ARM®v8-M Processor Debug

Version 1.0

ARM 100734_0100_0100_en

ARM®v8-M Processor Debug

Release Information

Document History

Issue	Date	Confidentiality	Change
0100	01 September 2016	Non-Confidential	First release

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow ARM’s trademark usage guidelines at *http://www.arm.com/about/trademark-usage-guidelines.php*

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

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.

Unrestricted Access is an ARM internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents
ARM®v8-M Processor Debug

Preface

Chapter 1 Debug for ARM®v8-M

Preface

This preface introduces the *ARM®v8-M Processor Debug* .

It contains the following:

- *About this book* on page 6.
- *Feedback* on page 8.

About this book

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r1p2, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

Using this book

This book is organized into the following chapters:

Chapter 1 Debug for ARM®v8-M

Debugging is a key part of software development and is often considered to be the most time-consuming part of the process. It enables software developers to halt program execution and determine the cause of any problems. Developers often add breakpoint instructions into software, and data or hardware watchpoints to examine the values of program variables or the contents of registers. Debug facilities that are provided by a system are a vital consideration for any developer.

Glossary

The ARM Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes 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.

`monospace`

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>`

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure 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.

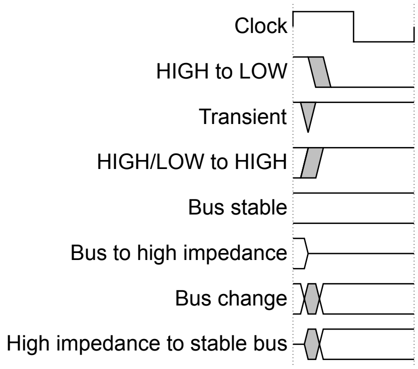

Figure 1 Key to timing diagram conventions

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.
- LOW for active-LOW signals.

Lowercase n

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

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *ARM®v8-M Processor Debug* .
- The number ARM 100734_0100_0100_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Debug for ARM®v8-M

Debugging is a key part of software development and is often considered to be the most time-consuming part of the process. It enables software developers to halt program execution and determine the cause of any problems. Developers often add breakpoint instructions into software, and data or hardware watchpoints to examine the values of program variables or the contents of registers. Debug facilities that are provided by a system are a vital consideration for any developer.

It contains the following sections:

1.1 Debug tools

ARM®v8-M processors provide hardware features that enable debug tools to collect information about core activity and program execution, halt the core, and step through code execution.

You can set software or hardware breakpoints on specific instructions, causing the debugger to take control when the core reaches that instruction.

Software breakpoints work by replacing the instruction with the opcode of the `BKPT` instruction. Software breakpoints can only be used on code that is stored in RAM, but have the advantage that they can be used in large numbers. The debug software tracks where it has placed software breakpoints and what opcodes were originally at those addresses so that it can replace the correct code when executing the breakpoint instruction.

Hardware breakpoints use comparators that are built into the core and stop execution when execution reaches the specified address. Hardware breakpoints can be used anywhere in memory as they do not require changes to code, but the hardware provides limited numbers of hardware breakpoint units.

Debug tools can support more complex breakpoints. For example, stopping on any instruction in a range of addresses, or only when a specific sequence of events occurs or hardware is in a specific state.

Data watchpoints, or breakpoints, give debugger control when a particular data address or address range is read or written. On hitting a breakpoint, or when single-stepping, you can inspect and change the contents of ARM registers and memory. A special case of changing memory is code download. Debug tools typically enable you to change your code, recompile, and then download the new image to the system.

1.2 Types of debug

There are two types of debug available, invasive, and non-invasive.

1.2.1 Invasive debug

Invasive debug is divided into halting debug (using an external debugger) and Monitor debug (using software on the processor). In either case, the debug logic of the core generates a debug event in response to some circumstance, such as a breakpoint being reached. How the system handles that debug event is what distinguishes Monitor debug from halting debug. Invasive debug is not possible if the **DBGEN** signal is tied low.

Halting debug

In halting debug, the debug event causes the core to enter debug state. In debug state, the core is halted, meaning that it no longer fetches instructions. Instead, the core executes instructions under the direction of a debugger running on a different host that is connected through an external interface.

Monitor debug

In Monitor debug, the debug event causes a debug exception to be raised. The exception must be handled by dedicated debug monitor software running on the same core. Monitor debug assumes that there is software support.

1.2.2 Non-invasive debug

Non-invasive debug enables observation of the core behavior while it is executing. It is possible to record memory accesses that are performed (including address and data values) and generate a real-time trace of the program, seeing peripheral accesses, stack, and heap accesses and changes to variables using methods like profiling or ITM. Non-invasive debug is controlled by the NIDEN signal.

Instrumentation Trace Macrocell

The *Instrumentation Trace Macrocell* (ITM) is an optional application-driven trace source that supports `printf()` style debugging to trace operating system and application events, and generates diagnostic system information. The ITM generates trace information as packets from software traces, hardware traces, time stamping, and global system timestamping sources.

Profiling

Profiling is a technique that lets you identify sections of code that consume large proportions of the total execution time. A profiler identifies which parts of the code are frequently executed and which occupy the most core cycles. A profiler can also help you identify bottlenecks, situations where the performance of the system is constrained by a few functions. This data is collected using instrumentation, an execution trace, or sampling.

Profiling can be considered as a form of dynamic code analysis.

There are two basic approaches to gathering information: time-based sampling, and event-based sampling.

Time-based sampling

The state of the system is sampled at a periodic interval. The size of the sampling interval can affect the results. For example, a smaller sampling interval can increase execution time but produce more detailed data.

Event-based sampling

Sampling is driven by occurrences of an event, which means that the time between sampling intervals is variable. Events can often be hardware-related, for example, cache misses.

1.3 Accessing debug features

ARMv8-M processors can contain several programmable debug registers which control the debug features available to the software engineer. Not all these are visible to software. These registers are normally accessed through an external debugger, such as ARM DS-5, and through a *Debug Access Port* (DAP). This is what debug tools, for example, DS-5 and uLINK are used for.

Debug logic can only provide information on the current state of a halted processor, it cannot provide trace information (a history of what has executed just before and just after a particular trigger event).

Debug components like the *Data Watchpoint and Trace* unit (DWT), *Instrumentation Trace Macrocell* (ITM), and *Embedded Trace Module* (ETM), can be used to supply this information.

1.4 Debug support

There are four levels of debug available in ARMv8-M.

The debug implementation levels are:

Table 1-1 Levels of debug

Level	ARMv8-M architecture	ARMv8-M architecture with Main Extension
Minimum	No debug support	Support for the DebugMonitor exception
Basic	Support for halting debug	Support for halting debug
Comprehensive	Not applicable without the Main Extension	Adds basic trace support
Program trace	Adds ETM and TPIU.	Adds ETM

To see which are available for a particular ARMv8-M processor, see the relevant *Technical Reference Manual* (TRM) for that processor.

1.5 Debug authentication

Debug is enabled using one or more of four signals, **DBGEN**, **NIDEN**, **SPIDEN**, and **SPNIDEN**. They can enable debug all, debug Non-secure, or debug none. There are different configurations of these signals for both ARMv8-M processors and development boards.

The debug control signals have the following meanings:

Table 1-2 Debug control signals

Signal Name	Description
DBGEN	Debug Enable
NIDEN	Non-Invasive Debug Enable
SPIDEN	Secure Privileged Invasive Debug Enable
SPNIDEN	Secure Privileged Non-Invasive Debug Enable

The ARMv8-M architecture with Security Extension supports the same external debug control signals as the ARMv8-M architecture with Main Extension with the same rules for enabling invasive, and non-invasive debug being applied. Unlike the ARMv8-M architecture with Main Extension specification, the baseline specification states that it is IMPLEMENTATION DEFINED whether many of the debug registers are accessible from software running on the processor. Unless otherwise stated, the Security Extensions do not change this behavior.

The following table shows how the various debug event sources are handled when invasive debug is either disabled or not permitted in the current state.

Table 1-3 Debug event behavior

Debug event source	Types of behavior based on the invasive debug settings		
	Disabled	Enabled but prohibited by security level	Enabled and allowed
BKPT instruction	HardFault	HardFault	Debug event generated
C_HALT. Internal halt request. Writes to DHCSR.C_HALT are not possible when debug is disabled	See note	Pended	
VC_CORERESET	Ignore		
All other Vector catch		Ignore	
FPB breakpoint. The processor cannot ignore an FPB Breakpoint and it will escalate to HardFault.			
EDBGRQ. External halt request. When EDBGRQ is ignored by the processor, the event remains pending in the system because the signal remains asserted.			
DWT watchpoint			

1.6 Debug registers

Description of the ARMv8-M architecture debug registers.

1.6.1 DCRSR

Secure registers are only accessible when DHCSR.S_SDE is 1. When a register is not accessible, it behaves as RAZ/WI when accessed by a debugger.

The existing REGSEL field is extended to provide direct debugger access to:

- The SP_main, SP_process, CONTROL, FAULTMASK, BASEPRI, and PRIMASK registers for each of:
  - — The current state view.
  - — Non-secure banked register.
  - — Secure banked register, if DHCSR.S_SDE == 1.
- The Secure stack limit registers, if DHCSR.S_SDE == 1.

If the floating-point extension is implemented, then because the FPSCR and S0-S31 registers can contain data belonging to a different state from the one the processor is in. These registers are only accessible to the debugger if invasive debug is enabled for the security state that is associated with the register state (as indicated by the FPCCR register).

1.6.2 DSCSR

Debug security control and status register, address `0xE000EE08`, controls the banking of memory-mapped registers when accessed by the debugger, and allows the debugger to read and modify the current security state of the processor.

Note

This method of bank selection means that the register aliasing is not used for accesses from the debugger. Accesses to the aliased addresses from the debugger have the same behavior as reserved addresses.

1.6.3 DHCSR

A new status bit, DHCSR.S_SDE, is added. This bit indicates whether Secure invasive debug is enabled. It is RES0 if Security Extensions are not implemented.

It normally reflects the state of the external authentication interface, but is frozen while the processor is in Debug state.

Writes to DHCSR.C_SNAPSTALL are ignored if DHCSR.S_SDE == 0.

Interrupts that target Secure state are not masked by DHCSR.C_MASKINTS when DHCSR.S_SDE is 0.

1.6.4 DAUTHCTRL

Debug authentication control, address `0xE000EE04`, resets to `0x0` allows the external Secure Debug Enable authentication interface signals (**SPIDEN** and **SPNIDEN**) to be overridden from software.

This register can only be accessed by software running in the Secure privileged state, unprivileged accesses generate a `BusFault`, and privileged Non-secure accesses behave as RAZ/WI. This register is not accessible from the debugger.

1.6.5 DAUTHSTATUS

Debug authentication status, address `0b0xE000EFB8`, indicates which debug features are present, and status of these features. This register is compatible with the CoreSight™ AUTHSTATUS register.

This register can only be accessed by the debugger or software running in a privileged mode, accesses from the other states generates a `BusFault`.

To aid detection of what debug features are enabled or present this register is present in all ARMv8-M processors, even if they do not implement the Security Extensions. However since processors without Security Extensions only implement the Non-secure state, the Secure debug features are reported as not implemented and disabled.

1.7 Memory accesses

The debugger can be used to access memory locations.

Any accesses to the address space include a new attribute NS-Attr, which is used to mark transactions as Non-secure or secure. The Security Extensions also define a new NS-Req attribute, which defines the security state that the core requests that the data transaction be performed in. Unless otherwise specified NS-Req is equal to the security state of the core.

Debugger accesses to memory, and any memory mapped registers are subject to the same security checks as data accesses generated by the processor, with the transaction attributes set as follows:

- NS-Req is set by the *Debug Access Port* (DAP) if DHCSR.S_SDE is 1.
- Otherwise NS-Req set to Non-secure.

Debugger accesses are subject to validation and attribution, that is, NS-Attr for debugger generated transactions is set by the *Secure Attribution Unit* (SAU) or *Implementation Defined Attribution Unit* (IDAU) in the same way as software generated accesses. However debugger accesses are not subject to any MPU checks.

If the SAU or IDAU blocks memory accesses originating from the debugger that do not have the required permission, an error response is returned to the DAP.

The debugger can request the access be performed as a Non-secure or a Secure access by setting the DAP register CSW.Prot to 1 or 0.

1.8 Other debug functionality

Describes the other types of debug functionality available.

1.8.1 FPB

Where a debugger can create unlimited soft breakpoints if code is in RAM, by simply swapping out a breakpointed instruction and replacing it with a BKPT instruction, the *Flash Patch and Breakpoint* (FPB) unit mimics this for non-volatile memory by returning a BKPT code for the comparator address.

Typically FPB implements hardware breakpoints, and patches code and data from Code space to System space. Flash Patch allows a breakpoint to be set in ROM code which causes a branch to another address. It can be used to replace instructions in ROM after it is masked.

The Flash Patch functionality is not implemented if the Security Extension is implemented.

The debug event that is associated with breakpoints defined by the FPB can be blocked if invasive debug is not enabled for the mode the processor was in when the breakpoint became active.

1.8.2 ETM and PTM

Trace is typically provided by an internal *Embedded Trace Macrocell* (ETM) or *Program Trace Macrocell* (PTM) connected to the core. The ETM and PTM are optional parts of most ARM processor-based systems. The trace blocks do not affect core behavior, but are able to monitor instruction execution and data accesses.

Using the ETM it might also be possible to record memory accesses (including address and data values) and generate a real-time trace of the program, seeing peripheral accesses, stack, and heap accesses and changes to variables.

Trace

For many real-time systems, it is not possible to use invasive debug methods. Consider, for example, an engine management system. While you can stop the core at a particular point, the engine will keep moving and you will not be able to do useful debug.

Any trace data that is generated is only logged if non-invasive debug is enabled for the security and privilege state that the processor was in when the data was generated. For trace information related to exceptions, the trace data must only be logged if non-invasive debug is enabled for the security state that the exception is taken to.

1.8.3 DWT

The *Data Watchpoint and Trace* (DWT) unit is a debug unit that provides watchpoints and system profiling for the processor. Data tracing is also available if the processor has been implemented with DWT and ITM trace. It provides a series of counters which are triggered by processor events.

The DWT unit contains several counters that can count:

- Clock cycles.
- Folded instructions (If-Then and some NOPs).
- LSU operations.
- Sleep cycles.
- Interrupt overhead.
- Cycles Per Instruction (CPI).

The DWT unit also contains a *Program Count Sample Register* (PCSR). The PCSR can be used by a debugger for crude profiling as it contains the instruction address of recently executed instructions.

The DWT unit interfaces to the core, ETM, and ITM.

When non-invasive debug is not enabled for the mode the processor is in, the DWT behaves as follows:

- The performance counters do not increment
- The DWT_PCSR register reads as `0xFFFFFFFF`
- The CMPMATCH[N] events are not generated
- Trace data from Secure accesses are not generated. However, time-based trace might be generated:
  - — Periodic PC samples have the PC value suppressed, but are generated.
  - — Synchronization packets can be generated.
  - — Trace events from a cycle counter match can be generated.

Watchpoint debug events are asynchronous to the instruction that caused the event. A watchpoint debug event might be generated by a Non-secure access, but pended because the processor enters Secure state with invasive debug prohibited before it is taken. Non-secure accesses include stack pushes and lazy state preservation of Non-secure register data.

Watchpoint debug events from a Cycle Counter match can be generated (and pended) when the processor is in Secure state with invasive debug prohibited. Otherwise, watchpoint debug events are never generated in Secure state when invasive debug is prohibited.

Note

Invasive debug prohibited here means either S_SDE == 0 or SDME == 0, depending on whether the processor would halt or generate a DebugMonitor exception for the debug event.

DWT_CTRL

A control bit, DWT_CTRL.CYCDISS, is added to stop the cycle counter from incrementing when executing Secure code. DWT_CTRL resets to 0.