. In this situation it *must* read the IDCODE register, see *The Identification Code Register, IDCODE* on page 12-52. This is mandated by this specification because a successful read of the IDCODE register confirms that the target is operational.

If there is no response at the second attempt, the debugger should force a line reset and then initiate retraining. This is necessary because the SW-DP is in a state where it only responds to a debug request. If the transfer that resulted in the original protocol error response was a write you can assume safely that no write occurred. If the original transfer was a read it is possible that the read was issued to an AP. Although this is unlikely, the possibility is a significant consideration, because reads are pipelined and the AP can implement a FIFO.

Access Port write buffering

The SW-DP implements a write buffer, that enables it to accept write operations even when other transactions are still outstanding. The DP issues an OK response to a write request if it can accept the write into its write buffer. This means that an OK response to a write request, other than a write to the DP ABORT Register, indicates only that the write has been accepted by the DP. It does not indicate that all previous transactions have completed.

If a write is accepted into the write buffer but later abandoned then the WDATAERR flag is set in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 12-53. A buffered write is abandoned if:

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

This means that if you make a series of AP write transactions, it might not be possible to determine which transaction failed from examining the ACK responses. However it might be possible to use other enquiries to find which write failed. For example, if you are using the auto-address increment (AddrInc) feature of a Memory Access Port (AHB-AP), then you can read the Transfer Address Register to find which was the final successful write transaction. See *AHB-AP Transfer Address Register* on page 11-38 and *Summary and description of the AHB-AP registers* on page 11-35 for more information.

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

- any AP read operation
- any DP operation other than a read of the IDCODE or CTRL/STAT Register, or a write of the ABORT Register.

Attempting these operations causes WAIT responses from the DP, until the write buffer is empty.

Note

If Pushed Verify or Pushed Compare is enabled, AP write transactions are converted into AP reads. These are then treated in the same way as other AP read operations. See *Pushed compare and pushed verify operations* on page 12-44 for details of these operations.

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

Summary of target responses

Table 12-7 summarizes the target SW-DP response to all possible debugger DP read operation requests.

Table 12-8 on page 12-34 summarizes the target SW-DP response to all possible debugger AP read operation requests.

Table 12-9 on page 12-35 summarizes the target SW-DP response to all possible debugger DP write operation requests, assuming the WDATA parity check is good.

Table 12-10 on page 12-36 summarizes the target SW-DP response to all possible debugger AP write operation requests, assuming the WDATA parity check is good.

Fault conditions that are not shown in these two tables are described in *Fault conditions not included in the target response tables* on page 12-36

Table 12-7 Target response summary for DP read transaction requests

ADDR [3:2]	Sticky flag set?	AP Ready?	SW-DP (target) response	
			ACK	Action
b00	X	X	OK	Respond with IDCODE value.
b01	X	X	OK	Respond with CRTL/STAT or WCR value[a].
b10	No	Yes	OK	Respond with RESEND value from previous AP read.
b11	No	Yes	OK	Respond with RDBUF value from previous AP read, and set READOK flag in CTRL/STAT Register.
b10	No	No	WAIT	No data phase, unless overrun detection is enabled[b].

Table 12-7 Target response summary for DP read transaction requests

ADDR [3:2]	Sticky flag set?	AP Ready?	SW-DP (target) response	
			ACK	Action
b10	Yes	X	FAULT	No data phase, unless overrun detection is enabled[b].
b11	No	No	WAIT	No data phase, unless overrun detection is enabled[b]. Clear READOK flag in CTRL/STAT Register.
b11	Yes	X	FAULT	No data phase, unless overrun detection is enabled[b]. Clear READOK flag in CTRL/STAT Register.

a. Which value is returned depends on the value of the CTRLSEL bit in the SELECT Register. in the DP. See *The AP Select Register, SELECT* on page 12-57.
b. See *Sticky overrun behavior* on page 12-30 for details of data phase when overrun detection is enabled.

Table 12-8 Target response summary for AP read transaction requests

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

a. If Pushed Verify or Pushed Compare is enabled, behavior is Unpredictable.
b. On the first of a sequence of AP reads, the value returned in the data phase is Unpredictable.
c. The AP register is addressed by the value of A[3:2] together with the value of the APBANKSEL field in the SELECT Register in the DP. See *The AP Select Register, SELECT* on page 12-57.
d. See *Sticky overrun behavior* on page 12-30 for details of data phase when overrun detection is enabled.

Table 12-9 Target response summary for DP write transaction requests

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

a. Writes might be accepted when other transactions are still outstanding, These writes might be abandoned subsequently. See *Access Port write buffering* on page 12-32 for more information.
b. See *Sticky overrun behavior* on page 12-30 for details of data phase when overrun detection is enabled.

Table 12-10 Target response summary for AP write transaction requests

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

a. Writes might be accepted when other transactions are still outstanding, These writes might be abandoned subsequently. See *Access Port write buffering* on page 12-32 for more information.
b. If Pushed Verify or Pushed Compare is enabled, the write is converted to a read of the addressed AP register, and the value returned by this read is compared with the supplied WDATA value, see *Pushed compare and pushed verify operations* on page 12-44 for more information. For an outline of how AP registers are addressed see footnote c to this table.
c. The AP register is addressed by the value of A[3:2] together with the value of the APBANKSEL field in the SELECT Register in the DP See *The AP Select Register, SELECT* on page 12-57.
d. See *Sticky overrun behavior* on page 12-30 for details of data phase when overrun detection is enabled.

Fault conditions not included in the target response tables

There are two fault conditions that are not included in possible operation requests listed in Table 12-7 on page 12-33 and Table 12-9 on page 12-35:

Protocol fault

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

WDATA fails parity check (write operations only)

The ACK response of the DP is sent before the parity check is performed, and can be found from Table 12-9 on page 12-35. When the parity check is performed and fails, the WDATAERR flag is set in the CTRL/STAT Register, see *The Control/Status Register, CTRL/STAT* on page 12-53.

Summary of host responses

Every access by a debugger to a SW-DP starts with an operation request. *Summary of target responses* on page 12-33 listed all possible requests from a debugger, and summarized how the SW-DP responds to each request.

Whenever a debugger issues an operation request to a SW-DP, it expects to receive a 3-bit acknowledgement, as listed in the ACK columns of Table 12-7 on page 12-33 and Table 12-7 on page 12-33. This section summarizes how the debugger must respond to this acknowledgement, for all possible cases. This is shown in Table 12-11.

Table 12-11 Summary of host (debugger) responses to the SW-DP acknowledge

Operation requested	ACK received	Host response	
		Data phase	Additional action
R	OK	Capture RDATA from target and check for valid parity and protocol.	Might have to re-issue original read request if parity or protocol fault and unable to flag data as invalid[a].
W	OK	Send WDATA.	Validity of this transfer is confirmed on next access.
X	WAIT	No data phase, unless overrun detection is enabled[b].	Normally, repeat the original operation request. See *The WAIT response* on page 12-29 for more information.
X	FAULT	No data phase, unless overrun detection is enabled[b].	Can send new headers, but only an access to DP register addresses b0X gives a valid response.
X	No ACK	Back off to allow for possible data phase.	Can attempt IDCODE Register read. Otherwise reset connection and retrain. See *Protocol Error responses* on page 12-31.
R	Invalid ACK	Back off to allow for possible data phase.	Can check CTRL/STAT Register to see if the response sent was OK.
W	Invalid ACK	Back off to ensure that target does not capture next header as WDATA.	Repeat the write access. A FAULT response is possible if the first response was sent as OK but not recognized as valid by the debugger. The subsequent write is not affected by the first, misread, response.

a. The host debugger might be able to support corrupted reads, or it might have to re-try the transfer.
b. If overrun detection is enabled, a data phase is required. On a read operation, the RDATA value is Unpredictable and the debugger must capture and discard this data. On a write operation the debugger must send a WDATA packet, that the target ignores.

12.3.5 Transfer timings

This section describes the interaction between the timing of transactions on the serial wire interface, and the DAP internal bus transfers. It shows when the target responds with a WAIT acknowledgement.

Figure 12-15 on page 12-38 shows the effect of signalling ACK = WAIT on the length of the packet.

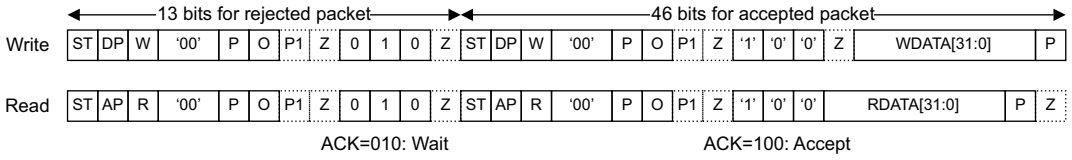


Figure 12-15 SW-DP acknowledgement timing

An access port access results in the generation of a transfer on the DAP internal bus. These transfers have an address phase and a data phase. The data phase can be extended by the access if it requires extra time to process the transaction, for example, if it has to perform an AHB access to the system bus to read data.

Table 12-12 shows the terms used in Figure 12-16 on page 12-39 to Figure 12-18 on page 12-40.

Table 12-12 Terms used in SW-DP timing

Term	Description
W.APACC	Write a DAP access port register.
R.APACC	Read a DAP access port register.
xxPACC	Read or write, to debug port or access port register.
WD[0]	First write packet data.
WD[-1]	Previous write packet data. A transaction that happened before the figures timeframe.
WD[1]	Second write packet data.
RD[0]	First read packet data.
RD[1]	Second read packet data.

Figure 12-16 on page 12-39 shows a sequence of write transfers. It shows that a single new transfer, WD[1], can be accepted by the serial engine, while a previous write transfer, WD[0], is completing. Any subsequent transfer must be stalled until the first transfer completes.

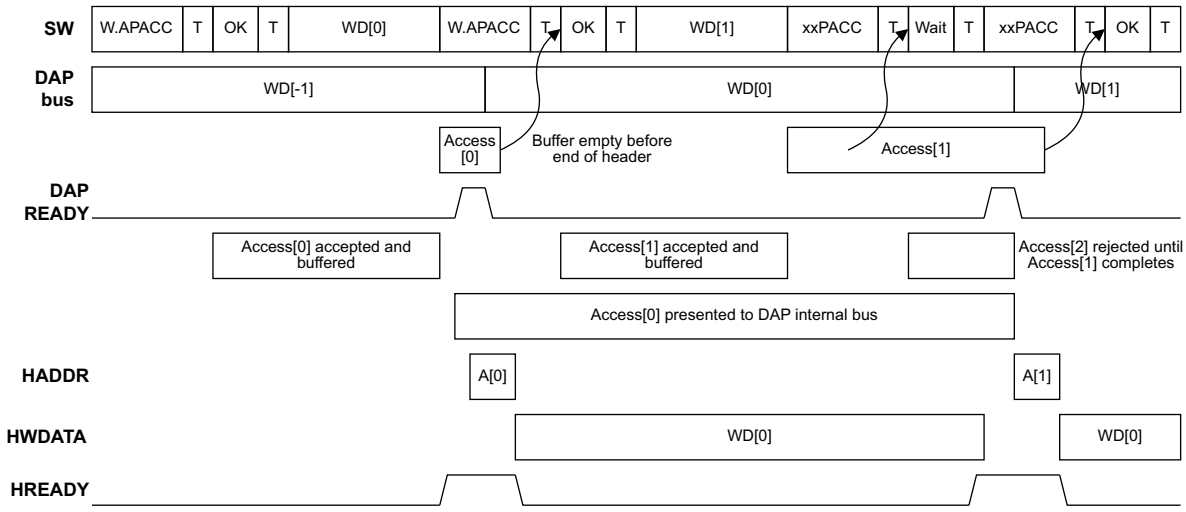


Figure 12-16 SW-DP to DAP bus timing for writes

Figure 12-17 shows a sequence of read transfers. It shows that the payload for an access port read transfer provides the data for the previous read request. A read transfer only stalls if the previous transfer has not completed, therefore the first read transfer returns undefined data. It is still necessary to return data to ensure that the protocol timing remains predictable.

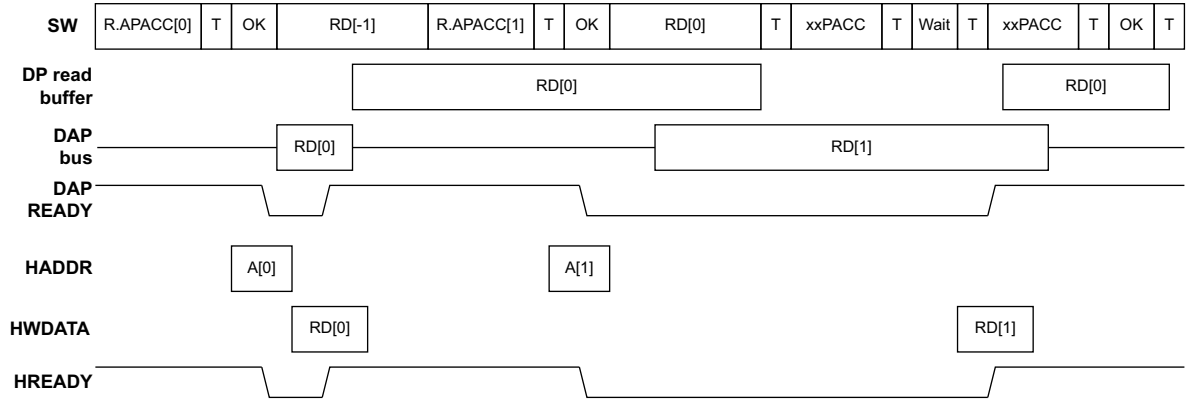


Figure 12-17 SW-DP to DAP bus timing for reads

Figure 12-18 shows a sequence of transfers separated by IDLE periods. It shows that the wire is always handed back to the host after any transfer.

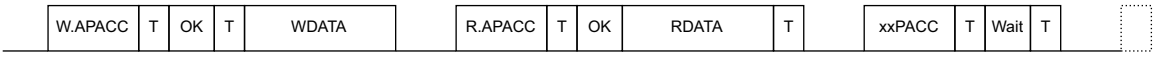


Figure 12-18 SW-DP idle timing

After the last bit in a packet, the line can be LOW, or idle, for any period longer than a single bit, to enable the Start bit to be detected for back-to-back transactions.

12.4 Common Debug Port (DP) features

This section describes features that are implemented by both SW-DP and JTAG-DP. These common features affect the way that a debugger is able to perform transactions with the DAP. It contains the following:

- *Sticky flags and DP error responses*.

12.4.1 Sticky flags and DP error responses

Within both a SW-DP and a JTAG-DP, sticky flags are used to indicate error conditions and to report the result of pushed compare and pushed verify operations. The different sticky flags are described in the following sections:

- *Read and write errors* on page 12-42
- *Overrun detection* on page 12-42
- *Protocol errors, SW-DP only* on page 12-43
- *Pushed compare and pushed verify operations* on page 12-44.

Note

When set, a sticky flag remains set until it is explicitly cleared. Even if the condition that caused the flag to be set no longer applies, the flag remains set until the debugger clears it. The method for clearing sticky flags is different for the JTAG-DP and the SW-DP. See *The Control/Status Register, CTRL/STAT* on page 12-53 for information about how these flags are cleared.

Errors can be returned by the DP itself, or can come from a debug register access, for example when the processor is powered down.

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

The DP response to an error condition might be:

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

This means that the debugger must check the Control/Status Register after performing a series of APACC transactions, to check if an error occurred. If a sticky flag is set, the debugger can clear the flag and then, if necessary, initiate more APACC transactions to

find the cause of the sticky flag condition. Because the flags are sticky, it is not necessary to check the flags after every transaction. The debugger only has to check the Control/Status Register periodically. This reduces the overhead of error checking.

12.4.2 Read and write errors

A read or write error might occur within the DP, or come from the system being debugged as the result of a (AHB-AP) access in response to an AP request. In either case, when the error is detected the Sticky Error flag, STICKYERR, in the Control/Status Register is set to b1.

A read/write error is also generated if the debugger makes an AP transaction request while the debug power domain is powered down.

12.4.3 Overrun detection

Debug Ports support an overrun detection mode. This mode enables an emulator on a high latency, high throughput connection to be sent blocks of commands. These should be sent with sufficient in-line delays to make overrun errors unlikely. However, if an overrun error occurs, the DP detects and flags the overrun errors, by setting a flag in the Control/Status Register. In overrun detection mode the debugger must check for overrun errors after each sequence of APACC transactions, by checking the Sticky Overrun flag in the Control/Status Register. It is not necessary for the emulator to react immediately to the overun condition.

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

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

In overrun detection mode, any other response, at any point, is treated as an error and causes the Sticky Overrun flag STICKYORUN in the DP Control/Status Register to be set to b1. The Sticky Error flag, STICKYERR, is not set.

The debugger must clear STICKYORUN to enable transactions to resume.

See *The Control/Status Register, CTRL/STAT* on page 12-53 for more information.

The behavior of the Debug Port when the Sticky Overrun flag is set is Debug Port defined.

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

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

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

12.4.4 Protocol errors, SW-DP only

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

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

If the SW-DP detects a parity error in a message header, the Debug Port does not respond to the message. The debugger must be aware of this possibility. If it does not receive a response to a message, the debugger must back off. It must then request a read of the IDCODE register, to ensure the Debug Port is responsive, before retrying the original access. For details of the IDCODE register see *The Identification Code Register, IDCODE* on page 12-52.

If the SW-DP detects a parity error in the data phase of a write transaction, it sets the Sticky Write Data Error flag, WDATAERR, in the Control/Status (CTRL/STAT) Register. Subsequent accesses from the debugger, other than IDCODE, CTRL/STAT or ABORT, results in a FAULT response. For details of the CTRL/STAT register see *The Control/Status Register, CTRL/STAT* on page 12-53.

On receiving a FAULT response from the SW-DP a debugger must read the CTRL/STAT register and check the sticky flag values. The WDATAERR flag is cleared by writing b1 to the WDERRCLR field of the Abort Register, see *The Abort Register, ABORT* on page 12-49.

12.4.5 Pushed compare and pushed verify operations

Both SW-DP and JTAG-DP Debug Ports support pushed operations, where the value written as an AP transaction is used at the DP level to compare against a target read:

- the debugger writes a value as an AP transaction
- the DP performs a read from the AP
- the DP compares the two values and updates the Sticky Compare flag, STICKYCMP, in the DP Control/Status register, based on the result of the comparison:
  - — pushed compare sets STICKYCMP to b1 if the values match
  - — pushed verify sets STICKYCMP to b1 if the values do not match.

  Whenever the STICKYCMP bit is set, on detection of a valid comparison, any outstanding transaction repeats are cancelled.

  For more information see *The Control/Status Register, CTRL/STAT* on page 12-53.

The DP includes a byte lane mask, so that the compare can be restricted to particular bytes in the word. This mask is set using the MASKLANE bits in the Control/Status register. For more information about this masking see *MASKLANE and the bit masking of the pushed compare and pushed verify operations* on page 12-56.

Figure 12-19 gives an overview of the pushed operations.

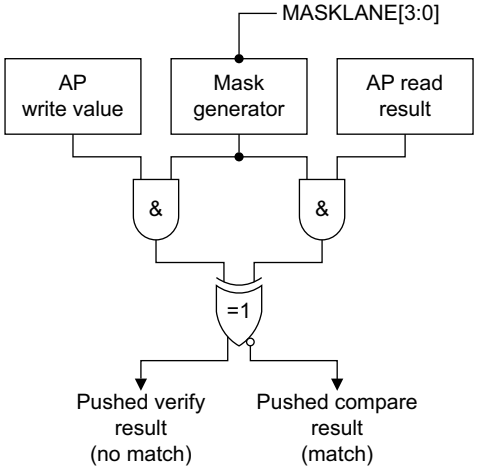


Figure 12-19 Pushed operations overview

Pushed operations improve performance where writes might be faster than reads. They are used as part of in-line tests, for example Flash ROM programming and monitor communication. Pushed operations are enabled using the Transaction Mode bits, TRNMODE, in the DP Control/Status Register, see *The Control/Status Register, CTRL/STAT* on page 12-53.

Considering pushed operations on a specific AP makes it easier to understand how these operations are implemented. On an AHB-AP, if you perform an AP write transaction to the Data Read/Write (DRW) Register, or to one of the Banked Data (BD0 to BD3) Registers, with either pushed compare or pushed verify active:

- The DP holds the data value from the AP write transaction in the pushed compare logic, see Figure 12-19 on page 12-44.
- The AP reads from the address indicated by the AP Transfer Address Register (TAR), see *AHB-AP Transfer Address Register* on page 11-38.
- The value returned by this read is compared with the value held in the pushed compare logic, and the STICKYCMP bit is set depending on the result. The comparison is masked as required by the MASKLANE bits. For more information see *The Control/Status Register, CTRL/STAT* on page 12-53.

As described, whenever an AP *write* transaction is performed with pushed compare or pushed verify active, the actual AP access that results is a *read* operation, not a write.

Note

Performing an AP read transaction with pushed compare or pushed verify active causes Unpredictable behavior.

On a Serial Wire DP, performing an AP read transaction with pushed compare or pushed verify active returns a value. This means the wire-level protocol remains coherent. However, the value returned is Unpredictable, and the read has Unpredictable side-effects.

Example use of pushed verify operation on a AHB-AP

You can use pushed verify to verify the contents of system memory.

- Make sure that the AHB-AP Control/Status Word (CSW) is set up to increment the Transfer Address Register (TAR) after each access. See *The Control/Status Register, CTRL/STAT* on page 12-53.
- Write to the Transfer Address Register (TAR) to indicate the start address of the Debug Register region that is to be verified, see *AHB-AP Transfer Address Register* on page 11-38.

- Write a series of expected values as AP transactions. On each write transaction, the DP issues an AP read access, compares the result against the value supplied in the AP write transaction, and sets the STICKYCMP bit in the CRL/STAT Register if the values do not match. See *The Control/Status Register, CTRL/STAT* on page 12-53.

  The TAR is incremented on each transaction.

In this way, the series of values supplied is compared against the contents of the AP locations, and STICKYCMP set if they do not match.

Example use of pushed find operation on a AHB-AP

You can use pushed find to search system memory for a particular word. If you use pushed find with byte lane masking you can search for one or more bytes.

- Make sure that the AHB-AP Control/Status Word (CSW) is set up to increment the TAR after each access. See *The Control/Status Register, CTRL/STAT* on page 12-53.
- Write to the Transfer Address Register (TAR) to indicate the start address of the Debug Register region that is to be searched. See *AHB-AP Transfer Address Register* on page 11-38.
- Write the value to be searched for as an AP write transaction. The DP repeatedly reads the location indicated by the TAR. On each DP read:
  - The value returned is compared with the value supplied in the AP write transaction. If they match, the STICKYCMP flag is set.
  - The TAR is incremented.

  This continues until STICKYCMP is set, or ABORT is used to terminate the search.

You could also use pushed find without address incrementing to poll a single location, for example to test for a flag being set on completion of an operation.

12.5 Debug Port Programmer's Model

Every Cortex-M3 system includes one, or both of:

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

This section contains:

- *JTAG-DP Registers*. This contains a summary of the JTAG-DP registers.
- *SW-DP Registers* on page 12-48. This contains a summary of the SW-DP registers.
- *Debug Port (DP) register descriptions* on page 12-49. This contains details of the DP registers, and describes implementation differences between SW-DP and JTAG-DP registers.

12.5.1 JTAG-DP Registers

The JTAG-DP register accessed depends on both:

- the Instruction Register (IR) value for the DAP access
- the address field of the DAP access.

For more information, see *Accessing the JTAG-DP registers* on page 12-48.

Table 12-13 shows the JTAG-DP register map.

Table 12-13 JTAG-DP register map

IR contents	Description	Address	Access	Reference	Notes
IDCODE	ID Code Register	-[a]	RO	*The Identification Code Register, IDCODE* on page 12-52	-
DPACC	-	0x0	RAZ/WI	-	Reserved. Read-as-zero, Writes ignored.
DPACC	DP Control/Status Register	0x4	R/W	*The Control/Status Register, CTRL/STAT* on page 12-53	-
DPACC	Select Register	0x8	R/W	*The AP Select Register, SELECT* on page 12-57	-

Table 12-13 JTAG-DP register map

IR contents	Description	Address	Access	Reference	Notes
DPACC	Read Buffer	0xC	RAZ/WI	*The Read Buffer, RDBUFF* on page 12-59	-
ABORT	DAP Abort Register	0x0	WO[b]	*The Abort Register, ABORT* on page 12-49	-
ABORT	-	0x4 - 0xC	-	-	- b

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

Accessing the JTAG-DP registers

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

IDCODE The IDCODE scan chain has no address field, and accesses the IDCODE register.

DPACC The DPACC scan chain accesses registers at addresses 0x0 to 0xC.

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

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

12.5.2 SW-DP Registers

For most register addresses on the SW-DP, different registers are addressed on read and write accesses. In addition, the CTRLSEL bit in the Select Register changes which register is accessed at address 0b01.

Table 12-14 shows the SW-DP register map.

Table 12-14 SW-DP register map

Address	CTRLSEL[a]	Description	Access[b]	Reference
b00	X	ID Code Register	R	*The Identification Code Register, IDCODE* on page 12-52
		Abort Register	W	*The Abort Register, ABORT*
b01	b0	DP Control/Status Register	R/W	*The Control/Status Register, CTRL/STAT* on page 12-53
	b1	Wire Control Register	R/W	*The Wire Control Register, WCR (SW-DP only)* on page 12-60
b10	X	Read Resend Register	R	*The Read Resend Register, RESEND (SW-DP only)* on page 12-62
		Select Register	W	*The AP Select Register, SELECT* on page 12-57
b11	X	Read Buffer	R	*The Read Buffer, RDBUFF* on page 12-59
		-	W	-

a. CTRLSEL bit in the SELECT register, see *The AP Select Register, SELECT* on page 12-57.
b. Entries in the Access column refer to whether the SWD protocol makes a read or a write access to the given address.

12.5.3 Debug Port (DP) register descriptions

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

The Abort Register, ABORT

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

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

SW-DP It is at address 0b00 on write operations when the DPnAP bit =1, see *Key to illustrations of operations* on page 12-22. Access to the Abort Register is not affected by the value of the CTRLSEL bit in the Select Register.

It is:

- A write-only register.
- Always accessible, and returns an OK response if a valid transaction is received. Abort Register accesses always complete on the first attempt.

Figure 12-20 shows the register bit assignments.

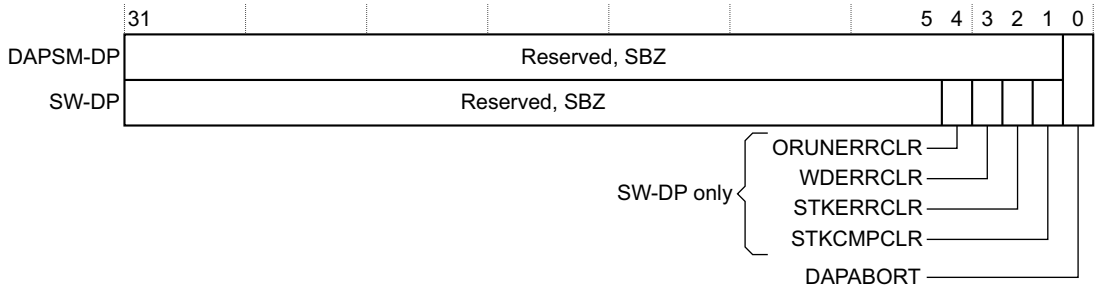


Figure 12-20 Abort Register bit assignments

Table 12-15 lists the bit functions of the Abort Register.

Table 12-15 Abort Register bit assignments

Bits	Function	Description
[31:5]	-	Reserved, SBZ.
[4][a]	ORUNERRCLR[a]	Write b1 to this bit to clear the STICKYORUN overrun error flag[b].
[3][a]	WDERRCLR[a]	Write b1 to this bit to clear the WDATAERR write data error flag[b].
[2][a]	STKERRCLR[a]	Write b1 to this bit to clear the STICKYERR sticky error flag[b].
[1][a]	STKCMPCLR[a]	Write b1 to this bit to clear the STICKYCMP sticky compare flag[b].
[0]	DAPABORT	Write b1 to this bit to generate a DAP abort. This aborts the current AP transaction. This should only be done if the debugger has received WAIT responses over an extended period.

a. Implemented on SW-DP only. On a JTAG-DP this bit is Reserved, SBZ.
b. In the Control/Status register.

DP Aborts

Writing b1 to bit [0] of the Abort Register generates a DP abort, causing the current AP transaction to abort. This also terminates the Transaction Counter, if it was active.

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

You should use this function only in extreme cases, where debug host software has observed stalled target hardware for an extended period. Stalled target hardware is indicated by WAIT responses.

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

Caution

On a JTAG-DP, for the Abort Register:

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

Clearing error and sticky compare flags, SW-DP only

When a debugger, connected to a SW-DP, checks the Control/Status register and finds that an error flag is set, or that the sticky compare flag is set, it must write to the Abort register to clear the error or sticky compare flag. Table 12-15 on page 12-50 listed the flags that might be set in the Control/Status register, and shows which bit of the Abort register is used to clear each of the flags. You can use a single write of the Abort register to clear multiple flags, if this is necessary.

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

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

The Identification Code Register, IDCODE

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

JTAG-DP It is accessed using its own scan chain.

SW-DP It is at address 0b00 on read operations when the DPnAP bit =1. Access to the Identification Code Register is not affected by the value of the CTRLSEL bit in the Select Register.

It is:

- a read-only register
- always accessible.

Figure 12-21 shows the register bit assignments.

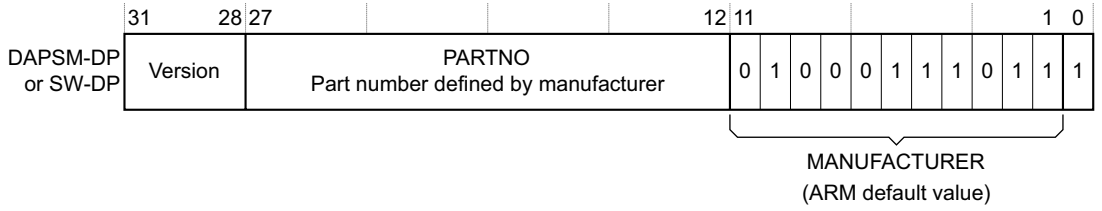


Figure 12-21 Identification Code Register bit assignments

Table 12-16 lists the bit functions of the Identification Code Register.

Table 12-16 Identification Code Register bit assignments

Bits	Function	Description
[31:28]	Version	Version code. The meaning of this field is Implementation-defined.

Table 12-16 Identification Code Register bit assignments

Bits	Function	Description
[27:12]	PARTNO	Part Number for the DP. This value is provided by the designer of the Debug Port and *must not* be changed. Current ARM-designed DPs have the following PARTNO values: **JTAG-DP** `0xBA00` **SW-DP** `0xBA10`
[11:1]	MANUFACTURER	JEDEC Manufacturer ID, an 11-bit JEDEC code that identifies the manufacturer of the device. See *The JEDEC Manufacturer ID*. The ARM default value for this field, shown in Figure 12-21 on page 12-52, is `0x23B`. Designers can change the value of this field. If the DAP is also used for boundary scan then this field *must* be set to the JEDEC Manufacturer ID assigned to the implementor.
[0]	-	Always 0b1.

The JEDEC Manufacturer ID

This code is also described as the JEP-106 manufacturer identification code, and can be subdivided into two fields, as shown in Table 12-17.

Table 12-17 JEDEC JEP-106 manufacturer ID code, with ARM Limited values

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

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

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

The Control/Status Register, CTRL/STAT

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

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

SW-DP It is at address 0b01 on read and write operations when the DPnAP bit =1 and the CTRLSEL bit in the Select Register is set to b0. For information about the CTRLSEL bit see *The AP Select Register, SELECT* on page 12-57.

It is a read-write register, in which some bits have different access rights. It is Implementation-defined whether some fields in the register are supported, and Table 12-18 shows which fields are required in all implementations.

Figure 12-22 shows the register bit assignments.

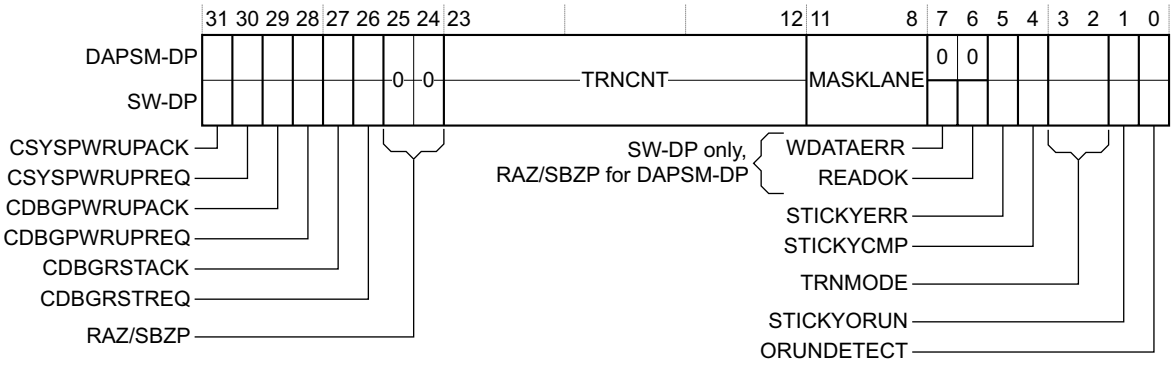


Figure 12-22 Control/Status Register bit assignments

Table 12-18 lists the bit functions of the Control/Status Register.

Table 12-18 Control/Status Register bit assignments

Bits	Access	Function	Description	Required?
[31]	RO	CSYSPWRUPACK	System power-up acknowledge.	No
[30]	R/W	CSYSPWRUPREQ	System power-up request. After a reset this bit is Low (0).	No
[29]	RO	CDBGPWRUPACK	Debug power-up acknowledge.	No
[28]	R/W	CDBGPWRUPREQ	Debug power-up request. After a reset this bit is Low (0).	No
[27]	RO	CDBGRSTACK	Debug reset acknowledge.	Yes
[26]	R/W	CDBGRSTREQ	Debug reset request. After a reset this bit is Low (0).	Yes
[25:24]	-	-	Reserved, RAZ/SBZP	
[21:12]	R/W	TRNCNT	Transaction counter. After a reset the value of this field is Unpredictable.	Yes

Table 12-18 Control/Status Register bit assignments (continued)

Bits	Access	Function	Description	Required?
[11:8]	R/W	MASKLANE	Indicates the bytes to be masked in pushed compare and pushed verify operations. See *MASKLANE and the bit masking of the pushed compare and pushed verify operations* on page 12-56. After a reset the value of this field is Unpredictable.	Yes
[7]	RO[a]	WDATAERR[a]	This bit is set to 1 if a Write Data Error occurs. It is set if: • there is a a parity or framing error on the data phase of a write • a write that has been accepted by the DP is then discarded without being submitted to the AP. This bit can only be cleared by writing b1 to the WDERRCLR field of the Abort Register, see *The Abort Register, ABORT* on page 12-49. After a reset this bit is Low (0).	Yes[a]
[6]	RO[a]	READOK[a]	This bit is set to 1 if the response to a previous AP or RDBUFF was OK. It is cleared to 0 if the response was not OK. This flag always indicates the response to the last AP read access. After a reset this bit is Low (0).	Yes[a]
[5]	RO[b]	STICKYERR	This bit is set to 1 if an error is returned by an AP transaction. To clear this bit: **On a JTAG-DP** Write b1 to this bit of this register. **On a SW-DP** Write b1 to the STKERRCLR field of the Abort Register, see *The Abort Register, ABORT* on page 12-49. After a reset this bit is Low (0).	Yes
[4]	RO[b]	STICKYCMP	This bit is set to 1 when a match occurs on a pushed compare or a pushed verify operation. To clear this bit: **On a JTAG-DP** Write b1 to this bit of this register. **On a SW-DP** Write b1 to the STKCMPCLR field of the Abort Register, see *The Abort Register, ABORT* on page 12-49. After a reset this bit is Low (0).	Yes
[3:2]	R/W	TRNMODE	This field sets the transfer mode for AP operations, see *Transfer mode (TRNMODE), bits [3:2]* on page 12-57. After a reset the value of this field is Unpredictable.	Yes

Table 12-18 Control/Status Register bit assignments (continued)

Bits	Access	Function	Description	Required?
[1]	RO[b]	STICKYORUN	If overrun detection is enabled (see bit [0] of this register), this bit is set to 1 when an overrun occurs. To clear this bit: **On a JTAG-DP** Write b1 to this bit of this register. **On a SW-DP** Write b1 to the ORUNERRCLR field of the Abort Register, see *The Abort Register, ABORT* on page 12-49. After a reset this bit is Low (0).	Yes
[0]	R/W	ORUNDETECT	This bit is set to b1 to enable overrun detection. After a reset this bit is Low (0).	Yes

a. Implemented on SW-DP only. On a JTAG-DP this bit is Reserved, RAZ/SBZP.
b. RO on SW-DP. On a JTAG-DP, this bit can be read normally, and writing b1 to this bit clears the bit to b0.

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

The MASKLANE field, bits [11:8] of the CTRL/STAT Register, is only relevant if the Transfer Mode is set to pushed verify or pushed compare operation, see *Transfer mode (TRNMODE), bits [3:2]* on page 12-57.

In the pushed operations, the word supplied in an AP write transaction is compared with the current value of the target AP address. The MASKLANE field lets you specify that the comparison is made using only certain bytes of the values. Each bit of the MASKLANE field corresponds to one byte of the AP values. Therefore, each bit is said to control one byte lane of the compare operation.

Table 12-19 shows how the bits of MASKLANE control the comparison masking.

Table 12-19 Control of pushed operation comparisons by MASKLANE

MASKLANE[a]	Meaning	Mask used for comparisons[b]
b1XXX	Include byte lane 3 in comparisons.	`0xFF------`
bX1XX	Include byte lane 2 in comparisons.	`0x--FF----`
bXX1X	Include byte lane 1 in comparisons.	`0x----FF--`
bXXX1	Include byte lane 0 in comparisons.	`0x------FF`

a. Bits [11:8] of the CTRL/STAT Register.
b. Bytes of the mask shown as -- are determined by the other bits of MASKLANE.

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

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

This field sets the transfer mode for AP operations. Table 12-20 lists the allowed values of this field, and their meanings.

Table 12-20 Transfer Mode, TRNMODE, bit definitions

TRNMODE[a]	AP Transfer mode
b00	Normal operation.
b01	Pushed verify operation.
b10	Pushed compare operation.
b11	Reserved.

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

In normal operation, AP transactions are passed to the AP for processing.

In pushed verify and pushed compare operations, the DP compares the value supplied in the AP transaction with the value held in the target AP address.

The AP Select Register, SELECT

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

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

SW-DP It is at address 0b10 on write operations when the DPnAP bit =1, and is a write-only register. Access to the AP Select Register is not affected by the value of the CTRLSEL bit.

Figure 12-23 on page 12-58 shows the register bit assignments.

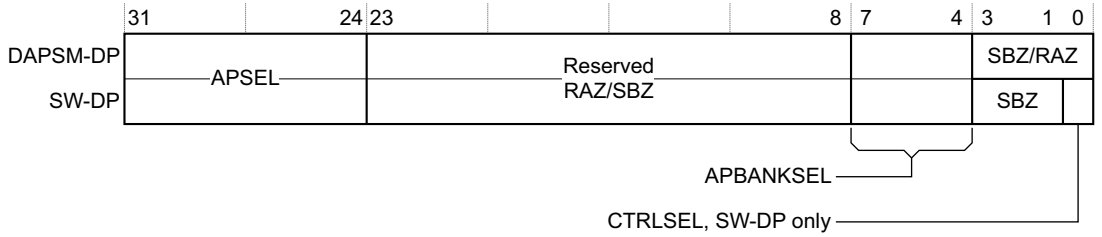


Figure 12-23 Bit assignments for the AP Select Register, SELECT

Table 12-21 lists the bit functions of the Select Register.

Table 12-21 Bit assignments for the AP Select Register, SELECT

Bits	Function	Description
[31:24]	APSEL	Selects the current AP. The reset value of this field is Unpredictable.[a]
[23:8]	-	Reserved. SBZ/RAZ[a].
[7:4]	APBANKSEL	Selects the active four-word register window on the current AP. The reset value of this field is Unpredictable.[a]
[3:1]	-	Reserved. SBZ/RAZ[a].
[0][b]	CTRLSEL[b]	SW-DP Debug Port address bank select, see *CTRLSEL, SW-DP only* on page 12-59. After a reset this field is b0. However the register is WO and you cannot read this value.

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

If APSEL is set to a non-existent AP then all AP transactions return zero on reads and are ignored on writes.

Note

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

CTRLSEL, SW-DP only

The CTRLSEL field, bit [0], controls which DP register is selected at address b01 on a SW-DP. Table 12-22 shows the meaning of the different values of CTRLSEL.

Table 12-22 CTRLSEL field bit definitions

CTRLSEL[a]	DP Register at address b01
0	CTRL/STAT, see *The Control/Status Register, CTRL/STAT* on page 12-53.
1	WCR, see *The Wire Control Register, WCR (SW-DP only)* on page 12-60.

a. Bit [0] of the SELECT Register.

The Read Buffer, RDBUFF

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

JTAG-DP It is at address `0xC` when the Instruction Register (IR) contains DPACC, and is a Read-as-zero, Writes ignored (RAZ/WI) register.

SW-DP It is at address 0b11 on read operations when the DPnAP bit =1, and is a read-only register. Access to the Read Buffer is not affected by the value of the CTRLSEL bit in the SELECT Register.

Read Buffer implementation and use on a JTAG-DP

On a JTAG-DP, the Read Buffer always reads as zero, and writes to the Read Buffer address are ignored.

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

Read Buffer implementation and use on a SW-DP

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

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

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

- A second AP register read. You can read the Control/Status Register, CSW, if you want to ensure that this second read has no side effects.
- A read of the DP Read Buffer.

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

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

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

It is a read/write register at address 0b01 on read and write operations when the CTRLSEL bit in the Select Register is set to b1. For information about the CTRLSEL bit see *The AP Select Register, SELECT* on page 12-57.

Note

When the CTRLSEL bit is set to b1, to enable access to the WCR, the DP Control/Status Register is not accessible.

Many features of the Wire Control Register are Implementation-defined.

Figure 12-24 shows the register bit assignments.

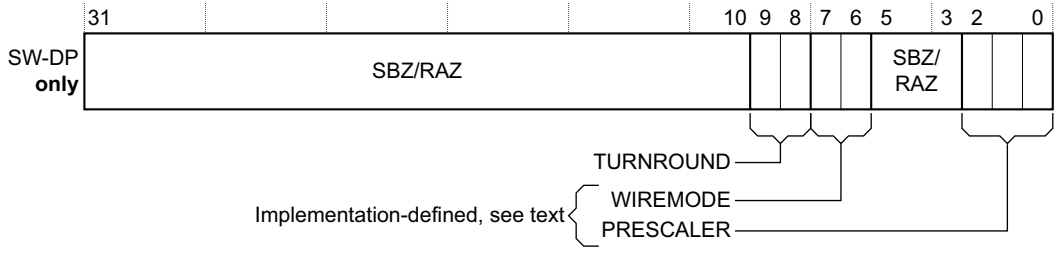


Figure 12-24 Bit assignments for the Wire Control Register (SW-DP only)

Table 12-23 lists the bit functions of the Wire Control Register.

Table 12-23 Bit assignments for the Wire Control Register (SW-DP only)

Bits	Function	Description
[31:10]	-	Reserved. SBZ/RAZ.
[9:8]	TURNROUND	Turnaround tri-state period, see *Turnaround tri-state period, TURNROUND, bits [9:8]*. After a reset this field is b00.
[7:6]	WIREMODE	Identifies the operating mode for the wire connection to the DP, see *Wire operating mode, WIREMODE, bits [7:6]*. After a reset this field is b01.
[5:3]	-	Reserved. SBZ/RAZ.
[2:0]	Reserved	-

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

This field defines the turnaround tri-state period. This turnaround period allows for pad delays when using a high sample clock frequency. Table 12-24 lists the allowed values of this field, and their meanings.

Table 12-24 Turnaround tri-state period field, TURNROUND, bit definitions

TURNROUND[a]	Turnaround tri-state period
b00	1 sample period.
b01	2 sample periods.
b10	3 sample periods.
b11	4 sample periods.

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

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

This field identifies SW-DP as operating in Synchronous mode only.

This field is required, and Table 12-25 lists the allowed values of the field, and their meanings.

Table 12-25 Wire operating mode, WIREMODE, bit definitions

WIREMODE[a]	Wire operating mode
b00	Reserved
b01	Synchronous (no oversampling).
b1X	Reserved.

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

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

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

It is a 32-bit read-only register at address 0b10 on read operations. Access to the Read Resend Register is not affected by the value of the CTRLSEL bit in the SELECT Register.

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

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

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

Chapter 13
Trace Port Interface Unit

This chapter describes the *Trace Port Interface Unit* (TPIU). It contains the following sections:

- *About the Trace Port Interface Unit* on page 13-2
- *TPIU registers* on page 13-8.

13.1 About the Trace Port Interface Unit

The *Trace Port Interface Unit* (TPIU) acts as a bridge between the on-chip trace data from the *Embedded Trace Macrocell* (ETM) and the *Instrumentation Trace Macrocell* (ITM), with separate IDs, to a data stream, encapsulating IDs where required, that is then captured by a *Trace Port Analyzer* (TPA).

The TPIU is specially designed for low-cost debug. It is a special version of the CoreSight TPIU, and it can be replaced by CoreSight components if system requirements demand the additional features of the CoreSight TPIU.

There are two configurations of the TPIU:

- A configuration that supports ITM debug trace.
- A configuration that supports both ITM and ETM debug trace.

Note

If your Cortex-M3 system uses the optional ETM component, you must use the TPIU configuration that supports both ITM and ETM debug trace. For a full description of the ETM, see Chapter 15 *Embedded Trace Macrocell*.

13.1.1 TPIU block diagrams

Figure 13-1 on page 13-3 and Figure 13-2 on page 13-4 show the component layout of the TPIU for both configurations.

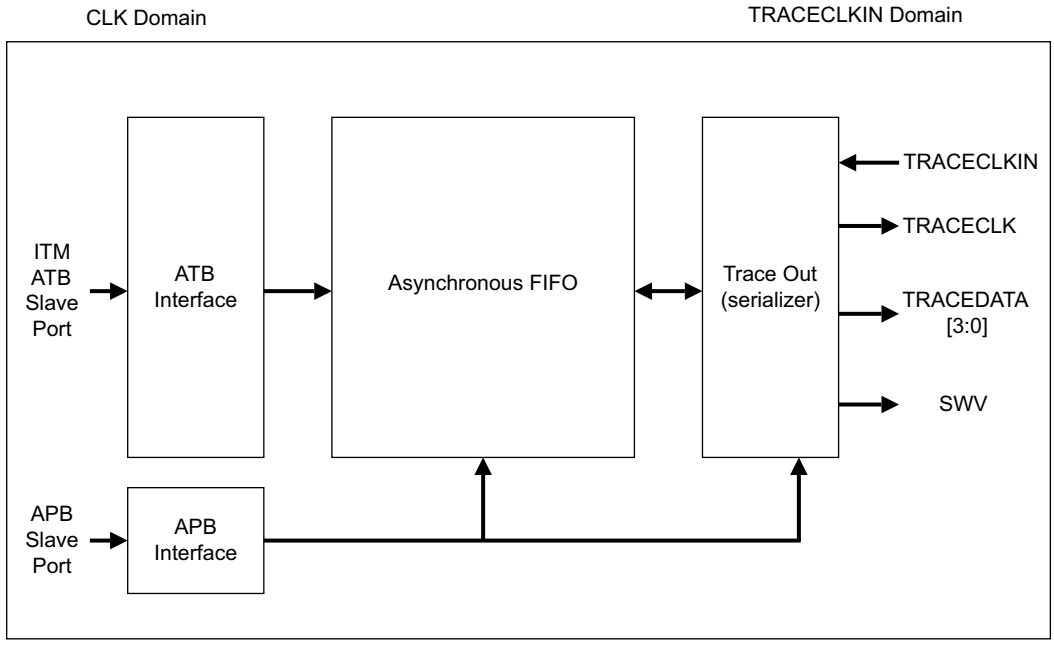


Figure 13-1 Block diagram of the TPIU (non-ETM version)

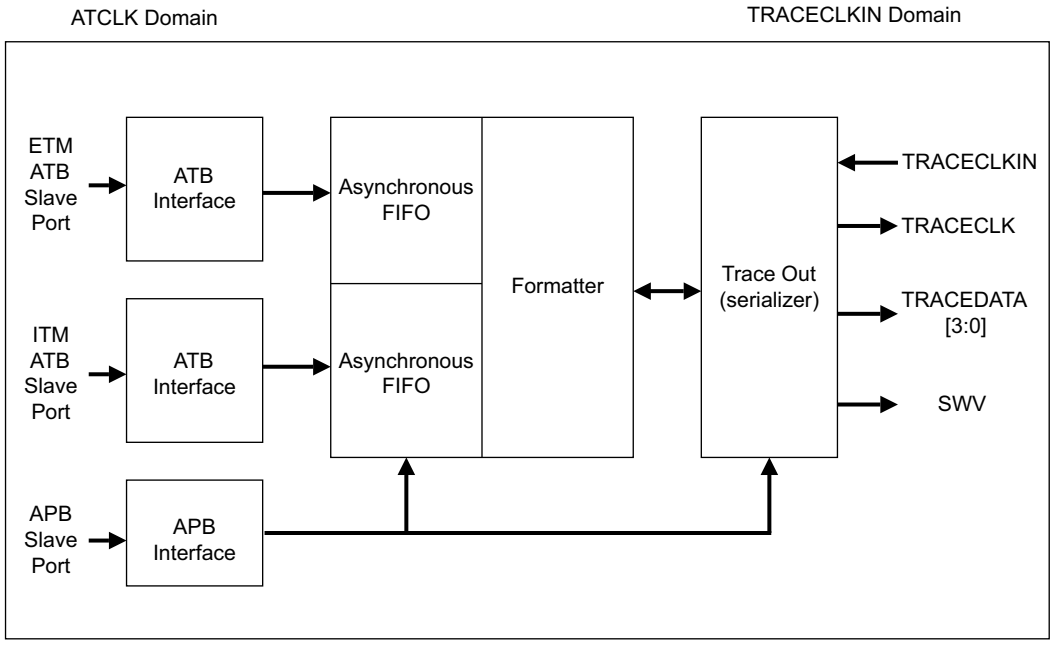


Figure 13-2 Block diagram of the TPIU (ETM version)

13.1.2 TPIU components

A description of the main components of the TPIU is given in the following sections. These include:

- *Asynchronous FIFO*
- *Formatter* on page 13-5
- *Trace out* on page 13-5
- *ATB interface* on page 13-5
- *APB interface* on page 13-5.

Asynchronous FIFO

The asynchronous FIFO enables trace data to be driven out at a speed that is not dependent on the speed of the core clock.

Formatter

The formatter inserts source ID signals into the data packet stream so that trace data can be re-associated with its trace source. The formatter is only present in the version of the TPIU for use with the ETM.

Trace out

The trace out block serializes formatted data before it goes off-chip.

ATB interface

The TPIU accepts trace data from a trace source, either direct from a trace source (ETM or ITM) or using a Trace Funnel. For more information, see *ATB interface*.

APB interface

The APB interface is the programming interface for the TPIU. For more information, see *APB interface*.

13.1.3 TPIU inputs and outputs

This section describes the TPIU inputs and outputs. It contains the following:

- *Trace out port*
- *ATB interface*
- *Miscellaneous configuration inputs* on page 13-6.

Trace out port

Table 13-1 describes the Trace Out Port signals.

Table 13-1 Trace Out Port signals

Name	Type	Description
TRACECLKIN	Input	Decoupled clock from ATB to enable easy control of the trace port speed. Typically this is derived from a controllable clock source on chip but could be driven by an external clock generator if a high speed pin is used. Data changes on the rising edge only.
TRESETn	Input	This is a reset signal for the **TRACECLKIN** domain. This signal is typically driven from Power on Reset, and should be synchronized to **TRACECLKIN**.

Table 13-1 Trace Out Port signals

Name	Type	Description
TRACECLK	Output	**TRACEDATA** changes on both edges of **TRACECLK**.
TRACEDATA[3:0]	Output	Output data for clocked modes.
SWV	Output	Output data for asynchronous modes.

ATB interface

There is one or two ATB interfaces depending on the TPIU configuration. Table 13-2 describes the ATB Port signals. The signals for port 2 are not used when the TPIU is configured with a single ATB interface.

Table 13-2 ATB Port signals

Name	Type	Description
CLK	Input	Trace bus and APB interface clock.
nRESET	Input	Reset for the **CLK** domain (ATB/APB interface).
CLKEN	Input	Clock enable for **CLK** domain.
ATVALID1S	Input	Data from trace source 1 is valid in this cycle.
ATREADY1S	Output	If this signal is asserted (**ATVALID** high), then the data was accepted this cycle from trace source 1.
ATDATA1S[7:0]	Input	Trace data input from source 1.
ATID1S[6:0]	Input	Trace source ID for source 1. This must not change dynamically.
ATVALID2S	Input	Data from trace source 2 is valid in this cycle.
ATREADY2	Output	If this signal is asserted (**ATVALID** high), then the data was accepted this cycle from trace source 2.
ATDATA2S[7:0]	Input	Trace data input from source 2.
ATID2S[6:0]	Input	Trace source ID for source 2. This must not change dynamically.

Miscellaneous configuration inputs

Table 13-3 on page 13-7 describes the miscellaneous configuration inputs.

Table 13-3 Miscellaneous configuration inputs

Name	Type	Description
MAXPORTSIZE [1:0]	Input	Defines the maximum number of pins available for synchronous trace output.
SyncReq	Input	Global trace synchronization trigger. Used to insert synchronization packets into the formatted data stream. Not used for non-ETM configurations.

13.2 TPIU registers

This section describes the TPIU registers. It contains the following:

- *Summary of the TPIU registers*
- *Description of the TPIU registers*.

13.2.1 Summary of the TPIU registers

Table 13-4 provides a summary of the TPIU registers.

Table 13-4 TPIU registers

Name of register	Type	Address	Reset value	Page
Supported Port Sizes Register	Read-only	0xE0040000	0bxx0x	page 13-8
Current Port Size Register	Read/write	0xE0040004	0x01	page 13-9
Current Output Speed Divisors Register	Read/write	0xE0040010	0x0000	page 13-9
Selected Pin Protocol Register	Read/write	0xE00400F0	0x01	page 13-10
Formatter and Flush Status Register	Read/write	0xE0040300	0x08	page 13-10
Formatter and Flush Control Register	Read-only	0xE0040304	0x00 or 0x102	page 13-11
Formatter Synchronization Counter Register	Read-only	0xE0040308	0x00	page 13-11
Integration Register: ITATBCTR2	Read-only	0xE0040EF0	0x0	page 13-12
Integration Register: ITATBCTR0	Read-only	0xE0040EF8	0x0	page 13-12

13.2.2 Description of the TPIU registers

A description of the TPIU registers follows.

Supported Port Sizes Register

This register is read/write. Each bit location represents a single port size that is supported on the device, that is, 4, 2 or 1 in bit locations [3:0]. If the bit is set then that port size is allowable. By default the RTL is designed to support all port sizes, set to 0x0000000B. This register is further constrained by the input tie-off **MAXPORTSIZE**. The external tie-off, **MAXPORTSIZE**, must be set during finalization of the ASIC to reflect the actual number of TRACEDATA signals being wired to physical pins. This is to ensure that tools do not attempt to select a port width that cannot be captured by an attached TPA. The value on **MAXPORTSIZE** causes bits within the Supported Port Size register that represent wider widths to be clear, that is, unsupported.

Figure 13-3 shows the bit assignments.

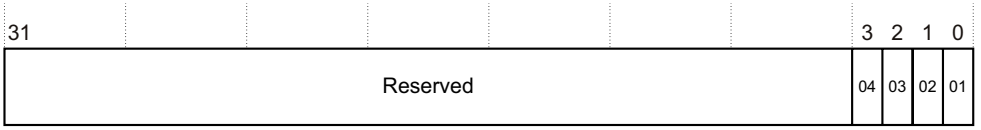


Figure 13-3 Supported Port Size Register bit assignments

Current Port Size Register

This register is read/write. The Current Port Size Register has the same format as the Supported Port Sizes register but only one bit is set, and all others must be zero. Writing values with more than one bit set or setting a bit that is not indicated as supported is not supported and causes unpredictable behavior.

It is more convenient to use the same format as the Supported Port Sizes register because it saves on having to decode the sizes later on in the device and also maintains the format from the other register bank for checking for valid assignments.

On reset this defaults to the smallest possible port size, 1 bit, and so reads as `0x00000001`.

Current Output Speed Divisors Register

Use the Current Output Speed Divisors Register to scale the baud rate of the asynchronous output.

Figure 13-4 shows the fields of the Current Output Speed Divisors Register.

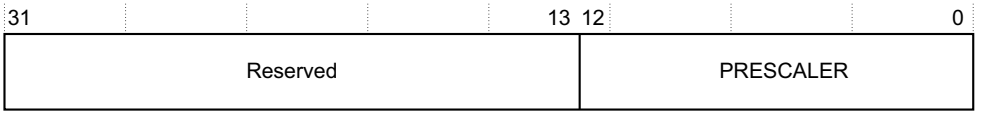


Figure 13-4 Current Output Speed Divisors Register bit assignments

Table 13-5 describes the fields of the Current Output Speed Divisors Register.

Table 13-5 Current Output Speed Divisors Register bit assignments

Field	Name	Definition
[31:13]	-	Reserved. RAZ/SBZP
[12:0]	PRESCALER	Divisor for TRACECLKIN is Prescaler + 1

Selected Pin Protocol Register

Use the Selected Pin Protocol Register to select which protocol to use for trace output.

The register address, access type, and Reset state are:

Address `0xE00400F0`
Access Read/write
Reset state `0x01`

Figure 13-5 shows the fields of the Selected Pin Protocol Register.

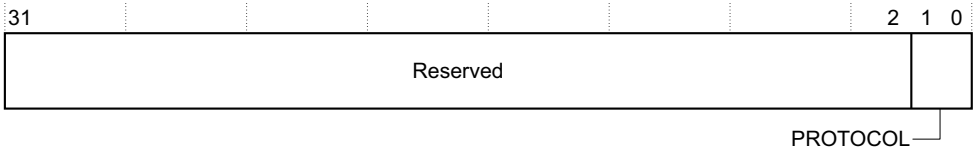


Figure 13-5 Selected Pin Protocol Register bit assignments

Table 13-6 describes the fields of the Selected Pin Protocol Register.

Table 13-6 Selected Pin Protocol Register bit assignments

Field	Name	Definition
[31:2]	-	Reserved.
[1:0]	PROTOCOL	00 - TracePort mode 01 - SerialWire Output (Manchester). This is the reset value. 10 - SerialWire Output (NRZ) 11 - Reserved

Note

If this register is changed while trace data is being output, data corruption occurs.

Formatter and Flush Status Register

Use the Formatter and Flush Status Register to read the status of TPIU formatter.

The register address, access type, and Reset state are:

Address `0xE0040300`
Access Read only
Reset state `0x08`

Figure 13-6 shows the fields of the Formatter and Flush Status Register.

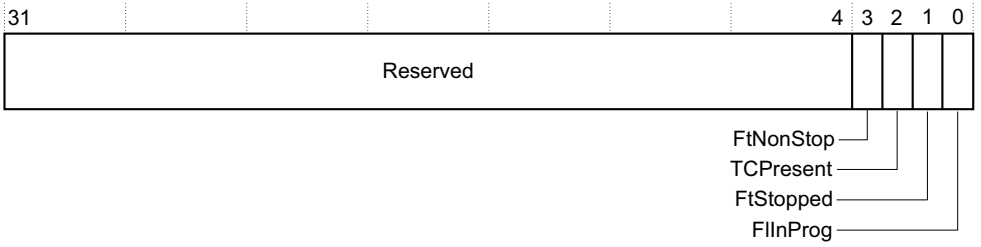


Figure 13-6 Formatter and Flush Status Register bit assignments

Table 13-7 describes the fields of the Formatter and Flush Status Register.

Table 13-7 Formatter and Flush Status Register bit assignments

Field	Name	Definition
[31:4]	-	Reserved
[3]	FtNonStop	Formatter cannot be stopped.
[2]	TCPresent	This bit always reads zero.
[1]	FtStopped	This bit always reads zero.
[0]	FlInProg	This bit always reads zero.

Formatter and Flush Control Register

Use the Formatter and Flush Control Register to read whether the formatter is present or not. This register is read only because dynamic control of the formatter is not possible. If the formatter is present, the register reads `0x102`, indicating formatter enabled and in continuous mode, with triggers indicated when **TRIGIN** is asserted.

The register address, access type, and Reset state are:

Address `0xE0040304`

Access Read only

Reset state `0x00` or `0x102`

Formatter Synchronization Counter Register

The global synchronization trigger is generated by the PC Sampler block. This means that there is no synchronization counter in the TPIU.

The register address, access type, and Reset state are:

Address `0xE0040308`

Access Read only

Reset state `0x00`

Integration Test Registers

Use the Integration Test Registers to perform topology detection of the TPIU with other devices in a Cortex-M3 system. These registers enable direct control of outputs and the ability to read the value of inputs. The processor provides two Integration Test Registers:

- Integration Test Register - ITATBCTR2
- Integration Test Register - ITATBCTR0.

Integration Test Register-ITATBCTR2

The register address, access type, and Reset state are:

Address `0xE0040EF0`

Access Read only

Reset state `0x0`

Figure 13-7 shows the fields of the Integration Test Register bit assignments.

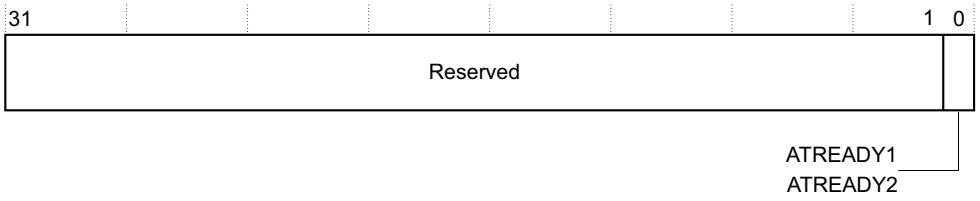


Figure 13-7 Integration Test Register bit assignments

Table 13-8 describes the fields of the Integration Test Register bit assignments.

Table 13-8 Integration Test Register bit assignments

Field	Name	Definition
[31:1]	-	Reserved
[0]	ATREADY1	This bit reads or sets the value of ATREADYS1 and ATREADYS2.

Integration Test Register-ITATBCTR0

The register address, access type, and Reset state are:

Address `0xE0040EF8`

Access Read only

Reset state `0x0`

Figure 13-8 shows the fields of the Integration Test Register bit assignments.

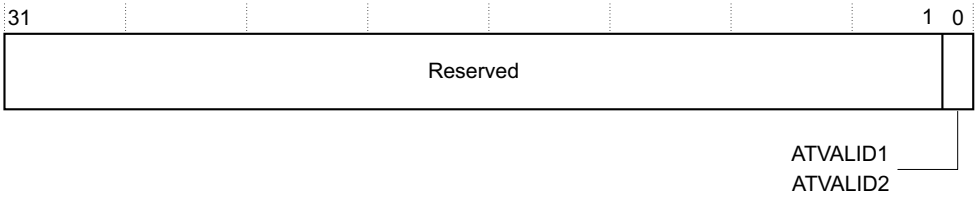


Figure 13-8 Integration Test Register bit assignments

Table 13-9 describes the fields of the Integration Test Register bit assignments.

Table 13-9 Integration Test Register bit assignments

Field	Name	Definition
[31:1]	-	Reserved
[0]	ATVALID1, ATVALID2	This bit reads or sets the value of ATVALIDS1 OR-ed with ATVALIDS2.

Chapter 14
Bus Interface

This chapter describes the processor bus interface. It contains the following sections:

- *About bus interfaces* on page 14-2
- *ICode bus interface* on page 14-3
- *DCode bus interface* on page 14-5
- *System interface* on page 14-6
- *External private peripheral interface* on page 14-8
- *Access alignment* on page 14-9
- *Unaligned accesses that cross regions* on page 14-10
- *Bit-band accesses* on page 14-11
- *Write buffer* on page 14-12.

14.1 About bus interfaces

The processor contains four bus interfaces:

- The ICode memory interface. Instruction fetches from Code memory space (`0x0000000` - `0x1FFFFFFF`) are performed over this 32-bit AHBLite bus. For more information, see *ICode bus interface* on page 14-3.
- The DCode memory interface. Data and debug accesses to Code memory space (`0x0000000` - `0x1FFFFFFF`) are performed over this 32-bit AHBLite bus. For more information, see *DCode bus interface* on page 14-5.
- The System interface. Instruction fetches, and data and debug accesses, to System space (`0x20000000` - `0xDFFFFFFF`, `0xE0100000` - `0xFFFFFFFF`) are performed over this 32-bit AHBLite bus. For more information, see *System interface* on page 14-6.
- The External *Private Peripheral Bus* (PPB). Data and debug accesses to External PPB space (`0xE0040000` - `0xE00FFFFF`) are performed over this 32-bit APB (AMBA v2.0) bus. The *Trace Port Interface Unit* (TPIU) and vendor specific peripherals are on this bus. For more information, see *External private peripheral interface* on page 14-8.

Note

The processor contains an internal Private Peripheral Bus for accesses to the *Nested Vectored Interrupt Controller* (NVIC), *Data Watchpoint and Trigger* (DWT), *Flash Patch and Breakpoint* (FPB), and *Memory Protection Unit* (MPU).

14.2 ICode bus interface

The ICode interface is a 32-bit AHBLite bus interface. Instruction fetches and vector fetches from Code memory space (`0x00000000 - 0x1FFFFFFF`) are performed over this bus.

Only the CM3Core instruction fetch bus can access the ICode interface, enabling optimal code fetch performance. All fetches are word wide. The number of instructions fetched per word depends on the code running and the alignment of the code in memory. This is described in Table 14-1.

Table 14-1 Instruction fetches

32-bit instruction fetch [31:16]	32-bit instruction fetch [15:0]	Description
Thumb[15:0]	Thumb[15:0]	All Thumb instructions are halfword aligned in memory, so two Thumb instructions are fetched at a time. For sequential code, an instruction fetch is performed every second cycle. Instruction fetches can be performed on back-to-back cycles if there is an interrupt or a branch.
Thumb-2[31:16]	Thumb-2[15:0]	If Thumb-2 code is word-aligned in memory, then a complete Thumb-2 instruction is fetched each cycle.
Thumb-2[15:0]	Thumb-2[31:16]	If Thumb-2 code is halfword aligned, then the first 32-bit fetch only returns the first halfword of the Thumb-2 instruction. A second fetch must be performed to fetch the second halfword. This scenario creates a wait cycle (a cycle where CM3Core is not able to execute an instruction) depending on the instruction in play. The additional cycle of latency only occurs for the first halfword aligned Thumb-2 instruction fetch. CM3Core contains a 3-entry fetch buffer, and so the upper halfword of halfword aligned Thumb-2 instructions exist in the fetch buffer for subsequent sequential Thumb-2 instructions.

All ICode instruction fetches are marked as cacheable and bufferable (**HPROTI[3:2]** = 2'b11), and as non-allocate and non-shareable (**MEMATTRI** = 2'b00). These attributes are hard wired. If an MPU is fitted, the MPU region attributes are ignored for the ICode bus.

HPROTI[0] indicates what is being fetched:

- 0 - instruction fetch
- 1 - vector fetch.

All ICode transactions are performed as non-sequentials.

14.2.1 Branch status signal

A branch status signal (**BRCHSTAT**) is exported on the ETM interface that indicates if there are any branches in the pipeline. This can be used, for example, by a prefetcher to prevent prefetching if a branch is about to be fetched. For more information about the branch status signal, see Chapter 16 *Embedded Trace Macrocell Interface*.

14.3 DCode bus interface

The DCode interface is a 32-bit AHBLite bus. Data and debug accesses to Code memory space (`0x00000000 - 0x1FFFFFFF`) are performed over this bus. Core data accesses have a higher priority than debug accesses. This means that debug accesses are waited until core accesses have completed when there are simultaneous core and debug access to this bus.

Control logic in this interface converts unaligned data and debug accesses into two or three (depending on the size and alignment of the unaligned access) aligned accesses. This stalls any subsequent data or debug access until the unaligned access has completed.

See *Access alignment* on page 14-9 for a description of unaligned accesses.

14.3.1 Exclusives

The DCode bus supports exclusive accesses. This is carried out using two sideband signals, **EXREQD** and **EXRESPD**. For more information, see *DCode interface* on page A-7.

14.3.2 Memory attributes

The processor exports memory attributes on the DCode bus by using a sideband bus called **MEMATTRD**. For more information, see *Memory attributes* on page 14-13.

14.4 System interface

The system interface is a 32-bit AHBLite bus. Instruction and vector fetches, and data and debug accesses to the System memory space (`0x20000000 - 0xDFFFFFFF`, `0xE0100000 - 0xFFFFFFFF`) are performed over this bus.

For simultaneous accesses to this bus, the arbitration order (in decreasing priority) is:

- data accesses
- instruction and vector fetches
- debug.

The System bus interface contains control logic to handle unaligned accesses, FPB remapped accesses, bit-band accesses, and pipelined instruction fetches.

14.4.1 Unaligned accesses

Unaligned data and debug accesses are converted into two or three (depending on the size and alignment of the unaligned access) aligned accesses. This stalls any subsequent accesses until the unaligned access has completed. For a description of unaligned accesses, see *Access alignment* on page 14-9.

14.4.2 Bit-band accesses

Accesses to the bit-band alias region are converted into accesses to the bit-band region. Bit-band writes take two cycles (they are converted into read-modify-write operations), and so bit-band write accesses stall any subsequent accesses until the bit-band access has completed. For a description of bit-band accesses, see *Bit-band accesses* on page 14-11.

14.4.3 Flash Patch remapping

Accesses to the Code memory space that are remapped to System memory space incur a cycle penalty to be remapped. This stalls any subsequent accesses until the Flash Patch access has completed. See *Flash Patch and Breakpoint* on page 11-6 for a description of Flash Patch.

14.4.4 Exclusives

The System bus supports exclusive accesses. This is carried out using two sideband signals, **EXREQS** and **EXRESPS**. For more information, see *System bus interface* on page A-8.

14.4.5 Memory attributes

The processor exports memory attributes on the System bus by using a sideband bus called **MEMATTRS**. For more information, see *Memory attributes* on page 14-13.

14.4.6 Pipelined instruction fetches

To provide a clean timing interface on the System bus, instruction and vector fetch requests to this bus are registered. This results in an additional cycle of latency because instructions fetched from the System bus take two cycles. This also means that back-to-back instruction fetches from the System bus are not possible.

Note

Instruction fetch requests to the ICode bus are not registered. Performance critical code should be run from the ICode interface.

14.5 External private peripheral interface

The external private peripheral interface is an *Advanced Peripheral Bus* (APB) (AMBA v2.0) bus. Data and debug accesses to the External Peripheral memory space (`0xE0040000 - 0xE00FFFFF`) are performed over this bus. Wait states are not supported on this bus. The TPIU and any vendor specific components populate this bus. Core data accesses have higher priority than debug accesses, so debug accesses are waited until core accesses have completed when there are simultaneous core and debug access to this bus. Only the address bits necessary to decode the External PPB space are supported on this interface. These address bits are bits [19:2] of **PADDR**.

PADDR31 is driven as a sideband signal on this bus. When the signal is HIGH, it indicates that the AHB-AP debug is the requesting master. When the signal is LOW, it indicates that the core is the requesting master.

Unaligned accesses to this bus are architecturally unpredictable and not supported. The processor drives out the original **HADDR[1:0]** request from the core and does not convert the request into multiple aligned accesses.

14.6 Access alignment

The Cortex-M3 processor supports unaligned data accesses using the ARMv6 model. The DCode and System bus interfaces contain logic that converts unaligned accesses to aligned accesses.

The unaligned data accesses are described in Table 14-2. The table shows the unaligned access in the first column, with the remaining columns showing what the access is converted into. Depending on the size and alignment of the unaligned access, it is converted into two or three aligned accesses.

Table 14-2 Bus mapper unaligned accesses

Unaligned access		Aligned access					
		Cycle 1		Cycle 2		Cycle 3	
Size	ADDR [1:0]	HSIZE	HADDR [1:0]	HSIZE	HADDR[1:0]	HSIZE	HADDR[1:0]
Halfword	00	Halfword	00	-	-	-	-
Halfword	01	Byte	01	Byte	10	-	-
Halfword	10	Halfword	10	-	-	-	-
Halfword	11	Byte	11	Byte	{(Addr+4)[31:2],2b00}	-	-
Word	00	Word	00	-	-	-	-
Word	01	Byte	01	Halfword	10	Byte	{(Addr+4)[31:2],2b00}
Word	10	Halfword	10	Halfword	{(Addr+4)[31:2],2b00}	-	-
Word	11	Byte	11	Halfword	{(Addr+4)[31:2],2b00}	Byte	{(Addr+4)[31:2],2b10}

Note

Unaligned accesses that cross into the bit-band alias region are not treated as bit-band requests, and the access is not remapped to the bit-band region. Instead, they are treated as a halfword or byte access to the bit-band alias region.

14.7 Unaligned accesses that cross regions

The CM3Core supports ARMv6 unaligned accesses. All accesses are performed as single, unaligned accesses by the CM3Core, and are converted into two or more aligned accesses by the DCode and System bus interfaces.

Note

All Cortex-M3 external accesses are aligned.

Unaligned support is only available for load/store singles (LDR, STR). Load/store double already supports word aligned accesses, but does not permit other unaligned accesses, and generates a fault if this is attempted.

Unaligned accesses that cross memory map boundaries are architecturally unpredictable. The processor behavior is boundary dependent, as follows:

- DCode accesses wrap within the region. For example, an unaligned halfword access to the last byte of Code space (`0x1FFFFFFF`) is converted by the DCode interface into a byte access to `0x1FFFFFFF` followed by a byte access to `0x00000000`.
- System accesses that cross into PPB space do not wrap within System space. For example, an unaligned halfword access to the last byte of System space (`0xDFFFFFFF`) is converted by the System interface into a byte access to `0xDFFFFFFF` followed by a byte access to `0xE0000000`. `0xE0000000` is not a valid address on the System bus.
- System accesses that cross into Code space do not wrap within System space. For example, an unaligned halfword access to the last byte of System space (`0xFFFFFFFF`) is converted by the System interface into a byte access to `0xFFFFFFFF` followed by a byte access to `0x00000000`. `0x00000000` is not a valid address on the System bus.
- Unaligned accesses are not supported to PPB space, and so there are no boundary crossing cases for PPB accesses.

Unaligned accesses that cross into the bit-band alias regions are also architecturally unpredictable. The processor performs the access to the bit-band alias address, but this does not result in a bit-band operation. For example, an unaligned halfword access to `0x21FFFFFF` is performed as a byte access to `0x21FFFFFF` followed by a byte access to `0x22000000` (the first byte of the bit-band alias).

Unaligned loads that match against a literal comparator in the FPB are not remapped. FPB only remaps aligned addresses.

14.8 Bit-band accesses

The System bus interface contains logic that controls bit-band accesses as follows:

- It remaps bit-band alias addresses to the bit-band region.
- For reads, it extracts the requested bit from the read byte, and returns this in the LSB of the read data returned to the core.
- For writes, it converts the write to an atomic read-modify-write operation.

For more information about bit-banding, see *Bit-banding* on page 4-5.

Note

- The CM3Core does not stall during bit-band operations unless it attempts to access the System bus while the bit-band operation is being carried out.
- Big endian accesses to the bit-band alias region must be byte-sized. Otherwise, the accesses are unpredictable.

14.9 Write buffer

To prevent bus wait cycles from stalling the Cortex-M3 processor during data stores, buffered stores to the DCode and System buses go through a one-entry write buffer. If the write buffer is full, subsequent accesses to the bus stall until the write buffer has drained. The write buffer is only used if the bus waits the data phase of the buffered store, otherwise the transaction completes on the bus.

DMB and DSB instructions wait for the write buffer to drain before completing. If an interrupt comes in while DMB/DSB is waiting for the write buffer to drain, the opcode after the DMB/DSB is returned to on the completion of the interrupt. This is because interrupt processing is a memory barrier operation.

14.10 Memory attributes

The processor exports memory attributes on the DCode and System buses by the addition of a sideband bus, MEMATTR.

Table 14-3 shows the relationship between **MEMATTR[0]** and **HPROT[3:2]**.

Table 14-3 Memory attributes

MEMATTR[0]	HPROT[3]	HPROT[2]	Description
0	0	0	Strongly ordered
0	0	1	Device
0	1	0	L1 cacheable, L2 not cacheable
1	0	0	Invalid
1	0	1	Invalid
1	1	0	Cache WT, allocate on read
0	1	1	Cache WB, allocate on read and write
1	1	1	Cache WB, allocate on read

Chapter 15
Embedded Trace Macrocell

This chapter describes the *Embedded Trace Macrocell* (ETM). It contains the following sections:

- *About the ETM* on page 15-2
- *Data tracing* on page 15-6
- *ETM Resources* on page 15-7
- *Trace output* on page 15-9
- *ETM architecture* on page 15-10
- *ETM programmer's model* on page 15-14.

15.1 About the ETM

The ETM is an optional debug component that enables reconstruction of program execution. The ETM is designed to be a high-speed, low-power debug tool that only supports instruction trace. This ensures that area is minimized, and that gate count is reduced.

15.1.1 ETM block diagram

Figure 15-1 on page 15-3 shows a block diagram of the ETM, and shows how the ETM interfaces to the *Trace Port Interface Unit* (TPIU) block.

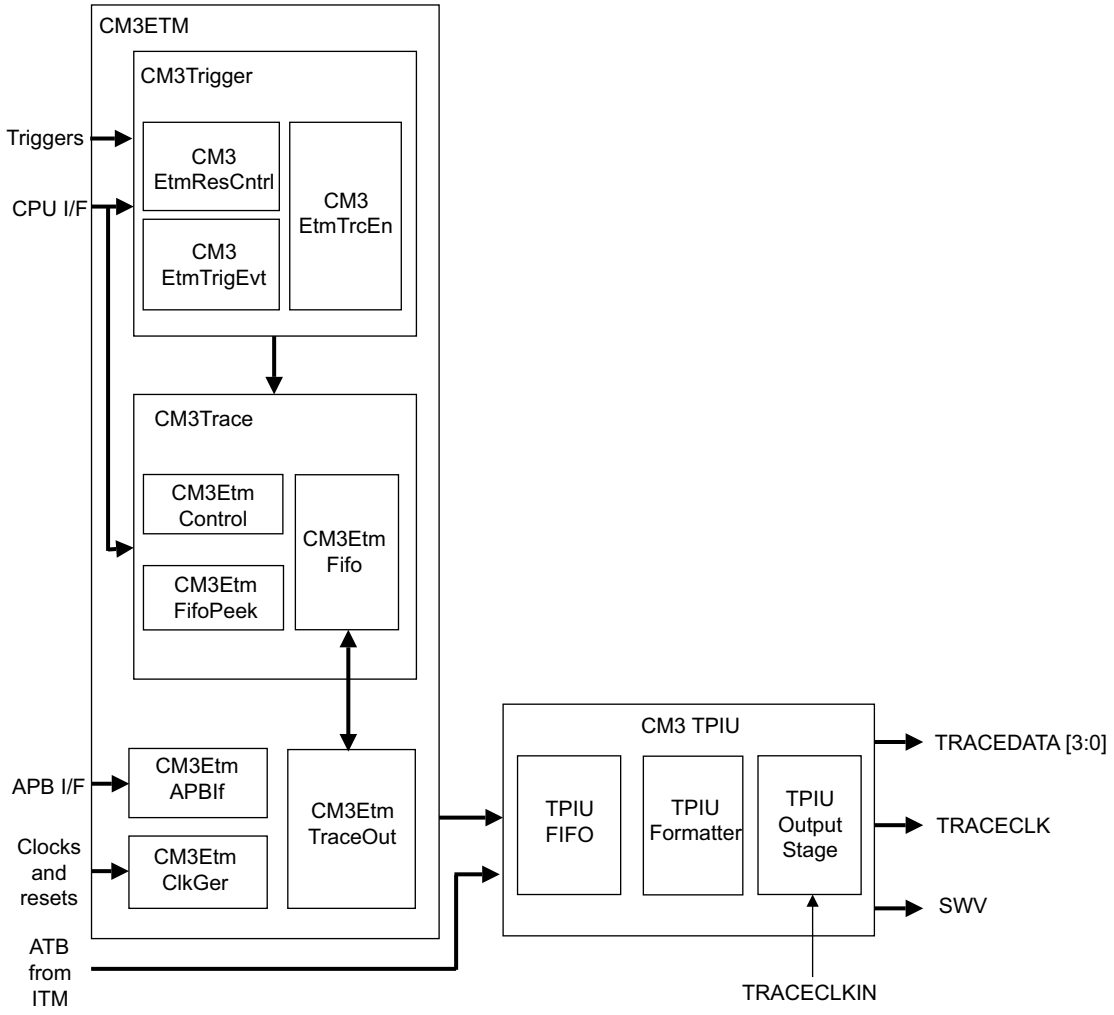


Figure 15-1 ETM block diagram

15.1.2 ETM resources

Table 15-1 lists the Cortex-M3 resources.

Table 15-1 Cortex-M3 resources

Feature	Present on Cortex-M3 ETM
Architecture version	ETMv3.4
Address comparator pairs	0
Data comparators	0
Context ID comparators	0
MMDs	0
Counters	0
Sequencer	No
Start/stop block	Yes
Embedded ICE comparators	4
External inputs	2
External outputs	0
Extended external inputs	0
Extended external input selectors	0
FIFOFULL	Yes
FIFOFULL level setting	Yes
Branch broadcasting	Yes
ASIC Control Register	No
Data suppression	No
Software access to registers	Yes
Readable registers	Yes
FIFO size	16 bytes
Minimum port size	8 bytes
Maximum port size	8 bytes

Table 15-1 Cortex-M3 resources

Feature	Present on Cortex-M3 ETM
Normal port mode	-
Normal half-rate clocking/1:1	Yes - asynchronous
Demux port mode	-
Demux half-rate clocking/1:2	No
Mux port mode/2:1	No
1:4 port mode	No
Dynamic port mode (including stalling)	No. Supported by asynchronous port mode.
CPRT data	No
Load PC first	No
Fetch comparisons	No
Load data traced	No

15.2 Data tracing

The Cortex-M3 system can perform low-bandwidth data tracing using the *Data Watchpoint and Trace* (DWT) and *Instruction Trace Macrocell* (ITM) components. To enable instruction trace to be supported with a low pin-count, data trace is not included in the ETM. This results in a considerable saving in gate count for the ETM, because the triggering resources can be simplified.

When the ETM is implemented in processor, the two trace sources (ITM and ETM) both feed into the *Trace Port Interface Unit* (TPIU), where they are combined and usually output over the trace port. DWT is able to provide either focused data trace, or global data trace (subject to FIFO overflow issues). The TPIU is optimized for the requirements of a single core Cortex-M3 system.

15.3 ETM Resources

Because the ETM does not generate data trace information, the lower bandwidth reduces the requirement for complex triggering capabilities. This means that the ETM does not include the following:

- internal comparators
- counters
- sequencers.

15.3.1 Periodic synchronization

The *Nested Vectored Interrupt Controller* (NVIC) contains a 12-bit counter that can write the PC value to the DWT. This counter is used to create periodic synchronization packets for the ETM.

15.3.2 Data and instruction address compare resources

The DWT provides four address comparators on the data bus which are used to provide debug functionality. Within the DWT unit, the functions triggered by a match can be specified, and one of these functions is to generate an ETM match input. These inputs are presented to the ETM as Embedded ICE comparator inputs.

A single DWT resource can trigger an ETM event and also generate instrumentation trace directly from the same event.

The four DWT comparators can also be individually configured to compare with the execute PC to allow the ETM access to a PC compare resource. These inputs are presented to the ETM as Embedded ICE comparator inputs.

Note

Using a DWT comparator as a PC comparator reduces the number of available data address comparisons.

See *Data Watchpoint and Trace* on page 11-12 for more information about the DWT unit.

External inputs

Two external inputs, **ETMEXTIN[1:0]**, enable additional on-chip IP to generate trigger/enable signals for the ETM.

Start/stop block

The start/stop block controls start/stop behavior by using the embedded ICE inputs to the ETM. These inputs are controlled by the DWT.

15.3.3 FIFO functionality

The FIFO size is 16 bytes.

A FIFOFULL output is provided, but the processor core does not support this as an input. An external mechanism is required to stall the core if FIFOFULL is used. Although stalling the core in a typical application is unlikely to be acceptable, it provides a mechanism for enabling 100% trace which could be compared with the partial trace obtained for a non-stalled run.

15.4 Trace output

The ETM outputs data 8-bits at a time, at the core clock speed. It does not support different trace port sizes and trace port modes. The TPIU is used to export trace output off chip. This output is compatible with the ATB protocol.

Because AFVALID functionality is not supported, the trace port cannot flush data from the ETM FIFO. However, with an 8-bit ATB port the FIFO always drains which makes AFVALID unnecessary.

The Cortex-M3 system is equipped with an optimized TPIU that is designed for use with the ETM and ITM. This TPIU does not support additional trace sources. However, additional trace sources can be added if the TPIU has been replaced with a more complex version, and more trace infrastructure.

Note

A trace ID register and output are provided for systems that use multiple trace sources.

The TPIU uses the ‘formatted’ trace output protocol. This means that there is no requirement for an extra pin for **TRACECTL** signal.

Trace output from the ETM is synchronous to the core clock. There is an asynchronous FIFO in the trace port interface. If you want to integrate the ETM into a multi-core system, you might have to use an asynchronous ATB bridge.

15.5 ETM architecture

The ETM is an instruction only ETM that implements ARM ETM architecture v3.4. It is based on the ARM ETM Architecture specification. For full details, see the *ARM Embedded Trace Macrocell Architecture Specification*.

All Thumb-2 instructions are traced as a single instruction. Instructions following an IT instruction are traced as normal conditional instructions. The decompressor does not have to refer to the IT instruction.

15.5.1 Restartable instructions

The ARMv7-M architecture can restart LSM instructions that are interrupted by an exception. The ETM traces an instruction that has been interrupted by an exception by indicating that it has been cancelled. On return from the exception, the ETM traces the same instruction again, regardless of the instruction being restarted or resumed.

15.5.2 Exception return

The ETM explicitly indicates return from an exception in the trace stream. This is because exception return functionality is encoded in a data-dependent manner, and an exception return behaves differently from a simple branch.

The packet encoding used to indicate a return from an exception is shown in Figure 15-2.

7	6	5	4	3	2	1	0
0	1	1	1	0	1	1	0

Figure 15-2 Exception return packet encoding

If a new, higher priority exception pre-empts the stack pop, the branch to the exception handler must indicate that the last instruction was cancelled, but that the return from exception instruction was not cancelled. If the return from exception packet is present, then this means that the previous instruction is complete.

15.5.3 Exception tracing

To trace exceptions, an optional field is added to a branch packet. This extra field specifies the exception information. A normal branch packet is encoded in 1-5 bytes of trace data, while the exception branch is as follows:

- 2-5 bytes of address

- 1-2 bytes of exception.

The exception mapping is designed to enable the most frequent exceptions to be encoded within one byte. The ETM exception tracing mapping is described in Table 15-2.

Table 15-2 Exception tracing mapping

Number of bytes	Exception	CurrentInt	Traced value
1 byte exception	None	0	0
1 byte exception	IRQ1	17	1
1 byte exception	IRQ2	18	2
1 byte exception	IRQ3	19	3
1 byte exception	IRQ4	20	4
1 byte exception	IRQ5	21	5
1 byte exception	IRQ6	22	6
1 byte exception	IRQ7	23	7
1 byte exception	IRQ0	16	8
1 byte exception	Usage Fault	6	9
1 byte exception	NMI	2	10
1 byte exception	SVC	11	11
1 byte exception	DebugMon	12	12
1 byte exception	MemManage	4	13
1 byte exception	PendSV	14	14
1 byte exception	SysTick	15	15
2 bytes exception	Reserved	8	16
2 bytes exception	Reset	1	17
2 bytes exception	Reserved	10	18
2 bytes exception	HardFault	3	19
2 bytes exception	Reserved	9	20
2 bytes exception	BusFault	5	21

Table 15-2 Exception tracing mapping (continued)

Number of bytes	Exception	CurrentInt	Traced value
2 bytes exception	Reserved	7	22
2 bytes exception	Reserved	13	23
2 bytes exception	IRQ8	24	24
2 bytes exception	IRQ9	25	25
2 bytes exception	IRQ10	26	26
.	.	.	.
.	.	.	.
.	.	.	.
2 bytes exception	IRQ239	512	512

The full branch with exception packet is shown in Figure 15-3 on page 15-12.

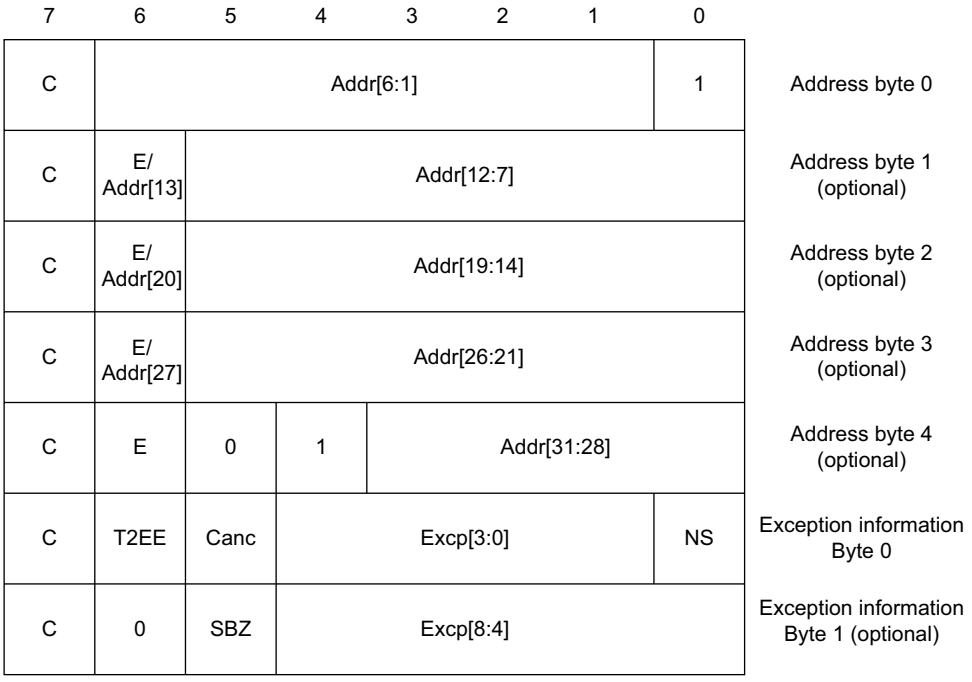


Figure 15-3 Exception encoding for branch packet

The final address byte uses bits [7:6] set to 0b01 to indicate the end of the address field. Exception data follows this field. Exception byte 0 sets bit [7] to 1 if a second exception byte follows. If there is no exception present, and only address bits [6:1] change, then a single byte is used. If an exception is present, then at least two bytes are used to signal the address.

When turning off trace immediately before entry to an exception handler, the ETM remains enabled until the exception is taken. This enables it to trace the branch address, exception type and resume information.

15.6 ETM programmer's model

The ETM programmer's model is described in detail in the ARM Embedded Trace Macrocell Architecture Specification. This section defines the implementation specific features of the ETM programmer's model.

15.6.1 APB interface

The ETM contains an *Advanced Peripheral Bus* (APB) slave interface that can read and write to the ETM registers. This interface is synchronous to the processor clock, and can be accessed by the core and the external debug interface through the SW-DP/JTAG-DP.

15.6.2 List of ETM registers

The ETM registers are listed in Table 15-3. For full details, see the ARM Embedded Trace Macrocell Architecture Specification.

Table 15-3 ETM registers

Name	Type	Address	Present	Description
ETM Control	R/W	0xE0041000	Yes	For a description, see page 15-17.
Configuration Code	RO	0xE0041004	Yes	For a description, see page 15-17.
Trigger event	WO	0xE0041008	Yes	Defines the event that controls the trigger.
ASIC Control	WO	0xE004100C	No	-
ETM Status	RO or R/W	0xE0041010	Yes	Provides information on the current status of the trace and trigger logic.
System Configuration	RO	0xE0041014	Yes	For a description, see page page 15-17.
TraceEnable	WO	0xE0041018, 0xE004101C	No	-
TraceEnable Event	WO	0xE0041020	Yes	Describes the TraceEnable enabling event.
TraceEnable Control 1	WO	0xE0041024	Yes	For a description, see page 15-17.
FIFOFULL Region	WO	0xE0041028	No	-
FIFOFULL Level	WO or R/W	0xE004102C	Yes	Holds the level below which the FIFO is considered full.

Table 15-3 ETM registers

Name	Type	Address	Present	Description
ViewData	WO	0xE0041030-0xE004103C	No	-
Address Comparators	WO	0xE0041040-0xE004113C	No	-
Counters	WO	0xE0041140-0xE004157C	No	-
Sequencer	R/W	0xE0041180-0xE0041194, 0xE0041198	No	-
External Outputs	WO	0xE00411A0-0xE00411AC	No	-
CID Comparators	WO	0xE00411B0-0xE00411BC	No	-
Implementation specific	WO	0xE00411C0-0xE00411DC	No	All RAZ. Ignore writes.
Synchronization Frequency	WO	0xE00411E0	No	Synchronization is generated from DWT.
ETM ID	RO	0xE00411E4	Yes	For a description, see page 15-18.
Configuration Code Extension	RO	0xE00411E8	Yes	For a description, see page 15-18.
Extended External Input Selector	WO	0xE00411EC	No	No extended external inputs implemented.
TraceEnable Start/Stop Embedded ICE	R/W	0xE00411F0	Yes	Bits 19:16 configure E-ICE inputs to use as stop resources. Bits 3:0 configure E-ICE inputs to use as start resources.
Embedded ICE Behavior Control	WO	0xE00411F4	No	Embedded ICE inputs use the default behavior.
CoreSight Trace ID	R/W	0xE0041200	Yes	Implemented as normal.
OS Save/Restore	WO	0xE0041304-0xE0041308	No	OS Save/Restore not implemented. RAZ, ignore writes.
ITMISCIN	RO	0xE0041EE0	Yes	Sets [1:0] to EXTIN[1:0], [4] to COREHALT.
ITTRIGOUT	WO	0xE0041EE8	Yes	Sets [0] to TRIGOUT.

Table 15-3 ETM registers

Name	Type	Address	Present	Description
ITATBCTR2	RO	0xE0041EF0	Yes	Sets [0] to ATREADY.
ITATBCTR0	WO	0xE0041EF8	Yes	Sets [0] to ATVALID.
Integration Mode Control	R/W	0xE0041F00	Yes	Implemented as normal.
Claim Tag	R/W	0xE0041FA0- 0xE0041FA4	Yes	Implements the 4-bit claim tag.
Lock Access	WO	0xE0041FB0- 0xE0041FB4	Yes	Implemented as normal.
Authentication Status	RO	0xE0041FB8	Yes	Implemented as normal.
Device Type	RO	0xE0041FCC	Yes	Reset value: 0x13.
Peripheral ID 4	RO	0xE0041FD0	Yes	0x04
Peripheral ID 5	RO	0xE0041FD4	Yes	0x00
Peripheral ID 6	RO	0xE0041FD8	Yes	0x00
Peripheral ID 7	RO	0xE0041FDC	Yes	0x00
Peripheral ID 0	RO	0xE0041FE0	Yes	0x24
Peripheral ID 1	RO	0xE0041FE4	Yes	0xb9
Peripheral ID 2	RO	0xE0041FE8	Yes	0x0b
Peripheral ID 3	RO	0xE0041FEC	Yes	0x00
Component ID 0	RO	0xE0041FF0	Yes	0x0d
Component ID 1	RO	0xE0041FF4	Yes	0x90
Component ID 2	RO	0xE0041FF8	Yes	0x05
Component ID 3	RO	0xE0041FFC	Yes	0xb1

15.6.3 Description of ETM registers

An additional description of some of the ETM registers is given in the following sections. For full details, see the *ARM Embedded Trace Macrocell Architecture Specification*.

ETM Control Register

The ETM Control Register controls general operation of the ETM, such as whether tracing is enabled.

Reset value: `0x00002411`

Implemented bits: 21, 17:16, 13, 11:4, 0

All other bits RAZ, ignore writes.

Configuration Code Register

The ETM Configuration Code Register enables the debugger to read the implementation specific configuration of the ETM.

Reset value: `0x8C800000`

Bits 22:20 fixed at 0 and not supplied by the ASIC. Bits 18:17 are supplied by the MAXEXTIN[1:0] input bus, and read the lower value of MAXEXTIN and the number 2 (the number of EXTINs). This indicates:

- software accesses supported
- trace start/stop block present
- no CID comparators
- FIFOFULL logic is present
- no external outputs
- 0-2 external inputs (controlled by MAXEXTIN)
- no sequencer
- no counters
- no MMDs
- no data comparators
- no address comparator pairs.

System Configuration Register

The System Configuration Register shows the ETM features supported by the ASIC.

Reset value: `0x00020D09`

Bits 11:10 are implemented as normal. Bits 9, 2:0 are fixed as 4’b0001.

TraceEnable Control 1 Register

The TraceEnable Control 1 Register is one of the registers that configures TraceEnable.

Only bit 25 is implemented. It controls the start/stop resource controls tracing.

ETM ID Register

The ETM ID Register holds the ETM architecture variant, and precisely defines the Programmer's Model for the ETM.

Reset value: `0x4114F240`

This indicates:

- ARM implementor.
- Special branch encoding. Affects bits 7:6 of each byte.
- Thumb-2 supported.
- Core family is found elsewhere.
- ETMv3.4.
- Implementation revision 0.

Configuration Code Extension Register

The Configuration Code Extension Register holds additional bits for ETM configuration code. It describes the extended external inputs.

Reset value: `0x00018800`

This register indicates:

- start/stop block uses E-ICE inputs
- four embedded ICE inputs
- no data comparisons supported
- all registers are readable
- no extended external input supported.

Chapter 16
Embedded Trace Macrocell Interface

This chapter describes the *Embedded Trace Macrocell* (ETM) interface. It contains the following sections:

- *About the ETM interface* on page 16-2
- *CPU ETM interface port descriptions* on page 16-3
- *Branch status interface* on page 16-5.

16.1 About the ETM interface

The ETM interface enables simple connection of an ETM to the processor. It provides a channel for instruction trace to the ETM.

16.2 CPU ETM interface port descriptions

The processor has a port which enables the ETM to determine the instruction execution sequence. These port descriptions are described in Table 16-1.

Table 16-1 ETM interface ports

Port name	Direction	Qualified by	Description
ETMIVALID	Output	-	Instruction in execute is valid. Marks that an opcode has entered the first cycle of execute.
ETMIBRANCH	Output	ETMIVALID	Opcode is a branch target. Marks that current code is the destination of a PC modifying event (branch, interrupt processing).
ETMIINDBR	Output	ETMIBRANCH	Opcode branch target is indirect. Marks that the current opcode is a branch target whose destination cannot be deduced from the PC contents. For example, LSU, register move, or interrupt processing.
ETMVALID	Output	No qualifier	Signals that the current data address as seen by the DWT is valid on this cycle.
ETMICCFAIL	Output	ETMIVALID	Opcode condition code fail or pass. Marks if the current opcode has failed or passed its conditional execution check. An opcode is conditionally executed if it is a conditional branch, or for all other opcode found in an IT block.
ETMINTSTAT[2:0]	Output	No qualifier	Interrupt status. Marks the interrupt status of the current cycle: 000 no status 001 interrupt entry 010 interrupt exit 011 interrupt return 100 - Vector fetch and stack push ETMINTSTAT Entry/Return is asserted in the first cycle of the new interrupt context. Exit occurs without ETMIVALID.
ETMINTNUM[8:0]	Output	ETMINTSTAT	Interrupt number. Marks the interrupt number of the current execution context.

Table 16-1 ETM interface ports (continued)

Port name	Direction	Qualified by	Description
ETMIA[31:1]	Output	-	Instruction address. Indicates the current fetch address of the opcode in execution, or of the last opcode executed. You can determine the context by examining: ETMIVALID HALTED SLEEPING The ETM examines this net when ETMIVALID and ETMIBRANCH are asserted. The DWT examines this net for PC samples and bus watching.
ETMFOLD	Output	ETMIVALID	Opcode fold. Indicates that an IT or NOP opcode has been folded in this cycle. PC advances past the current (16-bit) opcode and the IT/NOP instruction (16 bits). This affects the ETMIA.
ETMFLUSH	Output	-	Flush marker of PC event. A PC modifying opcode has executed or an interrupt push/pop has started. The ETM can use this control to complete outstanding packets in preparation for an ETMIBRANCH event.
ETMFINDBR	Output	ETMFLUSH	Flush is indirect. Marks that the flush hint destination cannot be deduced from the PC.
ETMCANCEL	Output	-	Current opcode in execute has been cancelled. Opcodes that are interrupted restart or continue on return to this execution context. These include: LDR/STR LDRD/STRD LDM/STM U/SMULL MLA U/SDIV MSR CPSID

16.3 Branch status interface

The branch status signal, **BRCHSTAT**, gives fetch time information about the opcode in decode and the next execute.

Figure 16-1 and Figure 16-2 show a conditional branch backwards not taken and taken. The branch occurs speculatively in the decode phase of the opcode. The branch target is a halfword unaligned 16-bit opcode.

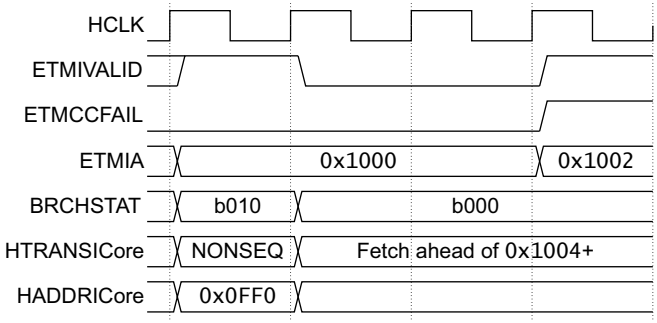


Figure 16-1 Conditional branch backwards not taken

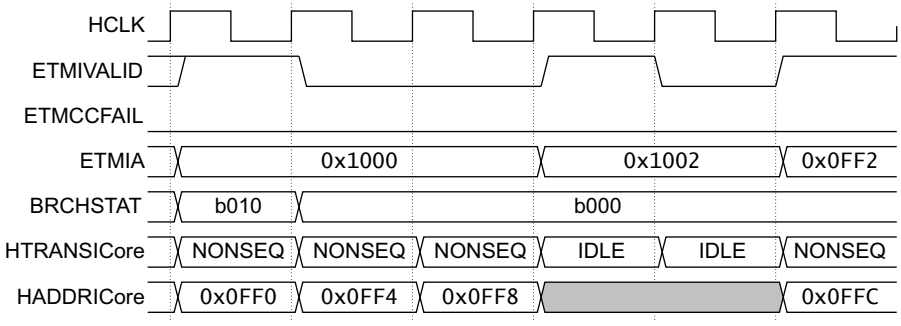


Figure 16-2 Conditional branch backwards taken

Note

HADDRICore and **HTRANSICore** are the address and transaction request signals from the processor, and not the signals on the external Cortex-M3 interface.

Figure 16-3 on page 16-6 and Figure 16-4 on page 16-6 show a conditional branch forwards not taken and taken. The branch occurs speculatively in the decode phase of the opcode. The branch target is a halfword unaligned 16-bit opcode.

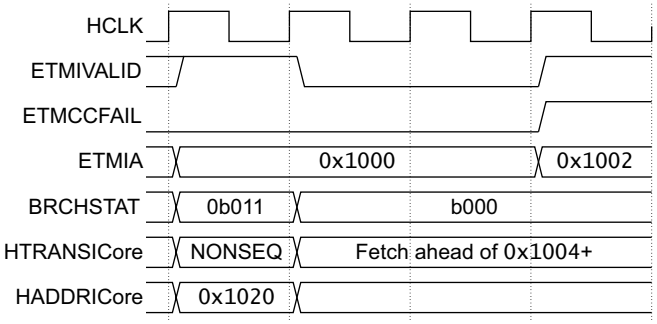


Figure 16-3 Conditional branch forwards not taken

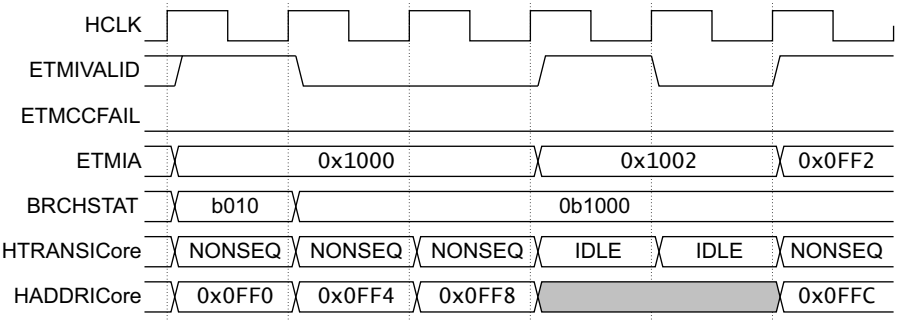


Figure 16-4 Conditional branch forwards taken

Figure 16-5 and Figure 16-6 on page 16-7 show an unconditional branch in this cycle, during the execute phase of the preceding opcode without and with pipeline stalls. The branch occurs in the decode phase of the opcode. The branch target is an aligned 32-bit opcode.

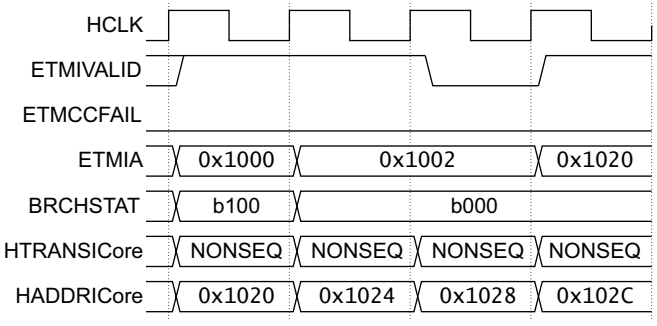


Figure 16-5 Unconditional branch without pipeline stalls

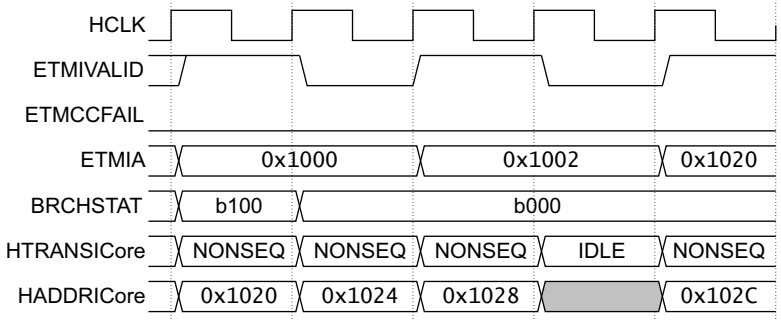


Figure 16-6 Unconditional branch with pipeline stalls

Figure 16-7 and Figure 16-8 show an unconditional branch in the next opcode. The branch occurs in the execute phase of the opcode. The branch target is an aligned and unaligned 32-bit ALU opcode.

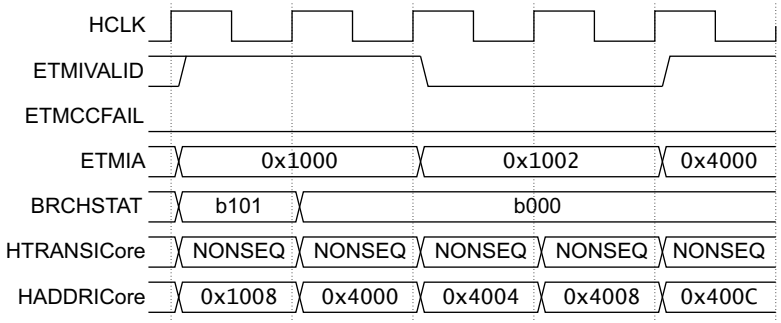


Figure 16-7 Unconditional branch in execute aligned

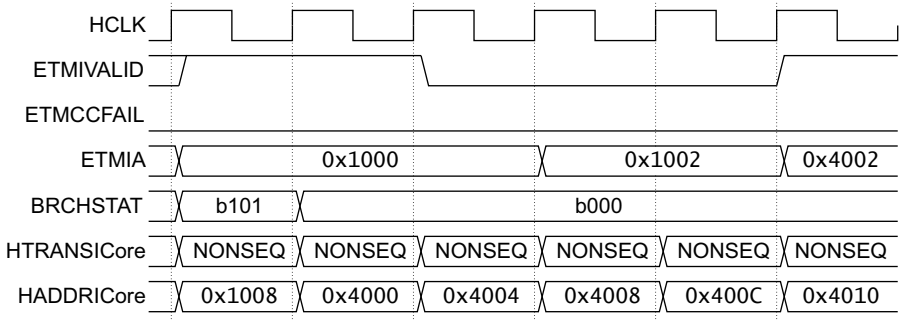


Figure 16-8 Unconditional branch in execute unaligned

Chapter 17
Instruction Timing

This chapter describes the instruction timings of the processor. It contains the following sections:

- *About instruction timing* on page 17-2
- *Processor instruction timings* on page 17-3
- *Load-store timings* on page 17-7

17.1 About instruction timing

The timing information provided in this chapter covers each instruction as well as interactions between instructions. It also contains information about factors which influence timings.

When looking at timings, it is important to understand the role that the system architecture plays. Every instruction must be fetched and every load/store must go out to the system. These factors are discussed here along with intended system design (and the implications for timing).

17.2 Processor instruction timings

Table 17-1 represents the Thumb-2 subset supported in the ARMv7-M architecture. It provides cycle information including annotations to explain how instruction stream interactions will affect timing. System effects, such as running code from slower memory, are also considered.

Table 17-1 Instruction timings

Instruction type	Size	Cycles count	Description
Data operations	16	1 (+P[1] if PC is destination)	ADC, ADD, AND, ASR, BIC, CMN, CMP, CPY, EOR, LSL, LSR, MOV, MUL, MVN, NEG, ORR, ROR, SBC, SUB, TST, REV, REVH, REVSH, SXTB, SXTH, UXTB, and UXTH. MUL is one cycle.
Branches	16	1+P[1]	B<cond>, B, BL, BX, and BLX. Note, no BLX with immediate. If branch taken, pipeline reloads (two cycles are added).
Load-store Single	16	2[2] (+P[1] if PC is destination)	LDR, LDRB, LDRH, LDRSB, LDRSH, STR, STRB, and STRH, and "T" variants.
Load-store Multiple	16	1+N[2] (+P[1] if PC loaded)	LDMIA, POP, PUSH, and STMIA.
Exception generating	16	-	BKPT stops in debug if debug enabled, fault if debug disabled. SVC faults to SVCall handler (see ARMv7-M architecture specification for details).
Data operations with immediate	32	1 (+P[1] if PC is destination)	ADC{S}. ADD{S}, CMN, RSB{S}, SBC{S}, SUB{S}, CMP, AND{S}, TST, BIC{S}, EOR{S}, TEQ, ORR{S}, MOV{S}, ORN{S}, and MVN{S}.
Data operations with large immediate	32	1	MOVW, MOVT, ADDW, and SUBW. MOVW and MOVT have a 16-bit immediate (so can replace literal loads from memory). ADDW and SUBW have a 12-bit immediate (so also can replace many from memory literal loads).
Bit-field operations	32	1	BFI, BFC, UBFX, and SBFX. These are bitwise operations allowing control of position and size in bits. These both support C/C++ bit fields (in structs) as well as many compare and some AND/OR assignment expressions.
Data operations with 3 register	32	1 (+P[1] if PC is destination)	ADC{S}. ADD{S}, CMN, RSB{S}, SBC{S}, SUB{S}, CMP, AND{S}, TST, BIC{S}, EOR{S}, TEQ, ORR{S}, MOV{S}, ORN{S}, and MVN{S}. No PKxxx instructions.

Table 17-1 Instruction timings (continued)

Instruction type	Size	Cycles count	Description
Shift operations	32	1	ASR{S}, LSL{S}, LSR{S}, and ROR {S}.
Miscellaneous	32	1	REV, REVH, REVSH, RBIT, CLZ, SXTB, SXTH, UXTB, and UXTH. Extension instructions same as corresponding ARM v6 16-bit instructions.
Table Branch	16	4+P[1]	Table branches for switch/case use. These are LDR with shifts and then branch.
Multiply	32	1 or 2	MUL, MLA, and MLS. MUL is one cycle and MLA and MLS are two cycles.
Multiply with 64-bit result	32	3-7[3]	UMULL, SMULL, UMLAL, and SMLAL. Cycle count based on input sizes. That is, ABS(inputs) < 64K will early terminate.
Load-store addressing	32	-	Supports Format PC+/-imm12, Rbase+imm12, Rbase+/-imm8, and adjusted register including shifts. "T" variants used when in Privilege mode.
Load-store Single	32	2[2] (+P[1] if PC is destination)	LDR, LDRB, LDRSB, LDRH, LDRSH, STR, STRB, and STRH, and "T" variants. PLD is a hint, so acts as NOP if no cache.
Load-store Multiple	32	1+N[2] (+P[1] if PC is loaded)	STM, LDM, LDRD, STRD, LDC, and STC.
Branches	32	1+N[2]	LDREX, STREX, LDREXB, LDREXH, STREXB, STREXH, CLREX. Will fault if no local monitor (is IMP DEF). LDREXD and STREXD are not included in this profile.
Load-store Special	32	1+P[1]	B, BL, and B<cond>. No BLX (1) since always changes state. No BXJ.
System	32	1	MSR(2) and MRS(2) replace MSR/MRS but also do more. These are used to access the other stacks and also the status registers. CPSIE/CPSID 32-bit forms are not supported. No RFE or SRS.
System	16	1	CPSIE and CPSID are quick versions of MSR(2) instructions and use the standard Thumb-2 encodings, but only allow use of "i" and "f" and not "a".

Table 17-1 Instruction timings (continued)

Instruction type	Size	Cycles count	Description
Extended32	32	1	NOP, Coprocessor (LDC, MCR, MCR2, MCRRMRC, MRC2, MRRC, and STC), and YIELD (hinted NOP). Note, no MRS (1), MSR (1), or SUBS (PC return link).
Combined Branch	16	1+P[1]	CBZ.
Extended	16	0-1[4]	IT and NOP (includes YIELD).
Divide	32	2-8[5]	SDIV and UDIV. 32/32 divides both signed and unsigned with 32-bit quotient result (no remainder, it can be derived by subtraction). This will early out when dividend and divisor are close in size.
Sleep	32	1+W[6]	WFI, WFE, and SEV are in the class of "hinted NOP" instructions used to control sleep behavior.
Barriers	16	1+B[7]	ISB, DSB, and DMB are barrier instructions which ensure certain actions have taken place before the next instruction is executed.
Saturation	32	1	SSAT and USAT perform saturation on a register. They perform 3 tasks: normalize the value using shift, test for overflow from a selected bit position (the Q value) and set the xPSR Q bit if so, saturate the value if overflow detected. Saturation refers to the largest unsigned value or the largest/smallest signed value for the size selected.

Cycle count information:

- P = pipeline reload
- N = count of elements
- W = sleep wait
- W = sleep wait

In general, each instruction takes one cycle (one core clock) to start executing (as shown above). Additional cycles may be taken due to fetch stalls.

1. Branches take one cycle for instruction and then pipeline reload for target instruction. Non-taken branches are 1 cycle total. Taken branches with an immediate are normally 1 cycle of pipeline reload (2 cycles total). Taken branches with register operand are normally 2 cycles of pipeline reload (3 cycles total). Pipeline reload is longer when branching to unaligned 32-bit instructions as well

as due to accesses to slower memory. A branch hint is emitted to the code bus, see section 14.2.1, which allows a slower system to pre-load; this can reduce the branch target penalty for slower memory, but never less than what is shown here.

2. Generally, load-store instructions take two cycles for the first access and one cycle for each additional access.

3. UMULL/SMULL/UMLAL/SMLAL use early termination depending on the size of source values. These are interruptible (abandoned/restarted), with worst case latency of one cycle. MLAL versions take four to seven cycles and MULL versions take three to five cycles. In both cases, the signed version is one cycle longer than the unsigned.

4. DIV timings depend on dividend and divisor. DIV is interruptible (abandoned/restarted), with worst case latency of one cycle. When dividend and divisor are similar in size, divide will terminate quickly. Minimum time is for cases of divisor larger than dividend and divisor of zero. A divisor of zero returns zero (not a fault), although a debug trap is available to catch this case.

5. Sleep is one cycle for the instruction plus as many sleep cycles as appropriate. WFE only uses one cycle when event has been. WFI will normally be more than one cycle unless an interrupt happens to pend exactly when entering WFI.

6. ISB takes one cycle (acts as branch). DMB and DSB will take one cycle unless data is pending in the write buffer or LSU. If an interrupt comes in during a barrier, it will be abandoned/restarted.

17.3 Load-store timings

This section describes how to optimally pair instructions. This achieves further reductions in timing.

- STR Rx,[Ry,#imm] is always one cycle. This is because the address generation is performed in the initial cycle, and the data store is performed at the same time as the next instruction is executing. If the store is to the store buffer, and the store buffer is full, the next instruction will be delayed until the store can complete. If the store is not to the store buffer (such as to the Code segment) and that transaction stalls, the impact on timing will only be felt if another load or store operation is executed before completion.
- LDR Rx!,[any] is not normally pipelined. That is, base update load will generally be at least a two-cycle operation (more if stalled). However, if the next instruction does not need to read from a register, the load will be reduced to one cycle. Non register reading instructions include CMP, TST, NOP, non-taken IT controlled instructions, and so on.
- LDR PC,[any] is always a blocking operation. This means minimally two cycles for the load, and three cycles for the pipeline reload. So at least five cycles (more if stalled on the load or the fetch).
- LDR Rx,[PC,#imm] may add a cycle due to contention with the fetch unit.
- TBB and TBH are also blocking operations. These are minimally two cycles for the load, one cycle for the add, and three cycles for the pipeline reload. This means at least six cycles (more if stalled on the load or the fetch).
- LDR any are pipelined when possible. This means that if the next instruction is an LDR or non-base updating STR, and the destination of the first LDR is not used to compute the address for the next instruction, then one cycle is removed from the cost of the next instruction. So, an LDR may be followed by an STR, such that the STR writes out what the LDR loaded. Further, multiple LDRs may be pipelined together. Some optimized examples:
  - LDR R0,[R1]; LDR R1,[R2] - normally three cycles total
  - LDR R0,[R1,R2]; STR R0,[R3,#20] - normally three cycles total
  - LDR R0,[R1,R2]; STR R1,[R3,R2] - normally three cycles total
  - LDR R0,[R1,R5]; LDR R1,[R2]; LDR R2,[R3,#4] - normally four cycles total.
- STR any may not be pipelined after. STR may only be pipelined when after an LDR, but nothing may be pipelined after the store. Even a stalled STR will normally only take two cycles, because of the store buffer (used for bit band, data

segment, and unaligned). Further, if the STR is followed by ALU operations (not a load or store), then the processor will drain the store without waiting at all. So, load-store operations should be avoided after STR instructions when possible.

- LDREX and STREX may be pipelined exactly as LDR. Because STREX is treated more like an LDR, it may be pipelined as explained for LDR. Equally LDREX is treated exactly as an LDR and so may be pipelined.
- LDRD, STRD may not be pipelined with preceding or following instructions. However, the two words are pipelined together. So, three cycles when not stalled.
- LDM, STM, LDC, STC may not be pipelined with preceding or following instructions. However, all elements after the first are pipelined together. So, a three element LDM will take 2+1+1 or 5 cycles when not stalled. Likewise, an eight element store will take nine cycles when not stalled. When interrupted, LDM and STM instructions will continue from where left off when returned to. The continue operation will add one or two cycles to the first element to get started.
- Unaligned Word or Halfword Loads or stores will add penalty cycles. A byte aligned halfword load or store will add one extra cycle to perform the operation as two bytes. A halfword aligned word load or store will add one extra cycle to perform the operation as two halfwords. A byte-aligned word load or store will add two extra cycles to perform the operation as a byte, a halfword, and a byte. These numbers will be increased if the memory stalls. A STR or STRH may not delay the processor due to the store buffer.

Appendix A
Signal Descriptions

This appendix lists and describes the processor interface signals. It contains the following sections:

- *Clocks* on page A-2
- *Resets* on page A-3
- *Miscellaneous* on page A-4
- *Interrupt interface* on page A-5
- *ICode interface* on page A-6
- *DCode interface* on page A-7
- *System bus interface* on page A-8
- *Private Peripheral Bus interface* on page A-9
- *ITM interface* on page A-10
- *AHB-AP interface* on page A-11
- *ETM interface* on page A-12
- *Test interface* on page A-13.

A.1 Clocks

Table A-1 lists the clock signals.

Table A-1 Clock signals

Name	Direction	Description
HCLK	Input	Main Cortex-M3 clock
FCLK	Input	Free-running Cortex-M3 clock
DAPCLK	Input	AHB-AP clock

A.2 Resets

Table A-2 lists the reset signals.

Table A-2 Reset signals

Name	Direction	Description
PORESETn	Input	Power-on reset. Resets entire Cortex-M3 system.
SYSRESETn	Input	System reset. Resets processor, non-debug portion of NVIC, Bus Matrix, and MPU. Debug components are not reset.
SYSRESETREQ	Output	System reset request.
DAPRESETn	Input	AHB-AP reset.

A.3 Miscellaneous

Table A-3 lists the leftover signals.

Table A-3 Miscellaneous signals

Name	Direction	Description
LOCKUP	Output	Indicates that the core is locked up.
SLEEPDEEP	Output	Indicates that the Cortex-M3 clock can be stopped.
SLEEPING	Output	Indicates that the Cortex-M3 clock can be stopped.
CURRPRI[7:0]	Output	Indicates what priority interrupt (or base boost) is currently used. **CURRPRI** represents the pre-emption priority, and does not indicate the secondary priority.
HALTED	Output	In halting debug mode. **HALTED** remains asserted while the core is in debug.
TXEV	Output	Event transmitted as a result of SEV instruction. This is a single cycle pulse.
TRCENA	Output	Trace Enable. This signal reflects the setting of bit [24] of the Debug Exception and Monitor Control Register. This signal can be used to gate the clock to the TPIU and ETM blocks to reduce power consumption when trace is disabled.
BIGEND	Input	Static endian select: 1 = big-endian 0 = little-endian This signal is sampled at reset, and cannot be changed when reset is inactive.
EDBGRQ	Input	External debug request.
PPBLOCK[5:0]	Input	Reserved. Must be tied to 6’b000000.
STCLK	Input	System Tick Clock.
STCALIB[25:0]	Input	System Tick Calibration.
RXEV	Input	Causes a wakeup from a WFE instruction.
VECTADDR[9:0]	Input Reserved	Must be tied to 10'b0000000000
VECTADDREN	Input Reserved	Must be tied to 1'b0.

A.4 Interrupt interface

Table A-4 lists the signals of the external interrupt interface.

Table A-4 Interrupt interface

Name	Direction	Description
INTISR[239:0]	Input	External interrupt signals
INTNMI	Input	Non-maskable interrupt

A.5 ICode interface

Table A-5 lists the signals of the ICode interface.

Table A-5 ICode interface

Name	Direction	Description
HADDRI[31:0]	Output	32-bit instruction address bus
HTRANSI[1:0]	Output	Indicates whether the current transfer is IDLE or NONSEQUENTIAL.
HSIZEI[2:0]	Output	Indicates the size of the instruction fetch. All instruction fetches are 32-bit on Cortex-M3.
HBURSTI[2:0]	Output	Indicates if the transfer is part of a burst. All instruction fetches and literal loads are performed as SINGLE on Cortex-M3.
HPROTI[3:0]	Output	Provides information on the access. Always indicates cacheable and bufferable on this bus. **HPROTI[0]** = 0 indicates instruction fetch **HPROTI[0]** = 1 indicates vector fetch
MEMATTRI[1:0]	Output	Memory attributes. Always 00 for this bus (non-allocate, non-shareable).
BRCHSTAT[3:0]	Output	Provides hint information on the current or coming AHB fetch requests. Conditional opcodes could be a speculation and subsequently discarded. 0000 No hint 0001 Conditional branch backwards in decode 0010 Conditional branch in decode 0011 Conditional branch in execute 0100 Unconditional branch in decode 0101 Unconditional branch in execute 0110 Reserved 0111 Reserved 1000 Conditional branch in decode taken (cycle after **IHTRANS**) 1001 ... 1111 Reserved
HRDATAI[31:0]	Input	Instruction read bus.
HREADYI	Input	When HIGH indicates that a transfer has completed on the bus. This signal is driven LOW to extend a transfer.
HRESPI[1:0]	Input	The transfer response status. OKAY or ERROR.

A.6 DCode interface

Table A-6 lists the signals of the DCode interface.

Table A-6 DCode interface

Name	Direction	Description
HADDRD[31:0]	Output	32-bit data address bus
HTRANSD[1:0]	Output	Indicates whether the current transfer is IDLE, NONSEQUENTIAL, or SEQUENTIAL.
HWRITED	Output	Write not read
HSIZED[2:0]	Output	Indicates the size of the access. Can be 8, 16, or 32 bits.
HBURSTD[2:0]	Output	Indicates if the transfer is part of a burst. Data accesses are performed as INCR on Cortex-M3.
HPROTD[3:0]	Output	Provides information on the access.
EXREQD		Exclusive request.
MEMATTRD[1:0]	Output	Memory attributes. Bit 0 = allocate Bit 1 = shareable.
HWDATAD[31:0]	Input	Data read bus.
HREADYD	Input	When HIGH indicates that a transfer has completed on the bus. This signal is driven LOW to extend a transfer.
HRESPD[1:0]	Input	The transfer response status. OKAY or ERROR.
HRDATAD[31:0]	Input	Read data.
EXRESPD	Input	Exclusive response.

A.7 System bus interface

Table A-7 lists the signals of the system bus interface.

Table A-7 System bus interface

Name	Direction	Description
HADDRS[31:0]	Output	32-bit address.
HTRANSS[1:0]	Output	Indicates the type of the current transfer. Can be IDLE, NONSEQUENTIAL, OR SEQUENTIAL.
HSIZES[2:0]	Output	Indicates the size of the access. Can be 8, 16, or 32 bits.
HBURSTS[2:0]	Output	Indicates if the transfer is part of a burst.
HPROTS[3:0]	Output	Provides information on the access.
HWDATAS[31:0]	Output	32-bit write data bus.
HWRITES	Output	Write not read.
HMASTLOCKS	Output	Indicates a transaction that must be atomic on the bus. This is only used for bit-band writes (performed as read-modify-write).
EXREQS	Output	Exclusive request.
MEMATTRS[1:0]	Output	Memory attributes. Bit 0 = Allocate, Bit 1 = shareable.
HRDATAS[31:0]	Input	Read data bus.
HREADYS	Input	When HIGH indicates that a transfer has completed on the bus. The signal is driven LOW to extend a transfer.
HRESPS[1:0]	Input	The transfer response status. OKAY or ERROR.
EXRESPS	Input	Exclusive response.

A.8 Private Peripheral Bus interface

Table A-8 lists the signals of the DCode interface.

Table A-8 Private Peripheral Bus interface

Name	Direction	Description
PADDR[19:2]	Output	17-bit address. Only the bits that are relevant to the External Private Peripheral Bus are driven.
PADDR31	Output	This signal is driven HIGH when the AHB-AP is the requesting master. It is driven LOW when DCore is the requesting master.
PSEL	Output	Indicates that a data transfer is requested.
PENABLE	Output	Strobe to time all accesses. Used to indicate the second cycle of an APB transfer.
PWDATA[31:0]	Output	32-bit write data bus.
PWRITE	Output	Write not read.
PRDATA[31:0]	Input	Read data bus.

A.9 ITM interface

Table A-9 lists the signals of the ITM interface.

Table A-9 ITM interface

Name	Direction	Description
ATVALID	Output	ATB valid
AFREADY	Output	ATB flush
ATDATA[7:0]	Output	ATB data
ATIDITM[6:0]	Output	ITM ID for TPIU
ATREADY	Input	ATB ready

A.10 AHB-AP interface

Table A-10 lists the signals of the AHB-AP interface.

Table A-10 AHB-AP interface

Name	Direction	Description
DAPRDATA[31:0]	Output	The read bus is driven by the selected AHB-AP during read cycles (when **DAPWRITE** is LOW).
DAPREADY	Output	The AHB-AP uses this signal to extend a DAP transfer.
DAPSLVERR	Output	The error response is because of: • Master port produced an error response, or transfer not initiated because of **DAPEN** preventing a transfer. • Access to AP register not accepted after a **DAPABORT** operation.
DAPCLKEN	Input	DAP clock enable (power saving).
DAPEN	Input	AHB-AP enable.
DAPADDR[31:0]	Input	DAP address bus.
DAPSEL	Input	Select signal generated from the DAP decoder to each AP. This signal indicates that the slave device is selected, and a data transfer is required. There is a **DAPSEL** signal for each slave. The signal is not generated by the driving DP. The decoder monitors the address bus and asserts the relevant **DAPSEL.**
DAPENABLE	Input	This signal is used to indicate the second and subsequent cycles of a DAP transfer from DP to AHB-AP.
DAPWRITE	Input	When HIGH indicates a DAP write access from DP to AHB-AP. When LOW indicates a read access.
DAPWDATA[31:0]	Input	The write bus is driven by the DP block during write cycles (when **DAPWRITE** is HIGH).
DAPABORT	Input	Aborts the current transfer. The AHB-AP returns **DAPREADY** HIGH without affecting the state of the transfer in progress in the AHB Master Port.

A.11 ETM interface

Table A-11 lists the signals of the ETM interface.

Table A-11 ETM interface

Name	Direction	Description
ETMTRIGGER[3:0]	Output	Trigger from DWT. One bit for each of the four DWT comparators.
ETMTRIGINOTD[3:0]	Output	Indicates if the ETM is triggered on an instruction or data match.
ETMIVALID	Output	Instruction valid.
ETMIA[31:1]	Output	PC of the instruction being executed.
ETMICCFAIL	Output	Condition Code fail. Indicates if the current instruction has failed or passed its conditional execution check.
ETMIBRANCH	Output	Opcode is a branch target.
ETMIINDBR	Output	Opcode is an indirect branch target.
ETMINTSTAT[2:0]	Output	Interrupt status. Marks interrupt status of current cycle. 000 - no status 001 - interrupt entry 010 - interrupt exit 011 - interrupt return 100 - vector fetch and stack push. **ETMINTSTAT** entry/return is asserted in the first cycle of the new interrupt context. Exit occurs without **ETMIVALID**.
ETMINTNUM[8:0]	Output	Marks the interrupt number of the current execution context.
ETMFLUSH	Output	A PC modifying opcode has executed, or an interrupt push/pop has started.
ETMPWRUP	Output	ETM is enabled
ETMDVALID	Output	Data valid
ETMCANCEL	Output	Instruction cancelled
ETMFINDBR	Input	Flush is indirect. Marks flush hint destination cannot be inferred from the PC.
ETMFOLD		Opcode fold. An IT or NOP opcode has been folded in this cycle. PC advances past the current (16-bit) opcode plus the IT/NOP instruction (16 bits). This is reflected in the ETMIA.
DSYNC		Synchronization pulse from DWT.

A.12 Test interface

Table A-12 lists the signals of the Test interface.

Table A-12 Test interface

Name	Direction	Description
SE	Input	Scan enable.
RSTBYPASS	Input	Reset bypass for scan testing. **PORESETn** is the only reset used during scan testing.

Glossary

This glossary describes some of the terms used in technical documents from ARM Limited.

Abort A mechanism that indicates to a core that the attempted memory access is invalid or not allowed or that the data returned by the memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid or protected instruction or data memory.

See also Data Abort, External Abort and Prefetch Abort.

Addressing modes Various mechanisms, shared by many different instructions, for generating values used by the instructions.

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 and AHB-Lite.

Advanced Microcontroller Bus Architecture (AMBA)
A family of protocol specifications that describe a strategy for the interconnect. AMBA is the ARM open standard for on-chip buses. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a *System-on-Chip* (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules.

Advanced Peripheral Bus (APB)
A simpler bus protocol than AXI and AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

AHB *See* Advanced High-performance Bus.

AHB Access Port (AHB-AP)
An optional component of the DAP that provides an AHB interface to a SoC.

AHB-AP *See* AHB Access Port.

AHB-Lite A subset of the full AMBA AHB protocol specification. It provides all of the basic functions required by the majority of AMBA AHB slave and master designs, particularly when used with a multi-layer AMBA interconnect. In most cases, the extra facilities provided by a full AMBA AHB interface are implemented more efficiently by using an AMBA AXI protocol interface.

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.

Advanced Trace Bus (ATB)
A bus used by trace devices to share CoreSight capture resources.

APB *See* Advanced Peripheral Bus.

Application Specific Integrated Circuit (ASIC)
An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.

Application Specific Standard Part/Product (ASSP)
An integrated circuit that has been designed to perform a specific application function. Usually consists of two or more separate circuit functions combined as a building block suitable for use in a range of products for one or more specific application markets.

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, ARMv7-M architecture.

ARM instruction An instruction of the ARM *Instruction Set Architecture* (ISA). These cannot be executed by the Cortex-M3.

ARM state The processor state in which the processor executes the instructions of the ARM ISA. The processor only operates in Thumb state, never in ARM state.

ASIC *See* Application Specific Integrated Circuit.

ASSP *See* Application Specific Standard Part/Product.

ATB *See* Advanced Trace Bus.

ATB bridge A synchronous ATB bridge provides a register slice to facilitate timing closure through the addition of a pipeline stage. It also provides a unidirectional link between two synchronous ATB domains.

An asynchronous ATB bridge provides a unidirectional link between two ATB domains with asynchronous clocks. It is intended to support connection of components with ATB ports residing in different clock domains.

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 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.

Beat Alternative word for an individual data transfer within a burst. For example, an INCR4 burst comprises four beats.

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.

Boundary scan chain

A boundary scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

Branch folding Branch folding is a technique where the branch instruction is completely removed from the instruction stream presented to the execution pipeline.

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.

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.

See also Beat.

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.

Clock gating Gating a clock signal for a macrocell with a control signal and using the modified clock that results to control the operating state of the macrocell.

Clocks Per Instruction (CPI)
See Cycles Per Instruction (CPI).

Cold reset Also known as power-on reset.

See also Warm reset.

Context The environment that each process operates in for a multitasking operating system.

See also Fast context switch.

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.

CPI *See* Cycles per instruction.

Cycles Per instruction (CPI)
Cycles per instruction (or clocks per instruction) is a measure of the number of computer instructions that can be performed in one clock cycle. This figure of merit can be used to compare the performance of different CPUs that implement the same instruction set against each other. The lower the value, the better the performance.

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.

DCode Memory Memory space at `0x00000000` to `0x1FFFFFFF`.

Debug Access Port (DAP)
A TAP block that acts as an AMBA, AHB or AHB-Lite, master for access to a system bus. The DAP is the term used to encompass a set of modular blocks that support system wide debug. The DAP is a modular component, intended to be extendable to support optional access to multiple systems such as memory mapped AHB and CoreSight APB through a single debug interface.

Debugger A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Embedded Trace Macrocell (ETM)
A hardware macrocell that, when connected to a processor core, outputs instruction trace information on a trace port.

Endianness Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

See also Little-endian and Big-endian

ETM See *Embedded Trace Macrocell*.

Exception An error or event which can cause the processor to suspend the currently executing instruction stream and execute a specific exception handler or interrupt service routine. The exception could be an external interrupt or NMI, or it could be 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 service routine to deal with the exception.

Exception handler
See Interrupt service routine.

Exception vector *See* Interrupt vector.

External PPB PPB memory space at `0xE0040000` to **`0xE00FFFFF`**.

Flash Patch and Breakpoint unit (FPB)
A set of address matching tags, that reroute accesses into flash to a special part of SRAM. This permits patching flash locations for breakpointing and quick fixes or changes.

Formatter The formatter is an internal input block in the ETB and TPIU that embeds the trace source ID within the data to create a single trace stream.

Halfword A 16-bit data item.

Halt mode One of two mutually exclusive debug modes. In halt 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.

Host A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

ICode Memory Memory space at `0x00000000` to `0x1FFFFFFF`.

Illegal instruction An instruction that is architecturally Undefined.

Implementation-defined
The behavior is not architecturally defined, but is defined and documented by individual implementations.

Implementation-specific
The behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

Instruction cycle count
The number of cycles for which an instruction occupies the Execute stage of the pipeline.

Instrumentation trace
A component for debugging real-time systems through a simple memory-mapped trace interface, providing `printf` style debugging.

Intelligent Energy Management (IEM)
A technology that enables dynamic voltage scaling and clock frequency variation to be used to reduce power consumption in a device.

Internal PPB PPB memory space at `0xE0000000` to `0xE003FFFF`.

Interrupt service routine 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 that contains the first instruction of the corresponding interrupt service routine.

Joint Test Action Group (JTAG)
The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

JTAG *See* Joint Test Action Group.

JTAG Debug Port (JTAG-DP)
An optional external interface for the DAP that provides a standard JTAG interface for debug access.

JTAG-DP *See* JTAG Debug Port.

LE Little endian view of memory in both byte-invariant and word-invariant systems. See also Byte-invariant, Word-invariant.

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.

Load/store architecture
A processor architecture where data-processing operations only operate on register contents, not directly on memory contents.

Load Store Unit (LSU)
The part of a processor that handles load and store transfers.

LSU *See* Load Store Unit.

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 Protection Unit (MPU)
Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not modify addresses.

Microprocessor *See* Processor.

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 Halt mode.

MPU *See* Memory Protection Unit.

Multi-layer An interconnect scheme similar to a cross-bar switch. Each master on the interconnect has a direct link to each slave, The link is not shared with other masters. This enables each master to process transfers in parallel with other masters. Contention only occurs in a multi-layer interconnect at a payload destination, typically the slave.

Penalty The number of cycles in which no useful Execute stage pipeline activity can occur because an instruction flow is different from that assumed or predicted.

Power-on reset *See* Cold reset.

PPB *See* Private Peripheral Bus.

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, Abort.

Private Peripheral Bus

Memory space at `0xE0000000` to `0xE00FFFFF`.

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.

RealView ICE A system for debugging embedded processor cores using a JTAG interface.

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.

SBO *See* Should Be One.

SBZ *See* Should Be Zero.

SBZP *See* Should Be Zero or Preserved.

Scan chain A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

Should Be One (SBO)
Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 produces Unpredictable results.

Should Be Zero (SBZ)
Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces Unpredictable results.

Should Be Zero or Preserved (SBZP)
Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

Serial-Wire Debug Port
An optional external interface for the DAP that provides a serial-wire bidirectional debug interface.

SW-DP *See* Serial-Wire Debug Port.

Synchronization primitive
The memory synchronization primitive instructions are those instructions that are used to ensure memory synchronization. That is, the LDREX and STREX instructions.

System memory Memory space at `0x20000000` to `0xFFFFFFFF`, excluding PPB space at `0xE0000000` to `0xE00FFFFF`.

TAP *See* Test access port.

Test Access Port (TAP)
The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

Thumb instruction A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb (16-bit) halfword aligned instructions is operating in Thumb state.

TPA See *Trace Port Analyzer*.

TPIU *See* Trace Port Interface Unit.

Trace Port Interface Unit (TPIU)
Drains trace data and acts as a bridge between the on-chip trace data and the data stream captured by a TPA.

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.

UNP *See* Unpredictable.

Unpredictable For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.

Warm reset Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

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.

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 buffer A pipeline stage for buffering write data to prevent bus stalls from stalling the processor.