Requirements List of RISC-V External Debug Support

Version 0.13.2[*]

by

Mehmet Öner

Version 1.0

*"RISC-V External Debug Support", Version 0.13.2, d5029366d59e8563c08b6b9435f82573b603e48e Editors: Tim Newsome <tim@sifive.com>, SiFive, Inc., Megan Wachs <megan@sifive.com>, SiFive, Inc., RISC-V International, March 2019, https://github.com/riscv/riscv-debug-spec
This specification is licensed under the Creative Commons Attribution 4.0 International License.
https://creativecommons.org/licenses/by/4.0/

About

The Document

In systems engineering approach, before doing anything with the design of the system under consideration, *requirements analysis* must be completed as one of the first tasks (if not the very first). Beginning with an itemized, atomic, classified and well defined list of requirements is essential. Because, following activities at various stages of development (like design coverage analysis, testing, verification, validation …) depend on the requirement specifications stated at the beginning.

RISC-V International provides the ISA (Instruction Set Architecture) and non-ISA requirement specifications for the RISC-V architecture (https://riscv.org/technical/specifications/). These documents in general are good written technical plain text documents. However, they lack some aspects of good requirement specification practices:

- Requirements are in free text form and not itemized: Itemized list of requirements enables requirement coverage in design, test, verification and validation phases.
- Text includes comments and information statements along with requirements: Statements must be clearly labeled and categorized.
- Some statements include more than one specifications: Each specification need to be isolated.
- Some specifications such as instruction definitions are distributed throughout the text: The distributed content need to be put together to have a complete the specification.

The aim of this document is providing an edited list of requirements for “RISC-V External Debug Support", Version 0.13.2, d5029366d59e8563c08b6b9435f82573b603e48e Editors: Tim Newsome <tim@sifive.com>, SiFive, Inc., Megan Wachs <megan@sifive.com>, SiFive, Inc., RISC-V International, March 2019, https://github.com/riscv/riscv-debug-spec. This specification is licensed under the Creative Commons Attribution 4.0 International License. https://creativecommons.org/licenses/by/4.0/

In particular:

- Statements were itemized and given an ID number.
- Each itemized statement was referenced to the original document to provide traceability
- Itemized statements were categorized
- Complex statements were broken into simpler atomic requirement statements when needed.
- Distributed requirement information was put together to form complete specifications.

Special attention was given to preserve original statements, even when dividing complex statements into simpler atomic statements. But occasionally, some statements were re-written as to form a formal requirement statement.

The Editor (or the systems engineer)

After 34 years of my career, I retired from my regular job in 2023. Now, I do part time consulting services to interested parties, while I do work on projects that interest me more than a regular work.

In my career I dealt with very diverse fields of engineering: Academics, C/C++ desktop programming, embedded systems, analog circuit design, DSP algorithms, VLSI/FPGA design, underwater acoustics are to name few.

I’ve always enjoyed designing controllers and processors with generic HDL for VLSI or FPGA. So, as my personal project to work on, I decided to design RISC-V cores with different capabilities.

Having some defense sector background, I find systems engineering approach very useful. After reading RISC-V specification documents, I decided to take the initiative and edit the documents into customer requirements list format which is a tough, tedious and time consuming work.

In case someone else could find these documents useful, I share them on github:
https://github.com/vizionerco/RISC-V

Best regards,

Mehmet Öner, Ph.D.

www.linkedin.com/in/mehmet-oner-00453733/

Table of Contents

Definitions

Table 1: Requirement Types

TYPE	NAME	EXPLANATION
H	Heading	Headings in the original document. Headings are included to provide context to the subsequent requirements
I	Information	These are statements that explain some aspects of the subject, but actually do not specify any requirement. For these types, the statement(s) in the text is given as is.
C	Comment	These statements are original comment statements in the RISC_V documentation which are explained as: “Commentary on our design decisions is formatted as in this paragraph. This non-normative text can be skipped if the reader is only interested in the specification itself.” For these types, the statement(s) in the text is given as is.
R	Requirement	These are statements that specify a specific need to be fulfilled. For these types, the statement(s) in the text is given as is where possible. In some cases, information is collected from different tables and figures to form a complete specification. In some cases, context is added in parenthesis to make the requirement self explanatory. In some cases, the statements are broken into single statements requirement statements.
O	Optional Requirement	These are statements that specify a property that is not mandatory to implement. However if it is chosen to fulfill, it should obey this requirement. Inclusion of the text is the same as R/Requirement type.
T	Tentative Requirement	These are requirements but are not frozen yet by the RISC-V committee. Inclusion of the text is the same as R/Requirement type.

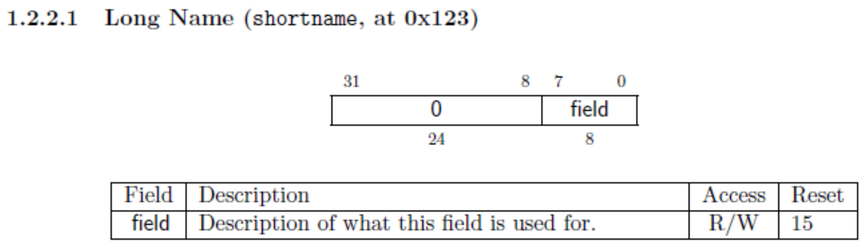
Table 2: Abbreviations & Definitions

SHORT	MEANING
DM	Debug Module
DMI	Debug Module Interface
DTM	Debug Transport Module
DXLEN	DXLEN of a hart is its widest supported XLEN, ignoring the current value of MXL in `misa`.
Hart	A single RISC-V core contains one or more hardware threads, called *harts*.
Platform	A *platform* is a single integrated circuit consisting of one or more *components*. Some components may be RISC-V cores, while others may have a different function. Typically they will all be connected to a single system bus.

CHAPTER 1 Introduction

ID	REFERENCE	TYPE	DEFINITION
RVD.1.1	1.0 (p.1)	H	Introduction
RVD.1.2	1.0 (p.1)	I	When a design progresses from simulation to hardware implementation, a user's control and understanding of the system's current state drops dramatically. To help bring up and debug low level software and hardware, it is critical to have good debugging support built into the hardware. When a robust OS is running on a core, software can handle many debugging tasks. However, in many scenarios, hardware support is essential.
RVD.1.3	1.0 (p.1)	I	This document outlines a standard architecture for external debug support on RISC-V platforms. This architecture allows a variety of implementations and tradeoffs, which is complementary to the wide range of RISC-V implementations. At the same time, this specification defines common interfaces to allow debugging tools and components to target a variety of platforms based on the RISC-V ISA.
RVD.1.4	1.0 (p.1)	O	System designers may choose to add additional hardware debug support, ...
RVD.1.5	1.0 (p.1)	I	… but this specification defines a standard interface for common functionality.
RVD.1.6	1.1 (p.1)	H	Terminology
RVD.1.7	1.1 (p.1)	I	A *platform* is a single integrated circuit consisting of one or more *components*. Some components may be RISC-V cores, while others may have a different function. Typically they will all be connected to a single system bus. A single RISC-V core contains one or more hardware threads, called *harts*.
RVD.1.8	1.1 (p.1)	I	*DXLEN* of a hart is its widest supported XLEN, ignoring the current value of MXL in `misa`.
RVD.1.9	1.1.1 (p.1)	H	Context
RVD.1.10	1.1.1 (p.1, p.2)	I	This document is written to work with: 1. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2 (the ISA Spec) 2. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10 (the Privileged Spec)
RVD.1.11	1.1.2 (p.2)	H	Versions
RVD.1.12	1.1.2 (p.2)	I	Version 0.13 of this document was ratified by the RISC-V Foundation's board.
RVD.1.13	1.1.2 (p.2)	I	Versions 0.13.x are bug fix releases to that ratified specification.
RVD.1.14	1.1.2 (p.2)	I	Version 0.14 will be forwards and backwards compatible with Version 0.13.
RVD.1.15	1.2 (p.2)	H	About This Document
RVD.1.16	1.2.1 (p.2)	H	Structure
RVD.1.17	1.2.1 (p.2)	I	This document contains two parts.
RVD.1.18	1.2.1 (p.2)	I	The main part of the document is the specification, which is given in the numbered sections.

ID	REFERENCE	TYPE	DEFINITION
RVD.1.19	1.2.1 (p.2)	I	The second part of the document is a set of appendices. The information in the appendices is intended to clarify and provide examples, but is not part of the actual specification.
RVD.1.20	1.2.2 (p.2)	H	Register Definition Format
RVD.1.21	1.2.2 (p.2)	I	All register definitions in this document follow the format shown below.



1.2.2.1 Long Name (shortname, at 0x123)

Field	Description	Access	Reset
field	Description of what this field is used for.	R/W	15

ID	REFERENCE	TYPE	DEFINITION
RVD.1.22	1.2.2 (p.2)	I	A simple graphic shows which fields are in the register. The upper and lower bit indices are shown to the top left and top right of each field. The total number of bits in the field are shown below it.
RVD.1.23	1.2.2 (p.2)	I	After the graphic follows a table which for each field lists its name, description, allowed accesses, and reset value.
RVD.1.24	1.2.2 (p.2) Table 1.2	I	The allowed accesses are listed in table below. Register Access Abbreviations

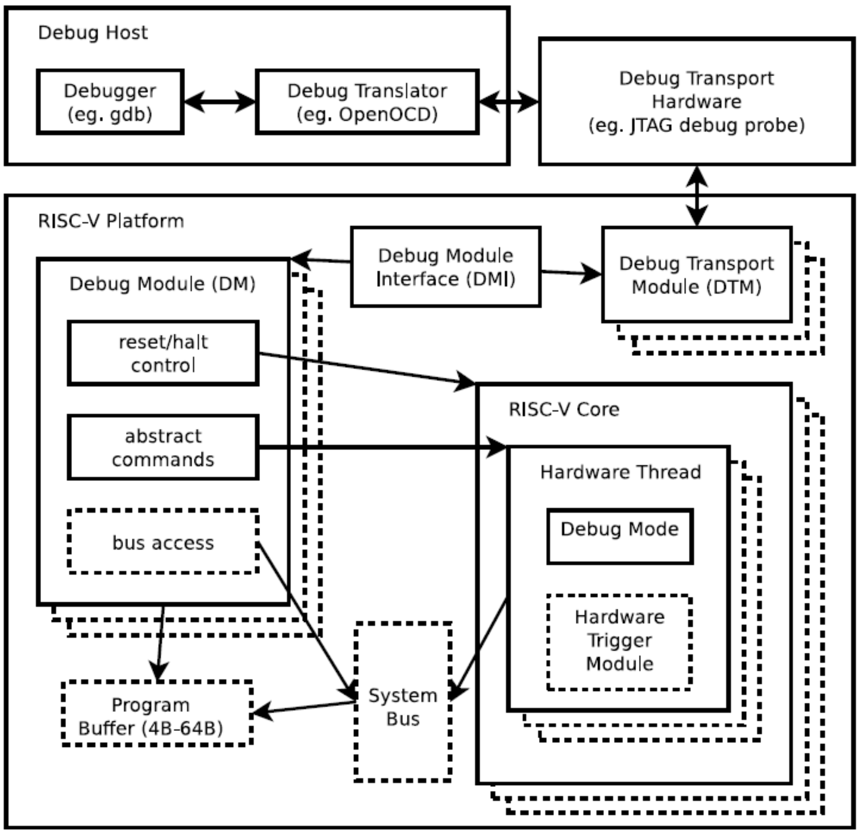
R	Read-only.
R/W	Read/Write.
R/W1C	Read/Write. For each bit in the field, writing 1 clears that bit. Writing 0 has no effect.
W	Write-only. When read this field returns 0.
W1	Write-only. Only writing 1 has an effect.
WARL	Write any, read legal. A debugger may write any value. If a value is unsupported, the implementation converts the value to one that is supported.

ID	REFERENCE	TYPE	DEFINITION
RVD.1.25	1.2.2 (p.2)	I	The reset value is either a constant or “Preset.” The latter means it is an implementation-specific legal value.
RVD.1.26	1.2.2 (p.2)	I	Names of registers and their fields are hyperlinks to their definition, and are also listed in the index on page 82. (hyperlink register names and index pages are not supported in this document)
RVD.1.27	1.3 (p.3)	H	Background
RVD.1.28	1.3 (p.3)	I	There are several use cases for dedicated debugging hardware, both internal to a CPU core and with an external connection.
RVD.1.29	1.3 (p.3)	I	This specification addresses the use cases listed below. • Debugging low-level software in the absence of an OS or other software. • Debugging issues in the OS itself. • Bootstrapping a system to test, configure, and program components before there is any executable code path in the system. • Accessing hardware on a system without a working CPU.
RVD.1.30	1.3 (p.3)	I	Implementations can choose not to implement every feature, which means some use cases might not be supported.

ID	REFERENCE	TYPE	DEFINITION
RVD.1.31	1.3 (p.3)	I	In addition, even without a hardware debugging interface, architectural support in a RISC-V CPU can aid software debugging and performance analysis by allowing hardware triggers and breakpoints.
RVD.1.32	1.4 (p.3)	H	Supported Features
RVD.1.33	1.4 (p.3, p.4)	I	The debug interface described in this specification supports the following features: 1. All hart registers (including CSRs) can be read/written. 2. Memory can be accessed either from the hart's point of view, through the system bus directly, or both. 3. RV32, RV64, and future RV128 are all supported. 4. Any hart in the platform can be independently debugged. 5. A debugger can discover almost[†] everything it needs to know itself, without user configuration. 6. Each hart can be debugged from the very first instruction executed. 7. A RISC-V hart can be halted when a software breakpoint instruction is executed. 8. Hardware single-step can execute one instruction at a time. 9. Debug functionality is independent of the debug transport used. 10. The debugger does not need to know anything about the microarchitecture of the harts it is debugging. 11. Arbitrary subsets of harts can be halted and resumed simultaneously. (Optional) 12. Arbitrary instructions can be executed on a halted hart. That means no new debug functionality is needed when a core has additional or custom instructions or state, as long as there exist programs that can move that state into GPRs. (Optional) 13. Registers can be accessed without halting. (Optional) 14. A running hart can be directed to execute a short sequence of instructions, with little overhead. (Optional) 15. A system bus master allows memory access without involving any hart. (Optional) 16. A RISC-V hart can be halted when a trigger matches the PC, read/write address/data, or an instruction opcode. (Optional)
RVD.1.34	1.4 (p.3, p.4)	I	This document does not suggest a strategy or implementation for hardware test, debugging or error detection techniques.
RVD.1.35	1.4 (p.3, p.4)	I	Scan, BIST, etc. are out of scope of this specification, but this specification does not intend to limit their use in RISC-V systems.
RVD.1.36	1.4 (p.3, p.4)	I	It is possible to debug code that uses software threads, but there is no special debug support for it.

† Notable exceptions include information about the memory map and peripherals.

CHAPTER 2 System Overview

ID	REFERENCE	TYPE	DEFINITION
RVD.2.1	2.0 (p.5)	H	System Overview
RVD.2.2	2.0 (p.5) Figure 2.1	R	The figure below shows the main components of External Debug Support. RISC-V Debug System Overview  Debug Host Debugger (eg. gdb) Debug Translator (eg. OpenOCD) Debug Transport Hardware (eg. JTAG debug probe) RISC-V Platform Debug Module (DM) reset/halt control abstract commands bus access Debug Module Interface (DMI) Debug Transport Module (DTM) RISC-V Core Hardware Thread Debug Mode Hardware Trigger Module Program Buffer (4B-64B) System Bus
RVD.2.3	2.0 (p.5)	O	Blocks shown in dotted lines are optional.
RVD.2.4	2.0 (p.5)	R	The user interacts with the Debug Host (e.g. laptop), which is running a debugger (e.g. gdb).
RVD.2.5	2.0 (p.5)	R	The debugger communicates with a Debug Translator (e.g. OpenOCD, which may include a hardware driver) to communicate with Debug Transport Hardware (e.g. Olimex USB-JTAG adapter).
RVD.2.6	2.0 (p.5)	R	The Debug Transport Hardware connects the Debug Host to the Platform's Debug Transport Module (DTM).
RVD.2.7	2.0 (p.5)	R	The DTM provides access to one or more Debug Modules (DMs) using the Debug Module Interface (DMI).
RVD.2.8	2.0 (p.5)	R	Each hart in the platform is controlled by exactly one DM.
RVD.2.9	2.0 (p.5)	O	Harts may be heterogeneous.
RVD.2.10	2.0 (p.5)	R	There is no further limit on the hart-DM mapping, but usually all harts in a single core are controlled by the same DM.
RVD.2.11	2.0 (p.5)	R	In most platforms there will only be one DM that controls all the harts in the platform.

ID	REFERENCE	TYPE	DEFINITION
RVD.2.12	2.0 (p.5)	R	DMs provide run control of their harts in the platform.
RVD.2.13	2.0 (p.5)	R	Abstract commands provide access to GPRs.
RVD.2.14	2.0 (p.5)	R	Additional registers are accessible through abstract commands or by writing programs to the optional Program Buffer.
RVD.2.15	2.0 (p.5)	R	The Program Buffer allows the debugger to execute arbitrary instructions on a hart.
RVD.2.16	2.0 (p.5)	R	This mechanism can also be used to access memory.
RVD.2.17	2.0 (p.5)	O	An optional system bus access block allows memory accesses without using a RISC-V hart to perform the access.
RVD.2.18	2.0 (p.5)	R	Each RISC-V hart may implement a Trigger Module.
RVD.2.19	2.0 (p.5)	R	When trigger conditions are met, harts will halt and inform the debug module that they have halted.
RVD.2.20			
RVD.2.21			
RVD.2.22	2.0 (p.18) Figure 7 Figure 8	R	The partitioning of the **device_id** to obtain the device directory indexes (DDI) to traverse the DDT radix-tree are as follows: Base format **device_id** partitioning

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DDI[2]								DDI[1]									DDI[0]						

Extended format **device_id** partitioning

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DDI[2]									DDI[1]									DDI[0]					

CHAPTER 3 Debug Module (DM)

ID	REFERENCE	TYPE	DEFINITION
RVD.3.1	3.0 (p.7)	H	Debug Module (DM)
RVD.3.2	3.0 (p.7)	I	The Debug Module implements a translation interface between abstract debug operations and their specific implementation.
RVD.3.3	3.0 (p.7)	I	It might support the following operations: (items 1-11)
RVD.3.4	3.0 (p.7)	R	1. Give the debugger necessary information about the implementation. (Required)
RVD.3.5	3.0 (p.7)	R	2. Allow any individual hart to be halted and resumed. (Required)
RVD.3.6	3.0 (p.7)	R	3. Provide status on which harts are halted. (Required)
RVD.3.7	3.0 (p.7)	R	4. Provide abstract read and write access to a halted hart's GPRs. (Required)
RVD.3.8	3.0 (p.7)	R	5. Provide access to a reset signal that allows debugging from the very rst instruction after reset. (Required)
RVD.3.9	3.0 (p.7)	O	6. Provide a mechanism to allow debugging harts immediately out of reset (regardless of the reset cause). (Optional)
RVD.3.10	3.0 (p.7)	O	7. Provide abstract access to non-GPR hart registers. (Optional)
RVD.3.11	3.0 (p.7)	O	8. Provide a Program Buffer to force the hart to execute arbitrary instructions. (Optional)
RVD.3.12	3.0 (p.7)	O	9. Allow multiple harts to be halted, resumed, and/or reset at the same time. (Optional)
RVD.3.13	3.0 (p.7)	O	10. Allow memory access from a hart's point of view. (Optional)
RVD.3.14	3.0 (p.7)	O	11. Allow direct System Bus Access. (Optional)
RVD.3.15	3.0 (p.7)	R	In order to be compliant with this specification an implementation must: 1. Implement all the required features listed above. 2. Implement at least one of Program Buffer, System Bus Access, or Abstract Access Memory command mechanisms. 3. Do at least one of: (a) Implement the Program Buffer. (b) Implement abstract access to all registers that are visible to software running on the hart including all the registers that are present on the hart and listed in Table 3.3. (c) Implement abstract access to at least all GPRs, `dcsr`, and `dpc`, and advertise the implementation as conforming to the "Minimal RISC-V Debug Specification 0.13.2", instead of the "RISC-V Debug Specification 0.13.2".
RVD.3.16	3.0 (p.7)	R	A single DM can debug up to 2^{20} harts.
RVD.3.17	3.1 (p.8)	H	Debug Module Interface (DMI)
RVD.3.18	3.1 (p.8)	R	Debug Modules are slaves to a bus called the Debug Module Interface (DMI).
RVD.3.19	3.1 (p.8)	R	The master of the bus is the Debug Transport Module(s).
RVD.3.20	3.1 (p.8)	O	The Debug Module Interface can be a trivial bus with one master and one slave, or use a more full-featured bus like TileLink or the AMBA Advanced Peripheral Bus. The details are left to the system designer.

ID	**REFERENCE**	**TYPE**	**DEFINITION**
RVD.3.21	3.1 (p.8)	R	The DMI uses between 7 and 32 address bits.
RVD.3.22	3.1 (p.8)	R	It supports read and write operations.
RVD.3.23	3.1 (p.8)	R	The bottom of the address space is used for the first (and usually only) DM.
RVD.3.24	3.1 (p.8)	R	Extra space can be used for custom debug devices, other cores, additional DMs, etc.
RVD.3.25	3.1 (p.8)	R	If there are additional DMs on this DMI, the base address of the next DM in the DMI address space is given in `nextdm`.
RVD.3.26	3.1 (p.8)	R	The Debug Module is controlled via register accesses to its DMI address space.
RVD.3.27	3.2 (p.8)	H	Reset Control
RVD.3.28	3.2 (p.8)	R	The Debug Module controls a global reset signal, ndmreset (non-debug module reset), which can reset, or hold in reset, every component in the platform, except for the Debug Module and Debug Transport Modules.
RVD.3.29	3.2 (p.8)	O	Exactly what is affected by this reset is implementation dependent, as long as it is possible to debug programs from the first instruction executed.
RVD.3.30	3.2 (p.8)	R	The Debug Module's own state and registers should only be reset at power-up and while dmactive in `dmcontrol` is 0.
RVD.3.31	3.2 (p.8)	R	The halt state of harts should be maintained across system reset provided that dmactive is 1, although trigger CSRs may be cleared.
RVD.3.32	3.2 (p.8)	I	Due to clock and power domain crossing issues, it may not be possible to perform arbitrary DMI accesses across system reset.
RVD.3.33	3.2 (p.8)	R	While ndmreset or any external reset is asserted, the only supported DM operation is accessing `dmcontrol`.
RVD.3.34	3.2 (p.8)	R	(during reset) The behavior of other accesses is undefined.
RVD.3.35	3.2 (p.8)	I	There is no requirement on the duration of the assertion of ndmreset.
RVD.3.36	3.2 (p.8)	R	The implementation must ensure that a write of ndmreset to 1 followed by a write of ndmreset to 0 triggers system reset.
RVD.3.37	3.2 (p.8)	R	The system may take an arbitrarily long time to come out of reset, as reported by allunavail, anyunavail.
RVD.3.38	3.2 (p.8)	R	Individual harts (or several at once) can be reset by selecting them, setting and then clearing hartreset.
RVD.3.39	3.2 (p.8)	O	In this case an implementation may reset more harts than just the ones that are selected.
RVD.3.40	3.2 (p.8)	O	The debugger can discover which other harts are reset (if any) by selecting them and checking anyhavereset and allhavereset.
RVD.3.41	3.2 (p.8)	R	When harts have been reset, they must set a sticky havereset state bit.
RVD.3.42	3.2 (p.8)	R	The conceptual havereset state bits can be read for selected harts in anyhavereset and allhavereset in `dmstatus`.
RVD.3.43	3.2 (p.8)	R	These bits must be set regardless of the cause of the reset.
RVD.3.44	3.2 (p.8)	R	The havereset bits for the selected harts can be cleared by writing 1 to ackhavereset in `dmcontrol`.
RVD.3.45	3.2 (p.8)	O	The havereset bits may or may not be cleared when dmactive is low.

ID	**REFERENCE**	**TYPE**	**DEFINITION**
RVD.3.46	3.2 (p.8)	R	When a hart comes out of reset and haltreq or resethaltreq are set, the hart will immediately enter Debug Mode. Otherwise it will execute normally.
RVD.3.47	3.3 (p.9)	H	Selecting Harts
RVD.3.48	3.3 (p.9)	R	Up to 2^{20} harts can be connected to a single DM.
RVD.3.49	3.3 (p.9)	R	The debugger selects a hart, and then subsequent halt, resume, reset, and debugging commands are specific to that hart.
RVD.3.50	3.3 (p.9)	R	To enumerate all the harts, a debugger must first determine HARTSELLEN by writing all ones to hartsel (assuming the maximum size) and reading back the value to see which bits were actually set.
RVD.3.51	3.3 (p.9)	R	Then it selects each hart starting from 0 until either anynonexistent in `dmstatus` is 1, or the highest index (depending on HARTSELLEN) is reached.
RVD.3.52	3.3 (p.9)	R	The debugger can discover the mapping between hart indices and `mhartid` by using the interface to read `mhartid`, or by reading the system's configuration string.
RVD.3.53	3.3.1 (p.9)	H	Selecting a Single Hart
RVD.3.54	3.3.1 (p.9)	R	All debug modules must support selecting a single hart.
RVD.3.55	3.3.1 (p.9)	R	The debugger can select a hart by writing its index to hartsel.
RVD.3.56	3.3.1 (p.9)	R	Hart indexes start at 0 and are contiguous until the final index.
RVD.3.57	3.3.2 (p.9)	H	Selecting Multiple Harts
RVD.3.58	3.3.2 (p.9)	R	Debug Modules may implement a Hart Array Mask register to allow selecting multiple harts at once.
RVD.3.59	3.3.2 (p.9)	R	The *n*th bit in the Hart Array Mask register applies to the hart with index *n*. If the bit is 1 then the hart is selected.
RVD.3.60	3.3.2 (p.9)	R	Usually a DM will have a Hart Array Mask register exactly wide enough to select all the harts it supports, but it's allowed to tie any of these bits to 0.
RVD.3.61	3.3.2 (p.9)	R	The debugger can set bits in the hart array mask register using `hawindowsel` and `hawindow`, then apply actions to all selected harts by setting hasel.
RVD.3.62	3.3.2 (p.9)	R	If this feature is supported, multiple harts can be halted, resumed, and reset simultaneously.
RVD.3.63	3.3.2 (p.9)	R	The state of the hart array mask register is not affected by setting or clearing hasel.
RVD.3.64	3.3.2 (p.9)	R	Only the actions initiated by `dmcontrol` can apply to multiple harts at once, Abstract Commands apply only to the hart selected by hartsel.
RVD.3.65	3.4 (p.9)	H	Hart States
RVD.3.66	3.4 (p.9)	R	Every hart that can be selected is in exactly one of four states.
RVD.3.67	3.4 (p.9)	R	Which state the selected harts are in is reflected by allnonexistent, anynonexistent, allunavail, anyunavail, allrunning, anyrunning, allhalted, and anyhalted.
RVD.3.68	3.4 (p.9)	R	Harts are nonexistent if they will never be part of this system, no matter how long a user waits. E.g. in a simple single-hart system only one hart exists, and all others are nonexistent. Debuggers may assume that a system has no harts with indexes higher than the first nonexistent one.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.69	3.4 (p.9)	R	Harts are unavailable if they might exist/become available at a later time, or if there are other harts with higher indexes than this one.
RVD.3.70	3.4 (p.9)	I	Harts may be unavailable for a variety of reasons including being reset, temporarily powered down, and not being plugged into the system.
RVD.3.71	3.4 (p.10)	R	Systems with very large number of harts may permanently disable some during manufacturing, leaving holes in the otherwise continuous hart index space. In order to let the debugger discover all harts, they must show up as unavailable even if there is no chance of them ever becoming available.
RVD.3.72	3.4 (p.10)	R	Harts are running when they are executing normally, as if no debugger was attached. This includes being in a low power mode or waiting for an interrupt, as long as a halt request will result in the hart being halted.
RVD.3.73	3.4 (p.10)	R	Harts are halted when they are in Debug Mode, only performing tasks on behalf of the debugger.
RVD.3.74	3.4 (p.10)	I	Which states a hart that is reset goes through is implementation dependent.
RVD.3.75	3.4 (p.10)	O	Harts may be unavailable while reset is asserted, and some time after reset is deasserted.
RVD.3.76	3.4 (p.10)	O	They might transition to running for some time after reset is deasserted.
RVD.3.77	3.4 (p.10)	R	Finally they end up either running or halted, depending on haltreq and resethaltreq.
RVD.3.78	3.5 (p.10)	H	Run Control
RVD.3.79	3.5 (p.10)	R	For every hart, the Debug Module tracks 4 conceptual bits of state: halt request, resume ack, halt-on-reset request, and hart reset.
RVD.3.80	3.5 (p.10)	O	(The hart reset and halt-on-reset request bits are optional.)
RVD.3.81	3.5 (p.10)	R	These 4 bits reset to 0, except for resume ack, which may reset to either 0 or 1.
RVD.3.82	3.5 (p.10)	R	The DM receives halted, running, and havereset signals from each hart.
RVD.3.83	3.5 (p.10)	R	The debugger can observe the state of resume ack in allresumeack and anyresumeack, and the state of halted, running, and havereset signals in allhalted, anyhalted, allrunning, anyrunning, allhavereset, and anyhavereset.
RVD.3.84	3.5 (p.10)	R	The state of the other bits cannot be observed directly.
RVD.3.85	3.5 (p.10)	R	When a debugger writes 1 to haltreq, each selected hart's halt request bit is set.
RVD.3.86	3.5 (p.10)	R	When a running hart, or a hart just coming out of reset, sees its halt request bit high, it responds by halting, deasserting its running signal, and asserting its halted signal.
RVD.3.87	3.5 (p.10)	R	Halted harts ignore their halt request bit.
RVD.3.88	3.5 (p.10)	R	When a debugger writes 1 to resumereq, each selected hart's resume ack bit is cleared and each selected, halted hart is sent a resume request.
RVD.3.89	3.5 (p.10)	R	Harts respond by resuming, clearing their halted signal, and asserting their running signal. At the end of this process the resume ack bit is set.
RVD.3.90	3.5 (p.10)	R	These status signals of all selected harts are reflected in allresumeack, anyresumeack, allrunning, and anyrunning.
RVD.3.91	3.5 (p.10)	R	Resume requests are ignored by running harts.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.92	3.5 (p.10)	R	When halt or resume is requested, a hart must respond in less than one second, unless it is unavailable.
RVD.3.93	3.5 (p.10)	C	(How this is implemented is not further specified. A few clock cycles will be a more typical latency).
RVD.3.94	3.5 (p.10)	O	The DM can implement optional halt-on-reset bits for each hart, which it indicates by setting hasresethaltreq to 1.
RVD.3.95	3.5 (p.10)	R	This means the DM implements the setresethaltreq and clrresethaltreq bits.
RVD.3.96	3.5 (p.10)	R	Writing 1 to setresethaltreq sets the halt-on-reset request bit for each selected hart.
RVD.3.97	3.5 (p.10)	R	When a hart's halt-on-reset request bit is set, the hart will immediately enter debug mode on the next deassertion of its reset.
RVD.3.98	3.5 (p.10)	R	This is true regardless of the reset's cause.
RVD.3.99	3.5 (p.10)	R	The hart's halt-on-reset request bit remains set until cleared by the debugger writing 1 to clrresethaltreq while the hart is selected, or by DM reset.
RVD.3.100	3.6 (p.11)	H	Abstract Commands
RVD.3.101	3.6 (p.11)	I	The DM supports a set of abstract commands, most of which are optional.
RVD.3.102	3.6 (p.11)	O	Depending on the implementation, the debugger may be able to perform some abstract commands even when the selected hart is not halted.
RVD.3.103	3.6 (p.11)	R	Debuggers can only determine which abstract commands are supported by a given hart in a given state by attempting them and then looking at cmderr in `abstractcs` to see if they were successful.
RVD.3.104	3.6 (p.11)	O	Commands may be supported with some options set, but not with other options set.
RVD.3.105	3.6 (p.11)	R	If a command has unsupported options set, the DM must set cmderr to 2 (not supported).
RVD.3.106	3.6 (p.11)	C	Example: Every system must support the Access Register command, but may not support accessing CSRs. If the debugger requests to read a CSR in that case, the command will return “notsupported."
RVD.3.107	3.6 (p.11)	R	Debuggers execute abstract commands by writing them to `command`.
RVD.3.108	3.6 (p.11)	R	They can determine whether an abstract command is complete by reading busy in `abstractcs`.
RVD.3.109	3.6 (p.11)	R	After completion, cmderr indicates whether the command was successful or not.
RVD.3.110	3.6 (p.11)	R	Commands may fail because a hart is not halted, not running, unavailable, or because they encounter an error during execution.
RVD.3.111	3.6 (p.11)	R	If the command takes arguments, the debugger must write them to the data registers before writing to `command`.
RVD.3.112	3.6 (p.11)	R	If a command returns results, the Debug Module must ensure they are placed in the data registers before busy is cleared.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.113	3.6 (p.11) Table 3.1	R	Which data registers are used for the arguments is described in the table below. Use of Data Registers.

Argument Width	arg0/return value	arg1	arg2
32	`data0`	`data1`	`data2`
64	`data0, data1`	`data2, data3`	`data4, data5`
128	`data0 - data3`	`data4 - data7`	`data8 - data11`

ID	REFERENCE	TYPE	DEFINITION
RVD.3.114	3.6 (p.11)	R	In all cases the least-significant word is placed in the lowest-numbered data register.
RVD.3.115	3.6 (p.11)	R	The argument width depends on the command being executed, and is DXLEN where not explicitly specified.
RVD.3.116	3.6 (p.11)	C	The Abstract Command interface is designed to allow a debugger to write commands as fast as possible, and then later check whether they completed without error. In the common case the debugger will be much slower than the target and commands succeed, which allows for maximum throughput. If there is a failure, the interface ensures that no commands execute after the failing one. To discover which command failed, the debugger has to look at the state of the DM (e.g. contents of `data0`) or hart (e.g. contents of a register modified by a Program Buffer program) to determine which one failed.
RVD.3.117	3.6 (p.11)	R	Before starting an abstract command, a debugger must ensure that haltreq, resumereq, and ackhavereset are all 0.
RVD.3.118	3.6 (p.11)	R	While an abstract command is executing (busy in `abstractcs` is high), a debugger must not change hartsel, and must not write 1 to haltreq, resumereq, ackhavereset, setresethaltreq, or clrresethaltreq.
RVD.3.119	3.6 (p.11)	R	If an abstract command does not complete in the expected time and appears to be hung, the following procedure can be attempted to abort the command: First the debugger resets the hart (using hartreset or ndmreset), and then it resets the Debug Module (using dmactive).
RVD.3.120	3.6 (p.12)	R	If an abstract command is started while the selected hart is unavailable or if a hart becomes unavailable while executing an abstract command, then the Debug Module may terminate the abstract command, setting busy low, and cmderr to 4 (halt/resume). Alternatively, the command could just appear to be hung (busy never goes low).
RVD.3.121	3.6.1 (p.12)	H	Abstract Command Listing
RVD.3.122	3.6.1 (p.12)	I	This section describes each of the different abstract commands and how their fields should be interpreted when they are written to `command`.
RVD.3.123	3.6.1 (p.12)	R	Each abstract command is a 32-bit value.
RVD.3.124	3.6.1 (p.12)	R	The top 8 bits contain cmdtype which determines the kind of command.
RVD.3.125	3.6.1 (p.12) Table 3.2	R	The table below lists all commands. Meaning of cmdtype

cmdtype	Command
0	Access Register Command
1	Quick Access
2	Access Memory Command

ID	REFERENCE	TYPE	DEFINITION
RVD.3.126	3.6.1.1 (p.12)	H	Access Register
RVD.3.127	3.6.1.1 (p.12)	I	This command gives the debugger access to CPU registers and allows it to execute the Program Buffer.

ID	**REFERENCE**	**TYPE**	**DEFINITION**
RVD.3.128	3.6.1.1 (p.12)	R	It performs the following sequence of operations: 1. If write is clear and transfer is set, then copy data from the register specified by regno into the `arg0` region of `data`, and perform any side effects that occur when this register is read from M-mode. 2. If write is set and transfer is set, then copy data from the `arg0` region of `data` into the register specified by regno, and perform any side effects that occur when this register is written from M-mode. 3. If aarpostincrement is set, increment regno. 4. Execute the Program Buffer, if postexec is set.
RVD.3.129	3.6.1.1 (p.12)	R	If any of these operations fail, cmderr is set and none of the remaining steps are executed.
RVD.3.130	3.6.1.1 (p.12)	R	An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure.
RVD.3.131	3.6.1.1 (p.12)	R	If the failure is that the requested register does not exist in the hart, cmderr must be set to 3 (exception).
RVD.3.132	3.6.1.1 (p.12)	R	Debug Modules must implement this command and must support read and write access to all GPRs when the selected hart is halted.
RVD.3.133	3.6.1.1 (p.12)	R	Debug Modules may optionally support accessing other registers, or accessing registers when the hart is running.
RVD.3.134	3.6.1.1 (p.12)	R	Each individual register (aside from GPRs) may be supported differently across read, write, and halt status.
RVD.3.135	3.6.1.1 (p.12)	C	The encoding of aarsize was chosen to match sbaccess in `sbcs`.
RVD.3.136	3.6.1.1 (p.12)	R	This command modifies `arg0` only when a register is read.
RVD.3.137	3.6.1.1 (p.12)	R	The other data registers are not changed.
RVD.3.138	3.6.1.1 (p.13) Table 3.3	R	Abstract Register Numbers

`0x0000 - 0x0fff`	CSRs. The “PC" can be accessed here through `dpc`.
`0x1000 - 0x101f`	GPRs
`0x1020 - 0x103f`	Floating point registers
`0xc000 - 0xffff`	Reserved for non-standard extensions and internal use.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.139	3.6.1.1 (p.13)	R	(see register layout below)

31 … 24	23	22 21 20	19	18	17	16	15 … 0
cmdtype	0	aarsize	aarpostincrement	postexec	transfer	write	regno

Field	Description
cmdtype	This is 0 to indicate Access Register Command.
aarsize	2: Access the lowest 32 bits of the register. 3: Access the lowest 64 bits of the register. 4: Access the lowest 128 bits of the register. If aarsize specifies a size larger than the register's actual size, then the access must fail. If a register is accessible, then reads of aarsize less than or equal to the register's actual size must be supported. This field controls the Argument Width as referenced in Table 3.1 (RVD.3.113).
aarpostincrement	0: No effect. This variant must be supported. 1: After a successful register access, regno is incremented (wrapping around to 0). Supporting this variant is optional.
postexec	0: No effect. This variant must be supported, and is the only supported one if progbufsize is 0. 1: Execute the program in the Program Buffer exactly once after performing the transfer, if any. Supporting this variant is optional.
transfer	0: No effect. This variant must be supported, and is the only supported one if progbufsize is 0. 1: Execute the program in the Program Buffer exactly once after performing the transfer, if any. Supporting this variant is optional.
write	When transfer is set: 0: Copy data from the specified register into arg0 portion of data. 1: Copy data from arg0 portion of data into the specified register.
regno	Number of the register to access, as described in Table 3.3. `dpc` may be used as an alias for PC if this command is supported on a non-halted hart.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.140	3.6.1.2 (p.14)	H	Quick Access
RVD.3.141	3.6.1.2 (p.14)	R	Perform the following sequence of operations: 1. If the hart is halted, the command sets cmderr to “halt/resume" and does not continue. 2. Halt the hart. If the hart halts for some other reason (e.g. breakpoint), the command sets cmderr to “halt/resume" and does not continue. 3. Execute the Program Buffer. If an exception occurs, cmderr is set to “exception" and the program buffer execution ends, but the quick access command continues. 4. Resume the hart.
RVD.3.142	3.6.1.2 (p.14)	O	Implementing this command is optional.
RVD.3.143	3.6.1.2 (p.14)	I	This command does not touch the `data` registers.
RVD.3.144	3.6.1.2 (p.14)	R	(see register layout below)

31 … 24	23 … 0
cmdtype	0

Field	Description
cmdtype	This is 1 to indicate Quick Access command.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.145	3.6.1.3 (p.14)	H	Access Memory

ID	REFERENCE	TYPE	DEFINITION
RVD.3.146	3.6.1.3 (p.14)	I	This command lets the debugger perform memory accesses, with the exact same memory view and permissions as the selected hart has. This includes access to hart-local memory-mapped registers, etc.
RVD.3.147	3.6.1.3 (p.14)	R	The command performs the following sequence of operations: 1. Copy data from the memory location specified in `arg1` into the `arg0` portion of data, if write is clear. 2. Copy data from the `arg0` portion of data into the memory location specified in `arg1`, if write is set. 3. If aampostincrement is set, increment `arg1`.
RVD.3.148	3.6.1.3 (p.14)	R	If any of these operations fail, cmderr is set and none of the remaining steps are executed.
RVD.3.149	3.6.1.3 (p.14)	R	An access may only fail if the hart, running M-mode code, might encounter that same failure when it attempts the same access.
RVD.3.150	3.6.1.3 (p.14)	R	An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure.
RVD.3.151	3.6.1.3 (p.14)	O	Debug Modules may optionally implement this command and may support read and write access to memory locations when the selected hart is running or halted.
RVD.3.152	3.6.1.3 (p.14)	R	If this command supports memory accesses while the hart is running, it must also support memory accesses while the hart is halted.
RVD.3.153	3.6.1.3 (p.15)	C	The encoding of aamsize was chosen to match sbaccess in `sbcs`.
RVD.3.154	3.6.1.3 (p.15)	R	This command modifies `arg0` only when memory is read.
RVD.3.155	3.6.1.3 (p.15)	R	It modifies `arg1` only if aampostincrement is set.
RVD.3.156	3.6.1.3 (p.15)	R	The other data registers are not changed.
RVD.3.157	3.6.1.3 (p.15)	R	

31 ... 24	23	22 21 20	19	18 17	16	15 14	13 ... 0
cmdtype	aamvirtual	aamsize	aampostincrement	0	write	target-specific	0

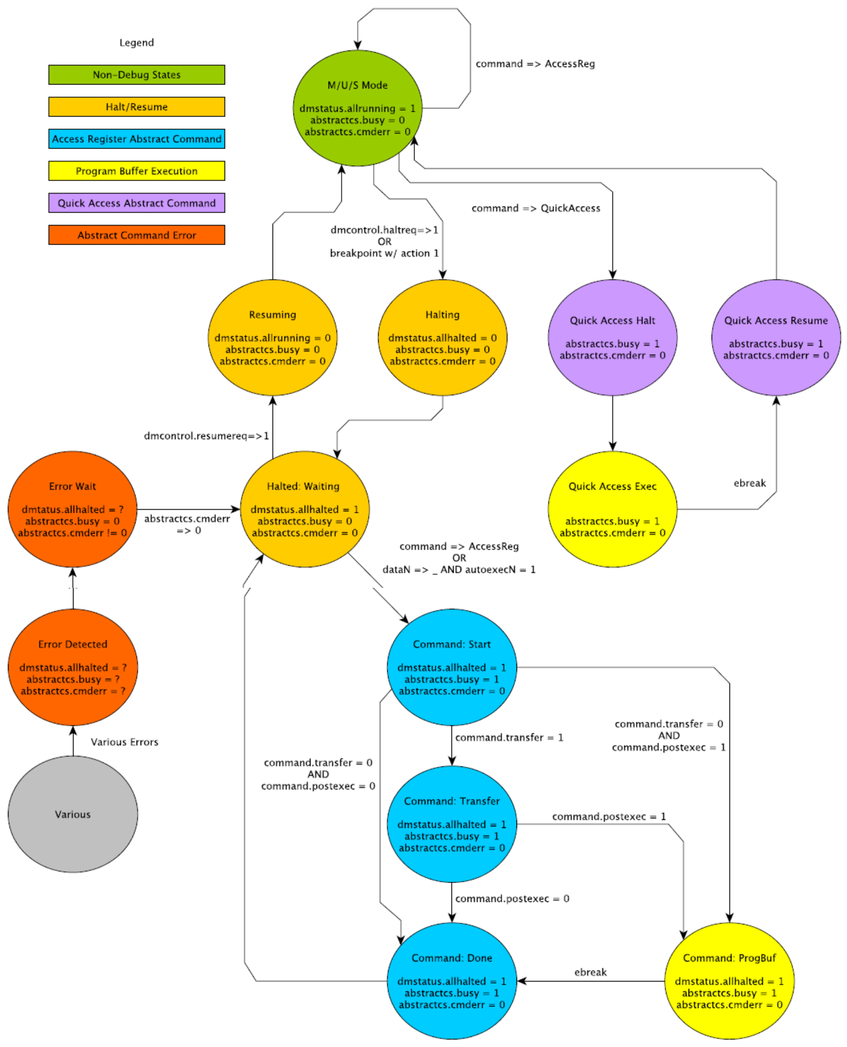
ID	REFERENCE	TYPE	DEFINITION
RVD.3.158	3.6.1.3 (p.15)	R	

Field	Description
cmdtype	This is 2 to indicate Access Memory Command.
aamvirtual	An implementation does not have to implement both virtual and physical accesses, but it must fail accesses that it doesn't support. 0: Addresses are physical (to the hart they are performed on). 1: Addresses are virtual, and translated the way they would be from M-mode, with MPRV set.
aamsize	0: Access the lowest 8 bits of the memory location. 1: Access the lowest 16 bits of the memory location. 2: Access the lowest 32 bits of the memory location. 3: Access the lowest 64 bits of the memory location. 4: Access the lowest 128 bits of the memory location.
aampostincrement	After a memory access has completed, if this bit is 1, increment `arg1` (which contains the address used) by the number of bytes encoded in aamsize.
write	0: Copy data from the memory location specified in `arg1` into `arg0` portion of data. 1: Copy data from arg0 portion of data into the memory location specified in `arg1`.
target-specific	These bits are reserved for target-specific uses.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.159	3.7 (p.16)	H	Program Buffer

ID	REFERENCE	TYPE	DEFINITION
RVD.3.160	3.7 (p.16)	I	To support executing arbitrary instructions on a halted hart, a Debug Module can include a Program Buffer that a debugger can write small programs to.
RVD.3.161	3.7 (p.16)	R	Systems that support all necessary functionality using abstract commands only may choose to omit the Program Buffer.
RVD.3.162	3.7 (p.16)	R	A debugger can write a small program to the Program Buffer, and then execute it exactly once with the Access Register Abstract Command, setting the postexec bit in `command`.
RVD.3.163	3.7 (p.16)	R	The debugger can write whatever program it likes (including jumps out of the Program Buffer), but the program must end with **ebreak** or **c.ebreak**.
RVD.3.164	3.7 (p.16)	R	An implementation may support an implied **ebreak** that is executed when a hart runs off the end of the Program Buffer.
RVD.3.165	3.7 (p.16)	R	This is indicated by impebreak.
RVD.3.166	3.7 (p.16)	R	With this feature, a Program Buffer of just 2 32-bit words can offer effcient debugging.
RVD.3.167	3.7 (p.16)	R	If progbufsize is 1, impebreak must be 1.
RVD.3.168	3.7 (p.16)	R	It is possible that the Program Buffer can hold only one 32 or 16-bit instruction, so the debugger must only write a single instruction in this case, regardless of its size.
RVD.3.169	3.7 (p.16)	R	This instruction can be a 32-bit instruction, or a compressed instruction in the lower 16 bits accompanied by a compressed **nop** in the upper 16 bits.
RVD.3.170	3.7 (p.16)	C	The slightly inconsistent behavior with a Program Buffer of size 1 is to accommodate hardware designs that prefer to stuff instructions directly into the pipeline when halted, instead of having the Program Buffer exist in the address space somewhere.
RVD.3.171	3.7 (p.16)	R	While these programs are executed, the hart does not leave Debug Mode (see Section 4.1).
RVD.3.172	3.7 (p.16)	R	If an exception is encountered during execution of the Program Buffer, no more instructions are executed, the hart remains in Debug Mode, and cmderr is set to 3 (exception error).
RVD.3.173	3.7 (p.16)	R	If the debugger executes a program that doesn't terminate with an **ebreak** instruction, the hart will remain in Debug Mode and the debugger will lose control of the hart.
RVD.3.174	3.7 (p.16)	R	Executing the Program Buffer may clobber dpc.
RVD.3.175	3.7 (p.16)	R	If that is the case, it must be possible to read/write dpc using an abstract command with postexec not set.
RVD.3.176	3.7 (p.16)	R	The debugger must attempt to save dpc between halting and executing a Program Buffer, and then restore dpc before leaving Debug Mode.
RVD.3.177	3.7 (p.16)	C	Allowing Program Buffer execution to clobber dpc allows for direct implementations that don't have a separate PC register, and do need to use the PC when executing the Program Buffer.
RVD.3.178	3.7 (p.16)	R	The Program Buffer may be implemented as RAM which is accessible to the hart.
RVD.3.179	3.7 (p.16)	R	A debugger can determine if this is the case by executing small programs that attempt to write and read back relative to pc while executing from the Program Buffer.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.180	3.7 (p.16)	R	If so, the debugger has more flexibility in what it can do with the program buffer.
RVD.3.181	3.8 (p.16)	H	Overview of States
RVD.3.182	3.8 (p.16)	R	Figure 3.1 shows a conceptual view of the states passed through by a hart during run/halt debugging as influenced by the different fields of `dmcontrol`, `abstractcs`, `abstractauto`, and `command`.
RVD.3.183	3.8 (p.17) Figure 3.1	R	Run/Halt Debug State Machine for single-hart systems. As only a small amount of state is visible to the debugger, the states and transitions are conceptual.


Legend
Non-Debug States
Halt/Resume
Access Register Abstract Command
Program Buffer Execution
Quick Access Abstract Command
Abstract Command Error
M/U/S Mode dmstatus.allrunning = 1 abstractcs.busy = 0 abstractcs.cmderr = 0
command => AccessReg
command => QuickAccess
dmcontrol.haltreq=>1 OR breakpoint w/ action 1
Resuming dmstatus.allrunning = 0 abstractcs.busy = 0 abstractcs.cmderr = 0
Halting dmstatus.allhalted = 0 abstractcs.busy = 0 abstractcs.cmderr = 0
Quick Access Halt abstractcs.busy = 1 abstractcs.cmderr = 0
Quick Access Resume abstractcs.busy = 1 abstractcs.cmderr = 0
dmcontrol.resumereq=>1
Error Wait dmtatus.allhalted = ? abstractcs.busy = 0 abstractcs.cmderr != 0
abstractcs.cmderr => 0
Halted: Waiting dmstatus.allhalted = 1 abstractcs.busy = 0 abstractcs.cmderr = 0
Quick Access Exec abstractcs.busy = 1 abstractcs.cmderr = 0
ebreak
command => AccessReg OR dataN => _ AND autoexecN = 1
Error Detected dmstatus.allhalted = ? abstractcs.busy = ? abstractcs.cmderr = ?
Various Errors
Various
Command: Start dmstatus.allhalted = 1 abstractcs.busy = 1 abstractcs.cmderr = 0
command.transfer = 1
command.transfer = 0 AND command.postexec = 1
command.transfer = 0 AND command.postexec = 0
Command: Transfer dmstatus.allhalted = 1 abstractcs.busy = 1 abstractcs.cmderr = 0
command.postexec = 1
command.postexec = 0
Command: Done dmstatus.allhalted = 1 abstractcs.busy = 1 abstractcs.cmderr = 0
ebreak
Command: ProgBuf dmstatus.allhalted = 1 abstractcs.busy = 1 abstractcs.cmderr = 0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.184	3.9 (p.18)	H	System Bus Access
RVD.3.185	3.9 (p.18)	O	A debugger can access memory from a hart's point of view using a Program Buffer or the Abstract Access Memory command. (Both these features are optional.)
RVD.3.186	3.9 (p.18)	O	A Debug Module may also include a System Bus Access block to provide memory access without involving a hart, regardless of whether Program Buffer is implemented.
RVD.3.187	3.9 (p.18)	R	The System Bus Access block uses physical addresses.
RVD.3.188	3.9 (p.18)	R	The System Bus Access block may support 8-, 16-, 32-, 64-, and 128-bit accesses.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.189	3.9 (p.18) Table 3.7	R	The table below shows which bits in `sbdata` are used for each access size. System Bus Data Bits

Access Size	Data Bits
8	`sbdata0` bits 7:0
16	`sbdata0` bits 15:0
32	`sbdata0`
64	`sbdata1`, `sbdata0`
128	`sbdata3`, `sbdata2`, `sbdata1`, `sbdata0`

ID	REFERENCE	TYPE	DEFINITION
RVD.3.190	3.9 (p.18)	I	Depending on the microarchitecture, data accessed through System Bus Access may not always be coherent with that observed by each hart.
RVD.3.191	3.9 (p.18)	R	It is up to the debugger to enforce coherency if the implementation does not.
RVD.3.192	3.9 (p.18)	I	This specification does not define a standard way to do this. Possibilities may include writing to special memory-mapped locations, or executing special instructions via the Program Buffer.
RVD.3.193	3.9 (p.18)	C	Implementing a System Bus Access block has several benefits even when a Debug Module also implements a Program Buffer. First, it is possible to access memory in a running system with minimal impact. Second, it may improve performance when accessing memory. Third, it may provide access to devices that a hart does not have access to.
RVD.3.194	3.10 (p.18)	H	Minimally Intrusive Debugging
RVD.3.195	3.10 (p.18)	I	Depending on the task it is performing, some harts can only be halted very briefly. There are several mechanisms that allow accessing resources in such a running system with a minimal impact on the running hart.
RVD.3.196	3.10 (p.18)	I	First, an implementation may allow some abstract commands to execute without halting the hart.
RVD.3.197	3.10 (p.18)	I	Second, the Quick Access abstract command can be used to halt a hart, quickly execute the contents of the Program Buffer, and let the hart run again. Combined with instructions that allow Program Buffer code to access the data registers, as described in 3.12.3, this can be used to quickly perform a memory or register access. For some systems this will be too intrusive, but many systems that can't be halted can bear an occasional hiccup of a hundred or less cycles.
RVD.3.198	3.10 (p.18)	I	Third, if the System Bus Access block is implemented, it can be used while a hart is running to access system memory.
RVD.3.199	3.11 (p.19)	H	Security
RVD.3.200	3.11 (p.19)	I	To protect intellectual property it may be desirable to lock access to the Debug Module.
RVD.3.201	3.11 (p.19)	O	To allow access during a manufacturing process and not afterwards, a reasonable solution could be to add a fuse bit to the Debug Module that can be used to be permanently disable it.
RVD.3.202	3.11 (p.19)	I	Since this is technology specific, it is not further addressed in this spec.
RVD.3.203	3.11 (p.19)	O	Another option is to allow the DM to be unlocked only by users who have an access key.
RVD.3.204	3.11 (p.19)	I	Between authenticated, authbusy, and authdata arbitrarily complex authentication mechanism can be supported.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.205	3.11 (p.19)	R	When authenticated is clear, the DM must not interact with the rest of the platform, nor expose details about the harts connected to the DM.
RVD.3.206	3.11 (p.19)	R	All DM registers should read 0, while writes should be ignored, with the following mandatory exceptions: 1. authenticated in `dmstatus` is readable. 2. authbusy in `dmstatus` is readable. 3. version in `dmstatus` is readable. 4. dmactive in `dmcontrol` is readable and writable. 5. `authdata` is readable and writable.
RVD.3.207	3.12 (p.19)	H	Debug Module Registers
RVD.3.208	3.12 (p.19)	R	The registers described in this section are accessed over the DMI bus.
RVD.3.209	3.12 (p.19)	R	Each DM has a base address (which is 0 for the first DM).
RVD.3.210	3.12 (p.19)	R	The register addresses below are offsets from this base address.
RVD.3.211	3.12 (p.19)	R	When read, unimplemented Debug Module DMI Registers return 0. Writing them has no effect.
RVD.3.212	3.12 (p.19)	R	For each register it is possible to determine that it is implemented by reading it and getting a non-zero value (e.g. `sbcs`), or by checking bits in another register (e.g. progbufsize).

ID	REFERENCE	TYPE	DEFINITION
RVD.3.213	3.12 (p.20) Table 3.8	R	Debug Module Debug Bus Registers

Address	Name
`0x04`	Abstract Data 0 (`data0`)
`0x0f`	Abstract Data 11 (`data11`)
`0x10`	Debug Module Control (`dmcontrol`)
`0x11`	Debug Module Status (`dmstatus`)
`0x12`	Hart Info (`hartinfo`)
`0x13`	Halt Summary 1 (`haltsum1`)
`0x14`	Hart Array Window Select (`hawindowsel`)
`0x15`	Hart Array Window (`hawindow`)
`0x16`	Abstract Control and Status (`abstractcs`)
`0x17`	Abstract Command (`command`)
`0x18`	Abstract Command Autoexec (`abstractauto`)
`0x19`	Configuration String Pointer 0 (`confstrptr0`)
`0x1a`	Configuration String Pointer 1 (`confstrptr1`)
`0x1b`	Configuration String Pointer 2 (`confstrptr2`)
`0x1c`	Configuration String Pointer 3 (`confstrptr3`)
`0x1d`	Next Debug Module (`nextdm`)
`0x20`	Program Buffer 0 (`progbuf0`)
`0x2f`	Program Buffer 15 (`progbuf15`)
`0x30`	Authentication Data (`authdata`)
`0x34`	Halt Summary 2 (`haltsum2`)
`0x35`	Halt Summary 3 (`haltsum3`)
`0x37`	System Bus Address 127:96 (`sbaddress3`)
`0x38`	System Bus Access Control and Status (`sbcs`)
`0x39`	System Bus Address 31:0 (`sbaddress0`)
`0x3a`	System Bus Address 63:32 (`sbaddress1`)
`0x3b`	System Bus Address 95:64 (`sbaddress2`)
`0x3c`	System Bus Data 31:0 (`sbdata0`)
`0x3d`	System Bus Data 63:32 (`sbdata1`)
`0x3e`	System Bus Data 95:64 (`sbdata2`)
`0x3f`	System Bus Data 127:96 (`sbdata3`)
`0x40`	Halt Summary 0 (`haltsum0`)

ID	REFERENCE	TYPE	DEFINITION
RVD.3.214	3.12.1 (p.20)	H	Debug Module Status (dmstatus, at 0x11)
RVD.3.215	3.12.1 (p.20)	R	This register reports status for the overall Debug Module as well as the currently selected harts, as defined in hasel.
RVD.3.216	3.12.1 (p.20)	I	Its address will not change in the future, because it contains version.
RVD.3.217	3.12.1 (p.20)	R	This entire register is read-only.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.218	3.12.1 (p.20, p.21)	R	

31	...	23	22	21	20	19	18
0			impebreak	0		allhavereset	anyhavereset

17	16	15	14	13
allresumeack	anyresumeack	allnonexistent	anynonexistent	allunavail

12	11	10	9	8
anyunavail	allrunning	anyrunning	allhalted	anyhalted

7	6	5	4	3	...	0
authenticated	authbusy	hasresethaltreq	confstrptrvalid	version		

ID	REFERENCE	TYPE	DEFINITION
RVD.3.219	3.12.1 (p.21)	R	

Field	Description	Access	Reset
impebreak	If 1, then there is an implicit **ebreak** instruction at the non-existent word immediately after the Program Buffer. This saves the debugger from having to write the **ebreak** itself, and allows the Program Buffer to be one word smaller. This must be 1 when progbufsize is 1.	R	Preset
allhavereset	This field is 1 when all currently selected harts have been reset and reset has not been acknowledged for any of them.	R	-
anyhavereset	This field is 1 when at least one currently selected hart has been reset and reset has not been acknowledged for that hart.	R	-
allresumeack	This field is 1 when all currently selected harts have acknowledged their last resume request.	R	-
anyresumeack	This field is 1 when any currently selected hart has acknowledged its last resume request.	R	-
allnonexistent	This field is 1 when all currently selected harts do not exist in this platform.	R	-
anynonexistent	This field is 1 when any currently selected hart does not exist in this platform.	R	-
allunavail	This field is 1 when all currently selected harts are unavailable.	R	-
anyunavail	This field is 1 when any currently selected hart is unavailable.	R	-
allrunning	This field is 1 when all currently selected harts are running.	R	-
anyrunning	This field is 1 when any currently selected hart is running.	R	-
allhalted	This field is 1 when all currently selected harts are halted.	R	-
anyhalted	This field is 1 when any currently selected hart is halted.	R	-

ID	REFERENCE	TYPE	DEFINITION
RVD.3.220	3.12.1 (p.22)	R	

Field	Description	Access	Reset
authenticated	0: Authentication is required before using the DM. 1: The authentication check has passed. On components that don't implement authentication, this bit must be preset as 1.	R	Preset
authbusy	0: The authentication module is ready to process the next read/write to `authdata`. 1: The authentication module is busy. Accessing `authdata` results in unspecified behavior. authbusy only becomes set in immediate response to an access to `authdata`.	R	0
hasresethaltreq	1 if this Debug Module supports halt-on-reset functionality controllable by the setresethaltreq and clrresethaltreq bits. 0 otherwise.	R	Preset
confstrptrvalid	0: `confstrptr0`-`confstrptr3` hold information which is not relevant to the configuration string. 1: `confstrptr0`-`confstrptr3` hold the address of the configuration string.	R	Preset
version	0: There is no Debug Module present. 1: There is a Debug Module and it conforms to version 0.11 of this specification. 2: There is a Debug Module and it conforms to version 0.13 of this specification. 15: There is a Debug Module but it does not conform to any available version of this spec.	R	2

RVD.3.221	3.12.2 (p.22)	H	Debug Module Control (`dmcontrol`, at 0x10)
RVD.3.222	3.12.2 (p.22)	I	This register controls the overall Debug Module as well as the currently selected harts, as defined in hasel.
RVD.3.223	3.12.2 (p.22)	I	Throughout this document we refer to hartsel, which is hartselhi combined with hartsello.
RVD.3.224	3.12.2 (p.22)	O	While the spec allows for 20 hartsel bits, an implementation may choose to implement fewer than that.
RVD.3.225	3.12.2 (p.22)	I	The actual width of hartsel is called HARTSELLEN.
RVD.3.226	3.12.2 (p.22)	R	It must be at least 0 and at most 20.
RVD.3.227	3.12.2 (p.22)	R	A debugger should discover HARTSELLEN by writing all ones to hartsel (assuming the maximum size) and reading back the value to see which bits were actually set.
RVD.3.228	3.12.2 (p.22)	R	Debuggers must not change hartsel while an abstract command is executing.
RVD.3.229	3.12.2 (p.22)	C	There are separate setresethaltreq and clrresethaltreq bits so that it is possible to write `dmcontrol` without changing the halt-on-reset request bit for each selected hart, when not all selected harts have the same configuration.
RVD.3.230	3.12.2 (p.22)	R	On any given write, a debugger may only write 1 to at most one of the following bits: resumereq, hartreset, ackhavereset, setresethaltreq, and clrresethaltreq.
RVD.3.231	3.12.2 (p.22)	R	The others must be written 0.
RVD.3.232	3.12.2 (p.23)	R	resethaltreq is an optional internal bit of per-hart state that cannot be read, but can be written with setresethaltreq and clrresethaltreq.

ID **REFERENCE** **TYPE** **DEFINITION**

RVD.3.233 3.12.2 (p.23) R

31	30	29	28	27	26	25 … 16
haltreq	resumereq	hartreset	ackhavereset	0	hasel	hartsello

15 … 6	5 4	3	2	1	0
hartselhi	0	setresethaltreq	clrresethaltreq	ndmreset	dmactive

RVD.3.234 3.12.2 (p.23) R

Field	**Description**	**Access**	**Reset**
haltreq	Writing 0 clears the halt request bit for all currently selected harts. This may cancel outstanding halt requests for those harts. Writing 1 sets the halt request bit for all currently selected harts. Running harts will halt whenever their halt request bit is set. Writes apply to the new value of hartsel and hasel.	W	-
resumereq	Writing 1 causes the currently selected harts to resume once, if they are halted when the write occurs. It also clears the resume ack bit for those harts. resumereq is ignored if haltreq is set. Writes apply to the new value of hartsel and hasel.	W1	-
hartreset	This optional field writes the reset bit for all the currently selected harts. To perform a reset the debugger writes 1, and then writes 0 to deassert the reset signal. While this bit is 1, the debugger must not change which harts are selected. If this feature is not implemented, the bit always stays 0, so after writing 1 the debugger can read the register back to see if the feature is supported. Writes apply to the new value of hartsel and hasel.	R/W	0
ackhavereset	0: No effect. 1: Clears `havereset` for any selected harts. Writes apply to the new value of hartsel and hasel.	W1	-

ID **REFERENCE** **TYPE** **DEFINITION**

RVD.3.235 3.12.2 (p.24) R

Field	Description	Access	Reset
hasel	Selects the definition of currently selected harts. 0: There is a single currently selected hart, that is selected by hartsel. 1: There may be multiple currently selected harts- the hart selected by hartsel, plus those selected by the hart array mask register. An implementation which does not implement the hart array mask register must tie this field to 0. A debugger which wishes to use the hart array mask register feature should set this bit and read back to see if the functionality is supported.	R/W	0
hartsello	The low 10 bits of hartsel: the DM-specific index of the hart to select. This hart is always part of the currently selected harts.	R/W	0
hartselhi	The high 10 bits of hartsel: the DM-specific index of the hart to select. This hart is always part of the currently selected harts.	R/W	0
setresethaltreq	This optional field writes the halt-on-reset request bit for all currently selected harts, unless clrresethaltreq is simultaneously set to 1. When set to 1, each selected hart will halt upon the next deassertion of its reset. The halt-on-reset request bit is not automatically cleared. The debugger must write to clrresethaltreq to clear it. Writes apply to the new value of hartsel and hasel. If hasresethaltreq is 0, this field is not implemented.	W1	-
clrresethaltreq	This optional field clears the halt-on-reset request bit for all currently selected harts. Writes apply to the new value of hartsel and hasel.	W1	-
ndmreset	This bit controls the reset signal from the DM to the rest of the system. The signal should reset every part of the system, including every hart, except for the DM and any logic required to access the DM. To perform a system reset the debugger writes 1, and then writes 0 to deassert the reset.	R/W	0

RVD.3.236 3.12.2 (p.25) R

Field	Description	Access	Reset
dmactive	This bit serves as a reset signal for the Debug Module itself. 0: The module's state, including authentication mechanism, takes its reset values (the dmactive bit is the only bit which can be written to something other than its reset value). 1: The module functions normally. No other mechanism should exist that may result in resetting the Debug Module after power up, with the possible (but not recommended) exception of a global reset signal that resets the entire platform. A debugger may pulse this bit low to get the Debug Module into a known state. Implementations may pay attention to this bit to further aid debugging, for example by preventing the Debug Module from being power gated while debugging is active.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.237	3.12.3 (p.25)	H	Hart Info (`hartinfo`, at 0x12)
RVD.3.238	3.12.3 (p.25)	R	This register gives information about the hart currently selected by hartsel.
RVD.3.239	3.12.3 (p.25)	O	This register is optional.
RVD.3.240	3.12.3 (p.25)	R	If it is not present it should read all-zero.
RVD.3.241	3.12.3 (p.25)	I	If this register is included, the debugger can do more with the Program Buffer by writing programs which explicitly access the data and/or `dscratch` registers.
RVD.3.242	3.12.3 (p.25)	R	This entire register is read-only.
RVD.3.243	3.12.3 (p.25)	R	(register layout below)

31 ... 24	23 ... 20	19 ... 17	16	15 ... 12	11 ... 0
0	nscratch	0	dataaccess	datasize	dataaddr

ID	REFERENCE	TYPE	DEFINITION
RVD.3.244	3.12.3 (p.25, p.26)	R	(field table below)

Field	Description	Access	Reset
nscratch	Number of `dscratch` registers available for the debugger to use during program buffer execution, starting from `dscratch0`. The debugger can make no assumptions about the contents of these registers between commands.	R	Preset
dataaccess	0: The data registers are shadowed in the hart by CSRs. Each CSR is DXLEN bits in size, and corresponds to a single argument, per Table 3.1 (RVD.3.113). 1: The data registers are shadowed in the hart's memory map. Each register takes up 4 bytes in the memory map.	R	Preset
datasize	If dataaccess is 0: Number of CSRs dedicated to shadowing the data registers. If dataaccess is 1: Number of 32-bit words in the memory map dedicated to shadowing the data registers. Since there are at most 12 data registers, the value in this register must be 12 or smaller.	R	Preset
dataaddr	If dataaccess is 0: The number of the first CSR dedicated to shadowing the data registers. If dataaccess is 1: Signed address of RAM where the data registers are shadowed, to be used to access relative to zero.	R	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.3.245	3.12.4 (p.26)	H	Hart Array Window Select (`hawindowsel`, at 0x14)
RVD.3.246	3.12.4 (p.26)	I	This register selects which of the 32-bit portion of the hart array mask register (see Section 3.3.2) is accessible in `hawindow`.
RVD.3.247	3.12.4 (p.26)	R	(register layout below)

31 ... 15	14 ... 0
0	hawindowsel

ID	REFERENCE	TYPE	DEFINITION
RVD.3.248	3.12.3 (p.25, p.26)	R	(field table below)

Field	Description	Access	Reset
hawindowsel	The high bits of this field may be tied to 0, depending on how large the array mask register is. E.g. on a system with 48 harts only bit 0 of this field may actually be writable.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.249	3.12.5 (p.26)	H	Hart Array Window (`hawindow`, at 0x15)
RVD.3.250	3.12.5 (p.26)	I	This register provides R/W access to a 32-bit portion of the hart array mask register (see Section 3.3.2).

ID	REFERENCE	TYPE	DEFINITION
RVD.3.251	3.12.5 (p.26)	R	The position of the window is determined by `hawindowsel`. I.e. bit 0 refers to hart `hawindowsel` * 32, while bit 31 refers to hart `hawindowsel` * 32 + 31.
RVD.3.252	3.12.5 (p.27)	R	Since some bits in the hart array mask register may be constant 0, some bits in this register may be constant 0, depending on the current value of `hawindowsel`.
RVD.3.253	3.12.5 (p.27)	R	

31	...	0
	maskdata	

ID	REFERENCE	TYPE	DEFINITION
RVD.3.254	3.12.6 (p.27)	H	Abstract Control and Status (`abstractcs`, at 0x16)
RVD.3.255	3.12.6 (p.27)	R	Writing this register while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.
RVD.3.256	3.12.6 (p.27)	C	datacount must be at least 1 to support RV32 harts, 2 to support RV64 harts, or 4 to support RV128 harts.
RVD.3.257	3.12.6 (p.27)	R	

31 ... 29	28 ... 24	23 ... 13	12	11	10 ... 8	7 ... 4	3 ... 0
0	progbufsize	0	busy	0	cmderr	0	datacount

ID	REFERENCE	TYPE	DEFINITION
RVD.3.258	3.12.6 (p.27, p.28)	R	

Field	Description	Access	Reset
progbufsize	Size of the Program Buffer, in 32-bit words. Valid sizes are 0 - 16.	R	Preset
busy	1: An abstract command is currently being executed. This bit is set as soon as `command` is written, and is not cleared until that command has completed.	R	0
cmderr	Gets set if an abstract command fails. The bits in this field remain set until they are cleared by writing 1 to them. No abstract command is started until the value is reset to 0. This field only contains a valid value if busy is 0. 0 (none): No error. 1 (busy): An abstract command was executing while `command`, `abstractcs`, or `abstractauto` was written, or when one of the `data` or `progbuf` registers was read or written. This status is only written if cmderr contains 0. 2 (not supported): The requested command is not supported, regardless of whether the hart is running or not. 3 (exception): An exception occurred while executing the command (e.g. while executing the Program Buffer). 4 (halt/resume): The abstract command couldn't execute because the hart wasn't in the required state (running/halted), or unavailable. 5 (bus): The abstract command failed due to a bus error (e.g. alignment, access size, or timeout). 7 (other): The command failed for another reason.	R/W1C	0
datacount	Number of data registers that are implemented as part of the abstract command interface. Valid sizes are 1 - 12.	R	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.3.259	3.12.7 (p.28)	H	Abstract Command (`command`, at 0x17)
RVD.3.260	3.12.7 (p.28)	R	Writes to this register cause the corresponding abstract command to be executed.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.261	3.12.7 (p.28)	R	Writing this register while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.
RVD.3.262	3.12.7 (p.28)	R	If cmderr is non-zero, writes to this register are ignored.
RVD.3.263	3.12.7 (p.28)	C	cmderr inhibits starting a new command to accommodate debuggers that, for performance reasons, send several commands to be executed in a row without checking cmderr in between. They can safely do so and check cmderr at the end without worrying that one command failed but then a later command (which might have depended on the previous one succeeding) passed.
RVD.3.264	3.12.7 (p.29)	R	

31	...	24	23	...	0
cmdtype			control		

ID	REFERENCE	TYPE	DEFINITION
RVD.3.265	3.12.7 (p.29)	R	

Field	Description	Access	Reset
cmdtype	The type determines the overall functionality of this abstract command.	W	0
control	This field is interpreted in a command-specific manner, described for each abstract command.	W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.266	3.12.8 (p.29)	H	Abstract Command Autoexec (`abstractauto`, at 0x18)
RVD.3.267	3.12.8 (p.29)	O	This register is optional.
RVD.3.268	3.12.8 (p.29)	I	Including it allows more effcient burst accesses.
RVD.3.269	3.12.8 (p.29)	R	A debugger can detect whether it is support by setting bits and reading them back.
RVD.3.270	3.12.8 (p.29)	R	Writing this register while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.
RVD.3.271	3.12.8 (p.29)	R	

31	...	16	15	...	12	11	...	0
autoexecprogbuf			0			autoexecdata		

ID	REFERENCE	TYPE	DEFINITION
RVD.3.272	3.12.8 (p.29)	R	

Field	Description	Access	Reset
autoexecprogbuf	When a bit in this field is 1, read or write accesses to the corresponding progbuf word cause the command in `command` to be executed again.	R/W	0
autoexecdata	When a bit in this field is 1, read or write accesses to the corresponding data word cause the command in `command` to be executed again.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.273	3.12.9 (p.29)	H	Configuration String Pointer 0 (`confstrptr0`, at 0x19)
RVD.3.274	3.12.9 (p.29)	R	When confstrptrvalid is set, reading this register returns bits 31:0 of the configuration string pointer.
RVD.3.275	3.12.9 (p.29)	R	Reading the other `confstrptr` registers returns the upper bits of the address.
RVD.3.276	3.12.9 (p.29)	R	When system bus mastering is implemented, this must be an address that can be used with the System Bus Access module.
RVD.3.277	3.12.9 (p.29)	R	Otherwise, this must be an address that can be used to access the configuration string from the hart with ID 0.
RVD.3.278	3.12.9 (p.30)	R	If confstrptrvalid is 0, then the `confstrptr` registers hold identifier information which is not further specified in this document.
RVD.3.279	3.12.9 (p.30)	I	The configuration string itself is described in the Privileged Spec.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.280	3.12.9 (p.30)	R	This entire register is read-only.
RVD.3.281	3.12.9 (p.30)	R	31 … 0 addr
RVD.3.282	3.12.10 (p.30)	H	Next Debug Module (`nextdm`, at 0x1d)
RVD.3.283	3.12.10 (p.30)	R	If there is more than one DM accessible on this DMI, this register contains the base address of the next one in the chain, or 0 if this is the last one in the chain.
RVD.3.284	3.12.10 (p.30)	R	This entire register is read-only.
RVD.3.285	3.12.10 (p.30)	R	31 … 0 addr
RVD.3.286	3.12.11 (p.30)	H	Abstract Data 0 (`data0`, at 0x04)
RVD.3.287	3.12.11 (p.30)	R	`data0` through `data11` are basic read/write registers that may be read or changed by abstract commands.
RVD.3.288	3.12.11 (p.30)	R	datacount indicates how many of them are implemented, starting at `data0`, counting up.
RVD.3.289	3.12.11 (p.30)	I	Table 3.1 (RVD.3.113) shows how abstract commands use these registers.
RVD.3.290	3.12.11 (p.30)	R	Accessing these registers while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.
RVD.3.291	3.12.11 (p.30)	R	Attempts to write them while busy is set does not change their value.
RVD.3.292	3.12.11 (p.30)	R	The values in these registers may not be preserved after an abstract command is executed.
RVD.3.293	3.12.11 (p.30)	R	The only guarantees on their contents are the ones offered by the command in question.
RVD.3.294	3.12.11 (p.30)	R	If the command fails, no assumptions can be made about the contents of these registers.
RVD.3.295	3.12.11 (p.30)	R	31 … 0 data
RVD.3.296	3.12.12 (p.30)	H	Program Buffer 0 (`progbuf0`, at 0x20)
RVD.3.297	3.12.12 (p.30)	R	`progbuf0` through `progbuf15` provide read/write access to the optional program buffer.
RVD.3.298	3.12.12 (p.30)	R	progbufsize indicates how many of them are implemented starting at `progbuf0`, counting up.
RVD.3.299	3.12.12 (p.31)	R	Accessing these registers while an abstract command is executing causes cmderr to be set to 1 (busy) if it is 0.
RVD.3.300	3.12.12 (p.31)	R	Attempts to write them while busy is set does not change their value.
RVD.3.301	3.12.12 (p.31)	R	31 … 0 data
RVD.3.302	3.12.13 (p.31)	H	Authentication Data (`authdata`, at 0x30)
RVD.3.303	3.12.13 (p.31)	R	This register serves as a 32-bit serial port to/from the authentication module.
RVD.3.304	3.12.13 (p.31)	R	When authbusy is clear, the debugger can communicate with the authentication module by reading or writing this register.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.305	3.12.13 (p.31)	R	There is no separate mechanism to signal overflow/underflow.
RVD.3.306	3.12.13 (p.31)	R	31 ... 0 data
RVD.3.307	3.12.14 (p.31)	H	Halt Summary 0 (`haltsum0`, at 0x40)
RVD.3.308	3.12.14 (p.31)	I	Each bit in this read-only register indicates whether one specific hart is halted or not.
RVD.3.309	3.12.14 (p.31)	R	Unavailable/nonexistent harts are not considered to be halted.
RVD.3.310	3.12.14 (p.31)	R	The LSB reflects the halt status of hart {hartsel[19:5],5'h0}, ...
RVD.3.311	3.12.14 (p.31)	R	… and the MSB reflects halt status of hart {hartsel[19:5],5'h1f}.
RVD.3.312	3.12.14 (p.31)	R	This entire register is read-only.
RVD.3.313	3.12.14 (p.31)	R	31 ... 0 haltsum0
RVD.3.314	3.12.15 (p.31)	H	Halt Summary 1 (`haltsum1`, at 0x13)
RVD.3.315	3.12.15 (p.31)	I	Each bit in this read-only register indicates whether any of a group of harts is halted or not.
RVD.3.316	3.12.15 (p.31)	R	Unavailable/nonexistent harts are not considered to be halted.
RVD.3.317	3.12.15 (p.31)	R	This register may not be present in systems with fewer than 33 harts.
RVD.3.318	3.12.15 (p.31)	R	The LSB reflects the halt status of harts {hartsel[19:10],10'h0} through {hartsel[19:10],10'h1f}.
RVD.3.319	3.12.15 (p.31)	R	The MSB reflects the halt status of harts {hartsel[19:10],10'h3e0} through {hartsel[19:10],10'h3ff}.
RVD.3.320	3.12.15 (p.31)	R	This entire register is read-only.
RVD.3.321	3.12.15 (p.32)	R	31 ... 0 haltsum1
RVD.3.322	3.12.16 (p.32)	H	Halt Summary 2 (`haltsum2`, at 0x34)
RVD.3.323	3.12.16 (p.32)	I	Each bit in this read-only register indicates whether any of a group of harts is halted or not.
RVD.3.324	3.12.16 (p.32)	R	Unavailable/nonexistent harts are not considered to be halted.
RVD.3.325	3.12.16 (p.32)	R	This register may not be present in systems with fewer than 1025 harts.
RVD.3.326	3.12.16 (p.32)	R	The LSB reflects the halt status of harts fhartsel[19:15],15'h0g through fhartsel[19:15],15'h3ffg.
RVD.3.327	3.12.16 (p.32)	R	The MSB reflects the halt status of harts fhartsel[19:15],15'h7c00g through fhartsel[19:15],15'h7fffg.
RVD.3.328	3.12.16 (p.32)	R	This entire register is read-only.
RVD.3.329	3.12.16 (p.32)	R	31 ... 0 haltsum2
RVD.3.330	3.12.17 (p.32)	H	Halt Summary 3 (`haltsum3`, at 0x35)
RVD.3.331	3.12.17 (p.32)	I	Each bit in this read-only register indicates whether any of a group of harts is halted or not.
RVD.3.332	3.12.17 (p.32)	R	Unavailable/nonexistent harts are not considered to be halted.
RVD.3.333	3.12.17 (p.32)	R	This register may not be present in systems with fewer than 32769 harts.

ID	REFERENCE	TYPE	DEFINITION
RVD.3.334	3.12.17 (p.32)	R	The LSB reflects the halt status of harts 20'h0 through 20'h7fff.
RVD.3.335	3.12.17 (p.32)	R	The MSB reflects the halt status of harts 20'hf8000 through 20'hfffff.
RVD.3.336	3.12.17 (p.32)	R	This entire register is read-only.
RVD.3.337	3.12.17 (p.32)	R	

31	...	0
	haltsum3	

ID	REFERENCE	TYPE	DEFINITION
RVD.3.338	3.12.18 (p.32)	H	System Bus Access Control and Status (`sbcs`, at 0x38)
RVD.3.339	3.12.18 (p.32, p.33)	R	

31 ... 29	28 ... 23	22	21	20
sbversion	0	sbbusyerror	sbbusy	sbreadonaddr

19 ... 17	16	15	14 ... 12	11 ... 5
sbaccess	sbautoincrement	sbreadondata	sberror	sbasize

4	3	2	1	0
sbaccess128	sbaccess64	sbaccess32	sbaccess16	sbaccess8

ID	REFERENCE	TYPE	DEFINITION
RVD.3.340	3.12.18 (p.33)	R	(see table below)

Field	Description	Access	Reset
sbversion	0: The System Bus interface conforms to mainline drafts of this spec older than 1 January, 2018. 1: The System Bus interface conforms to this version of the spec. Other values are reserved for future versions.	R	1
sbbusyerror	Set when the debugger attempts to read data while a read is in progress, or when the debugger initiates a new access while one is already in progress (while sbbusy is set). It remains set until it's explicitly cleared by the debugger. While this field is set, no more system bus accesses can be initiated by the Debug Module.	R/W1C	0
sbbusy	When 1, indicates the system bus master is busy. (Whether the system bus itself is busy is related, but not the same thing.) This bit goes high immediately when a read or write is requested for any reason, and does not go low until the access is fully completed. Writes to sbcs while sbbusy is high result in undefined behavior. A debugger must not write to sbcs until it reads sbbusy as 0.	R	0
sbreadonaddr	When 1, every write to `sbaddress0` automatically triggers a system bus read at the new address.	R/W	0
sbaccess	Select the access size to use for system bus accesses. 0: 8-bit 1: 16-bit 2: 32-bit 3: 64-bit 4: 128-bit If sbaccess has an unsupported value when the DM starts a bus access, the access is not performed and sberror is set to 4.	R/W	2
sbautoincrement	When 1, `sbaddress` is incremented by the access size (in bytes) selected in sbaccess after every system bus access.	R/W	0
sbreadondata	When 1, every read from `sbdata0` automatically triggers a system bus read at the (possibly auto-incremented) address.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.341	3.12.18 (p.34)	R	

Field	Description	Access	Reset
sberror	When the Debug Module's system bus master encounters an error, this field gets set. The bits in this field remain set until they are cleared by writing 1 to them. While this field is non-zero, no more system bus accesses can be initiated by the Debug Module. An implementation may report “Other" (7) for any error condition. 0: There was no bus error. 1: There was a timeout. 2: A bad address was accessed. 3: There was an alignment error. 4: An access of unsupported size was requested. 7: Other.	R/W1C	0
sbasize	Width of system bus addresses in bits. (0 indicates there is no bus access support.)	R	Preset
sbaccess128	1 when 128-bit system bus accesses are supported.	R	Preset
sbaccess64	1 when 64-bit system bus accesses are supported.	R	Preset
sbaccess32	1 when 32-bit system bus accesses are supported.	R	Preset
sbaccess16	1 when 16-bit system bus accesses are supported.	R	Preset
sbaccess8	1 when 8-bit system bus accesses are supported.	R	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.3.342	3.12.19 (p.34)	H	System Bus Address 31:0 (`sbaddress0`, at 0x39)
RVD.3.343	3.12.19 (p.34)	R	If sbasize is 0, then this register is not present.
RVD.3.344	3.12.19 (p.34)	R	When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.
RVD.3.345	3.12.19 (p.34)	R	If sberror is 0, sbbusyerror is 0, and sbreadonaddr is set then writes to this register start the following: 1. Set sbbusy. 2. Perform a bus read from the new value of `sbaddress`. 3. If the read succeeded and sbautoincrement is set, increment `sbaddress`. 4. Clear sbbusy.
RVD.3.346	3.12.19 (p.34)	R	

31	...	0
	address	

ID	REFERENCE	TYPE	DEFINITION
RVD.3.347	3.12.19 (p.35)	R	

Field	Description	Access	Reset
address	Accesses bits 31:0 of the physical address in `sbaddress`.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.348	3.12.20 (p.35)	H	System Bus Address 63:32 (`sbaddress1`, at 0x3a)
RVD.3.349	3.12.20 (p.35)	R	If sbasize is less than 33, then this register is not present.
RVD.3.350	3.12.20 (p.35)	R	When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.
RVD.3.351	3.12.20 (p.35)	R	

31	...	0
	address	

ID	REFERENCE	TYPE	DEFINITION
RVD.3.352	3.12.20 (p.35)	R	

Field	Description	Access	Reset
address	Accesses bits 63:32 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.353	3.12.21 (p.35)	H	System Bus Address 95:64 (`sbaddress2`, at 0x3b)

ID	REFERENCE	TYPE	DEFINITION
RVD.3.354	3.12.21 (p.35)	R	If sbasize is less than 65, then this register is not present.
RVD.3.355	3.12.21 (p.35)	R	When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.
RVD.3.356	3.12.21 (p.35)	R	

31	...	0
	address	

ID	REFERENCE	TYPE	DEFINITION
RVD.3.357	3.12.21 (p.35)	R	

Field	Description	Access	Reset
address	Accesses bits 95:64 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.358	3.12.22 (p.36)	H	System Bus Address 127:96 (`sbaddress3`, at 0x37)
RVD.3.359	3.12.22 (p.36)	R	If sbasize is less than 97, then this register is not present.
RVD.3.360	3.12.22 (p.36)	R	When the system bus master is busy, writes to this register will set sbbusyerror and don't do anything else.
RVD.3.361	3.12.21 (p.35)	R	

31	...	0
	address	

ID	REFERENCE	TYPE	DEFINITION
RVD.3.362	3.12.21 (p.35)	R	

Field	Description	Access	Reset
address	Accesses bits 127:96 of the physical address in `sbaddress` (if the system address bus is that wide).	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.363	3.12.23 (p.36)	H	System Bus Data 31:0 (`sbdata0`, at 0x3c)
RVD.3.364	3.12.23 (p.36)	R	If all of the sbaccess bits in `sbcs` are 0, then this register is not present.
RVD.3.365	3.12.23 (p.36)	R	Any successful system bus read updates `sbdata`.
RVD.3.366	3.12.23 (p.36)	R	If the width of the read access is less than the width of `sbdata`, the contents of the remaining high bits may take on any value.
RVD.3.367	3.12.23 (p.36)	R	If sberror or sbbusyerror both are not 0 then accesses do nothing.
RVD.3.368	3.12.23 (p.36)	R	If the bus master is busy then accesses set sbbusyerror, and don't do anything else.
RVD.3.369	3.12.23 (p.36)	R	Writes to this register start the following: 1. Set sbbusy. 2. Perform a bus write of the new value of `sbdata` to `sbaddress`. 3. If the write succeeded and sbautoincrement is set, increment `sbaddress`. 4. Clear sbbusy.
RVD.3.370	3.12.23 (p.36)	R	Reads from this register start the following: 1. “Return" the data. 2. Set sbbusy. 3. If sbreadondata is set, perform a system bus read from the address contained in `sbaddress`, placing the result in `sbdata`. 4. If sbautoincrement is set, increment `sbaddress`. 5. Clear sbbusy.
RVD.3.371	3.12.23 (p.37)	R	Only `sbdata0` has this behavior. The other `sbdata` registers have no side effects.
RVD.3.372	3.12.23 (p.37)	R	On systems that have buses wider than 32 bits, a debugger should access `sbdata0` after accessing the other `sbdata` registers.
RVD.3.373	3.12.23 (p.37)	R	

31	...	0
	data	

ID	REFERENCE	TYPE	Field	Description	Access	Reset
RVD.3.374	3.12.23 (p.37)	R	data	Accesses bits 31:0 of `sbdata`.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.375	3.12.24 (p.37)	H	System Bus Data 63:32 (`sbdata1`, at 0x3d)
RVD.3.376	3.12.24 (p.37)	R	If sbaccess64 and sbaccess128 are 0, then this register is not present.
RVD.3.377	3.12.24 (p.37)	R	If the bus master is busy then accesses set sbbusyerror, and don't do anything else.

ID	REFERENCE	TYPE	31	...	0
RVD.3.378	3.12.24 (p.37)	R		data	

ID	REFERENCE	TYPE	Field	Description	Access	Reset
RVD.3.379	3.12.24 (p.37)	R	data	Accesses bits 63:32 of `sbdata` (if the system bus is that wide).	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.380	3.12.25 (p.37)	H	System Bus Data 95:64 (`sbdata2`, at 0x3e)
RVD.3.381	3.12.25 (p.37)	R	This register only exists if `sbaccess128` is 1.
RVD.3.382	3.12.25 (p.37)	R	If the bus master is busy then accesses set sbbusyerror, and don't do anything else.

ID	REFERENCE	TYPE	31	...	0
RVD.3.383	3.12.25 (p.37)	R		data	

ID	REFERENCE	TYPE	Field	Description	Access	Reset
RVD.3.384	3.12.25 (p.38)	R	data	Accesses bits 95:64 of sbdata (if the system bus is that wide).	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.3.385	3.12.26 (p.38)	H	System Bus Data 127:96 (`sbdata3`, at 0x3f)
RVD.3.386	3.12.26 (p.38)	R	This register only exists if sbaccess128 is 1.
RVD.3.387	3.12.26 (p.38)	R	If the bus master is busy then accesses set sbbusyerror, and don't do anything else.

ID	REFERENCE	TYPE	31	...	0
RVD.3.388	3.12.26 (p.38)	R		data	

ID	REFERENCE	TYPE	Field	Description	Access	Reset
RVD.3.389	3.12.26 (p.38)	R	data	Accesses bits 127:96 of sbdata (if the system bus is that wide).	R/W	0

CHAPTER 4 RISC-V Debug

ID	REFERENCE	TYPE	DEFINITION
RVD.4.1	4.0 (p.39)	H	RISC-V Debug
RVD.4.2	4.0 (p.39)	I	Modifications to the RISC-V core to support debug are kept to a minimum.
RVD.4.3	4.0 (p.39)	R	There is a special execution mode (Debug Mode) and a few extra CSRs.
RVD.4.4	4.0 (p.39)	R	The DM takes care of the rest.
RVD.4.5	4.0 (p.39)	R	In order to be compliant with this specification an implementation must implement everything described in this section that is not explicitly listed as optional.
RVD.4.6	4.1 (p.39)	H	Debug Mode
RVD.4.7	4.1 (p.39)	R	Debug Mode is a special processor mode used only when a hart is halted for external debugging.
RVD.4.8	4.1 (p.39)	I	How Debug Mode is implemented is not specified here.
RVD.4.9	4.1 (p.39)	R	When executing code from the optional Program Buffer, the hart stays in Debug Mode and the following apply: (steps 1-13)
RVD.4.10	4.1 (p.39)	R	1. All operations are executed at machine mode privilege level, except that MPRV in `mstatus` may be ignored according to mprven.
RVD.4.11	4.1 (p.39)	R	2. All interrupts (including NMI) are masked.
RVD.4.12	4.1 (p.39)	R	3. Exceptions don't update any registers. That includes `cause`, `epc`, `tval`, `dpc`, and `mstatus`. They do end execution of the Program Buffer.
RVD.4.13	4.1 (p.39)	R	4. No action is taken if a trigger matches.
RVD.4.14	4.1 (p.39)	R	5. Counters may be stopped, depending on stopcount in `dcsr`.
RVD.4.15	4.1 (p.39)	R	6. Timers may be stopped, depending on stoptime in `dcsr`.
RVD.4.16	4.1 (p.39)	R	7. The **wfi** instruction acts as a **nop**.
RVD.4.17	4.1 (p.39)	R	8. Almost all instructions that change the privilege level have undefined behavior. This includes **ecall**, **mret**, **sret**, and **uret**. (To change the privilege level, the debugger can write prv in `dcsr`). The only exception is **ebreak**. When that is executed in Debug Mode, it halts the hart again but without updating `dpc` or `dcsr`.
RVD.4.18	4.1 (p.39)	R	9. Completing Program Buffer execution is considered output for the purpose of fence instructions.
RVD.4.19	4.1 (p.39)	R	10. All control transfer instructions may act as illegal instructions if their destination is in the Program Buffer. If one such instruction acts as an illegal instruction, all such instructions must act as an illegal instruction.
RVD.4.20	4.1 (p.40)	R	11. All control transfer instructions may act as illegal instructions if their destination is outside the Program Buffer. If one such instruction acts as an illegal instruction, all such instructions must act as an illegal instruction.
RVD.4.21	4.1 (p.40)	R	12. Instructions that depend on the value of the PC (e.g. `auipc`) may act as illegal instructions.
RVD.4.22	4.1 (p.40)	R	13. Effective XLEN is DXLEN.

ID	REFERENCE	TYPE	DEFINITION
RVD.4.23	4.1 (p.40)	C	In general, the debugger is expected to be able to simulate all the effects of MPRV. The exception is the case of Sv32 systems, which need MPRV functionality in order to access 34-bit physical addresses. Other systems are likely to tie mprven to 0.
RVD.4.24	4.2 (p.40)	H	Load-Reserved/Store-Conditional Instructions
RVD.4.25	4.2 (p.40)	R	The reservation registered by an **lr** instruction on a memory address may be lost when entering Debug Mode or while in Debug Mode. This means that there may be no forward progress if Debug Mode is entered between **lr** and **sc** pairs.
RVD.4.26	4.2 (p.40)	C	This is a behavior that debug users must be aware of. If they have a breakpoint set between a lr and sc pair, or are stepping through such code, the sc may never succeed. Fortunately in general use there will be very few instructions in such a sequence, and anybody debugging it will quickly notice that the reservation is not occurring. The solution in that case is to set a breakpoint on the rst instruction after the sc and run to it. A higher level debugger may choose to automate this.
RVD.4.27	4.3 (p.40)	H	Wait for Interrupt Instruction
RVD.4.28	4.3 (p.40)	R	If halt is requested while `wfi` is executing, then the hart must leave the stalled state, completing this instruction's execution, and then enter Debug Mode.
RVD.4.29	4.4 (p.40)	H	Single Step
RVD.4.30	4.4 (p.40)	R	A debugger can cause a halted hart to execute a single instruction and then re-enter Debug Mode by setting step before setting resumereq.
RVD.4.31	4.4 (p.40)	R	If executing or fetching that instruction causes an exception, Debug Mode is re-entered immediately after the PC is changed to the exception handler and the appropriate tval and cause registers are updated.
RVD.4.32	4.4 (p.40)	R	If executing or fetching the instruction causes a trigger to fire, Debug Mode is re-entered immediately after that trigger has fired.
RVD.4.33	4.4 (p.40)	R	In that case cause is set to 2 (trigger) instead of 4 (single step).
RVD.4.34	4.4 (p.40)	R	Whether the instruction is executed or not depends on the specific configuration of the trigger.
RVD.4.35	4.4 (p.40)	R	If the instruction that is executed causes the PC to change to an address where an instruction fetch causes an exception, that exception does not occur until the next time the hart is resumed.
RVD.4.36	4.4 (p.41)	R	Similarly, a trigger at the new address does not fire until the hart actually attempts to execute that instruction.
RVD.4.37	4.4 (p.41)	R	If the instruction being stepped over is **wfi** and would normally stall the hart, then instead the instruction is treated as **nop**.
RVD.4.38	4.5 (p.41)	H	Reset
RVD.4.39	4.5 (p.41)	R	If the halt signal (driven by the hart's halt request bit in the Debug Module) or resethaltreq are asserted when a hart comes out of reset, the hart must enter Debug Mode before executing any instructions, but after performing any initialization that would usually happen before the first instruction is executed.
RVD.4.40	4.6 (p.41)	H	**dret** Instruction
RVD.4.41	4.6 (p.41)	R	To return from Debug Mode, a new instruction is defined: **dret**.
RVD.4.42	4.6 (p.41)	R	It has an encoding of `0x7b200073`.

ID	REFERENCE	TYPE	DEFINITION
RVD.4.43	4.6 (p.41)	R	On harts which support this instruction, executing **dret** in Debug Mode changes `pc` to the value stored in `dpc`.
RVD.4.44	4.6 (p.41)	R	The current privilege level is changed to that specified by prv in `dcsr`.
RVD.4.45	4.6 (p.41)	R	The hart is no longer in debug mode.
RVD.4.46	4.6 (p.41)	R	Executing **dret** outside of Debug Mode causes an illegal instruction exception.
RVD.4.47	4.6 (p.41)	R	It is not necessary for the debugger to know whether an implementation supports **dret**, as the Debug Module will ensure that it is executed if necessary.
RVD.4.48	4.6 (p.41)	R	It is defined in this specification only to reserve the opcode and allow for reusable Debug Module implementations.
RVD.4.49	4.7 (p.41)	H	XLEN
RVD.4.50	4.7 (p.41)	R	While in Debug Mode, XLEN is DXLEN.
RVD.4.51	4.7 (p.41)	R	It is up to the debugger to determine the XLEN during normal program execution (by looking at misa) and to clearly communicate this to the user.
RVD.4.52	4.8 (p.41)	H	Core Debug Registers
RVD.4.53	4.8 (p.41)	R	The supported Core Debug Registers must be implemented for each hart that can be debugged.
RVD.4.54	4.8 (p.41)	R	They are CSRs, accessible using the RISC-V csr opcodes and optionally also using abstract debug commands.
RVD.4.55	4.8 (p.41)	R	These registers are only accessible from Debug Mode.
RVD.4.56	4.8 (p.42) Table 4.1	R	Core Debug Registers

Address	Name
`0x7b0`	Debug Control and Status (`dcsr`)
`0x7b1`	Debug PC (`dpc`)
`0x7b2`	Debug Scratch Register 0 (`dscratch0`)
`0x7b3`	Debug Scratch Register 1 (`dscratch1`)

RVD.4.57	4.8.1 (p.42)	H	Debug Control and Status (`dcsr`, at 0x7b0)
RVD.4.58	4.8.1 (p.42)	C	cause priorities are assigned such that the least predictable events have the highest priority.
RVD.4.59	4.8.1 (p.42)	R	

31 ... 28	27 ... 16	15	14	13	12	11	10
xdebugver	0	ebreakm	0	ebreaks	ebreaku	stepie	stopcount

9	8 ... 6	5	4	3	2	1 0
stoptime	cause	0	mprven	nmip	step	prv

ID	REFERENCE	TYPE	DEFINITION
RVD.4.60	4.8.1 (p.42)	R	

Field	Description	Access	Reset
xdebugver	0: There is no external debug support. 4: External debug support exists as it is described in this document. 15: There is external debug support, but it does not conform to any available version of this spec.	R	Preset
ebreakm	0: **ebreak** instructions in M-mode behave as described in the Privileged Spec. 1: **ebreak** instructions in M-mode enter Debug Mode.	R/W	0
ebreaks	0: **ebreak** instructions in S-mode behave as described in the Privileged Spec. 1: **ebreak** instructions in S-mode enter Debug Mode.	R/W	0
ebreaku	0: **ebreak** instructions in U-mode behave as described in the Privileged Spec. 1: **ebreak** instructions in U-mode enter Debug Mode.	R/W	0
stepie	0: Interrupts are disabled during single stepping. 1: Interrupts are enabled during single stepping. Implementations may hard wire this bit to 0. In that case interrupt behavior can be emulated by the debugger. The debugger must not change the value of this bit while the hart is running.	WARL	0

ID REFERENCE TYPE DEFINITION

RVD.4.61 4.8.1 (p.43) R

Field	Description	Access	Reset
stopcount	0: Increment counters as usual. 1: Don't increment any counters while in Debug Mode or on **ebreak** instructions that cause entry into Debug Mode. These counters include the `cycle` and `instret` CSRs. This is preferred for most debugging scenarios. An implementation may hardwire this bit to 0 or 1.	WARL	Preset
stoptime	0: Increment timers as usual. 1: Don't increment any hart-local timers while in Debug Mode. An implementation may hardwire this bit to 0 or 1.	WARL	Preset
cause	Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, hardware should set cause to the cause with the highest priority. 1: An **ebreak** instruction was executed. (priority 3) 2: The Trigger Module caused a breakpoint exception. (priority 4, highest) 3: The debugger requested entry to Debug Mode using haltreq. (priority 1) 4: The hart single stepped because step was set. (priority 0, lowest) 5: The hart halted directly out of reset due to resethaltreq. It is also acceptable to report 3 when this happens. (priority 2) Other values are reserved for future use.	R	0
mprven	0: MPRV in `mstatus` is ignored in Debug Mode. 1: MPRV in `mstatus` takes effect in Debug Mode. Implementing this bit is optional. It may be tied to either 0 or 1.	WARL	Preset
nmip	When set, there is a Non-Maskable-Interrupt (NMI) pending for the hart. Since an NMI can indicate a hardware error condition, reliable debugging may no longer be possible once this bit becomes set. This is implementation dependent.	R	0

RVD.4.62 4.8.1 (p.44) R

Field	Description	Access	Reset
step	When set and not in Debug Mode, the hart will only execute a single instruction and then enter Debug Mode. If the instruction does not complete due to an exception, the hart will immediately enter Debug Mode before executing the trap handler, with appropriate exception registers set. The debugger must not change the value of this bit while the hart is running.	R/W	0
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is described in Table 4.5. A debugger can change this value to change the hart's privilege level when exiting Debug Mode. Not all privilege levels are supported on all harts. If the encoding written is not supported or the debugger is not allowed to change to it, the hart may change to any supported privilege level.	R/W	3

RVD.4.63 4.8.2 (p.44) H Debug PC (`dpc`, at 0x7b1)

RVD.4.64 4.8.2 (p.44) R Upon entry to debug mode, `dpc` is updated with the virtual address of the next instruction to be executed.

ID	REFERENCE	TYPE	DEFINITION
RVD.4.65	4.8.2 (p.44) Table 4.3	R	The behavior is described in more detail in the table below: Virtual address in DPC upon Debug Mode Entry

Cause	Virtual Address in DPC
ebreak	Address of the ebreak instruction
single step	Address of the instruction that would be executed next if no debugging was going on. Ie. pc + 4 for 32-bit instructions that don't change program flow, the destination PC on taken jumps/branches, etc.
trigger module	If `timing` is 0, the address of the instruction which caused the trigger to re. If `timing` is 1, the address of the next instruction to be executed at the time that debug mode was entered.
halt request	Address of the next instruction to be executed at the time that debug mode was entered

ID	REFERENCE	TYPE	DEFINITION
RVD.4.66	4.8.2 (p.44)	R	When resuming, the hart's PC is updated to the virtual address stored in `dpc`.
RVD.4.67	4.8.2 (p.44)	R	A debugger may write `dpc` to change where the hart resumes.
RVD.4.68	4.8.2 (p.45)	R	

DXLEN-1	...	0
	dpc	

ID	REFERENCE	TYPE	DEFINITION
RVD.4.69	4.8.3 (p.45)	H	Debug Scratch Register 0 (`dscratch0`, at 0x7b2)
RVD.4.70	4.8.3 (p.45)	R	Optional scratch register that can be used by implementations that need it.
RVD.4.71	4.8.3 (p.45)	R	A debugger must not write to this register unless `hartinfo` explicitly mentions it (the Debug Module may use this register internally).
RVD.4.72	4.8.4 (p.45)	H	Debug Scratch Register 1 (`dscratch1`, at 0x7b3)
RVD.4.73	4.8.4 (p.45)	R	Optional scratch register that can be used by implementations that need it.
RVD.4.74	4.8.4 (p.45)	R	A debugger must not write to this register unless `hartinfo` explicitly mentions it (the Debug Module may use this register internally).
RVD.4.75	4.9 (p.45)	H	Virtual Debug Registers
RVD.4.76	4.9 (p.45)	R	A virtual register is one that doesn't exist directly in the hardware, but that the debugger exposes as if it does.
RVD.4.77	4.9 (p.45)	R	Debug software should implement them, but hardware can skip this section.
RVD.4.78	4.9 (p.45)	R	Virtual registers exist to give users access to functionality that's not part of standard debuggers without requiring them to carefully modify debug registers while the debugger is also accessing those same registers.
RVD.4.79	4.9 (p.45) Table 4.4	R	Virtual Core Debug Registers

Address	Name
virtual	Privilege Level (`priv`)

ID	REFERENCE	TYPE	DEFINITION
RVD.4.80	4.9.1 (p.45)	H	Privilege Level (`priv`, at virtual)
RVD.4.81	4.9.1 (p.45)	R	Users can read this register to inspect the privilege level that the hart was running in when the hart halted.
RVD.4.82	4.9.1 (p.45)	R	Users can write this register to change the privilege level that the hart will run in when it resumes.

ID	REFERENCE	TYPE	DEFINITION
RVD.4.83	4.9.1 (p.45)	R	This register contains prv from `dcsr`, but in a place that the user is expected to access.
RVD.4.84	4.9.1 (p.45)	R	The user should not access `dcsr` directly, because doing so might interfere with the debugger.
RVD.4.85	4.9.1 (p.45)	R	(see register layout below)
RVD.4.86	4.9.1 (p.46) Table 4.5	R	Privilege Level Encoding
RVD.4.87	4.9.1 (p.46)	R	(see field table below)

RVD.4.85:

1	0
prv	

RVD.4.86 — Privilege Level Encoding:

Encoding	Privilege Level
0	User/Application
1	Supervisor
3	Machine

RVD.4.87:

Field	Description	Access	Reset
prv	Contains the privilege level the hart was operating in when Debug Mode was entered. The encoding is described in Table 4.5, and matches the privilege level encoding from the Privileged Spec. A user can write this value to change the hart's privilege level when exiting Debug Mode.	R/W	0

CHAPTER 5 Trigger Module

ID	REFERENCE	TYPE	DEFINITION
RVD.5.1	5.0 (p.47)	H	Trigger Module
RVD.5.2	5.0 (p.47)	I	Triggers can cause a breakpoint exception, entry into Debug Mode, or a trace action without having to execute a special instruction. This makes them invaluable when debugging code from ROM.
RVD.5.3	5.0 (p.47)	R	They can trigger on execution of instructions at a given memory address, or on the address/data in loads/stores.
RVD.5.4	5.0 (p.47)	R	These are all features that can be useful without having the Debug Module present, so the Trigger Module is broken out as a separate piece that can be implemented separately.
RVD.5.5	5.0 (p.47)	R	A hart can be compliant with this specification without implementing any trigger functionality at all, but if it is implemented then it must conform to this section.
RVD.5.6	5.0 (p.47)	R	Triggers do not fire while in Debug Mode.
RVD.5.7	5.0 (p.47)	I	Each trigger may support a variety of features.
RVD.5.8	5.0 (p.47)	R	A debugger can build a list of all triggers and their features as follows: 1. Write 0 to `tselect`. 2. Read back `tselect` and check that it contains the written value. If not, exit the loop. 3. Read `tinfo`. 4. If that caused an exception, the debugger must read `tdata1` to discover the type. (If type is 0, this trigger doesn't exist. Exit the loop.) 5. If info is 1, this trigger doesn't exist. Exit the loop. 6. Otherwise, the selected trigger supports the types discovered in info. 7. Repeat, incrementing the value in `tselect`.
RVD.5.9	5.0 (p.47)	C	The above algorithm reads back `tselect` so that implementations which have 2^n triggers only need to implement n bits of `tselect`.
RVD.5.10	5.0 (p.47)	C	The algorithm checks tinfo and type in case the implementation has m bits of `tselect` but fewer than 2^m triggers.
RVD.5.11	5.0 (p.47)	R	It is possible for a trigger with the "enter Debug Mode" action (1) and another trigger with the "raise a breakpoint exception" action (0) to fire at the same time.
RVD.5.12	5.0 (p.47)	R	The preferred behavior is to have both actions take place.
RVD.5.13	5.0 (p.47)	O	It is implementation-dependent which of the two happens first.
RVD.5.14	5.0 (p.47)	I	This ensures both that the presence of an external debugger doesn't affect execution and that a trigger set by user code doesn't affect the external debugger.
RVD.5.15	5.0 (p.47)	R	If this is not implemented, then the hart must enter Debug Mode and ignore the breakpoint exception.
RVD.5.16	5.0 (p.47, p.48)	R	In the latter case, hit of the trigger whose action is 0 must still be set, giving a debugger an opportunity to handle this case.
RVD.5.17	5.0 (p.48)	R	What happens with trace actions when triggers with different actions are also firing is left to the trace specification.
RVD.5.18	5.1 (p.48)	H	Native M-Mode Triggers

ID	REFERENCE	TYPE	DEFINITION
RVD.5.19	5.1 (p.48)	I	Triggers can be used for native debugging.
RVD.5.20	5.1 (p.48)	R	On a fully featured system triggers will be set using u or s, and when firing they can cause a breakpoint exception to trap to a more privileged mode.
RVD.5.21	5.1 (p.48)	R	It is possible to set triggers natively to fire in M mode as well.
RVD.5.22	5.1 (p.48)	R	In that case there is no higher privilege mode to trap to.
RVD.5.23	5.1 (p.48)	R	When such a trigger causes a breakpoint exception while already in a trap handler, this will leave the system unable to resume normal execution.
RVD.5.24	5.1 (p.48)	R	On full-featured systems this is a remote corner case that can probably be ignored.
RVD.5.25	5.1 (p.48)	R	On systems that only implement M mode, however, it is recommended to implement one of two solutions to this problem.
RVD.5.26	5.1 (p.48)	I	This way triggers can be useful for native debugging of even M mode code.
RVD.5.27	5.1 (p.48)	R	The simple solution is to have the hardware prevent triggers with action=0 from firing while in M mode and while MIE in mstatus is 0. Its limitation is that interrupts might be disabled at other times when a user might want triggers to fire.
RVD.5.28	5.1 (p.48)	R	A more complex solution is to implement mte and mpte in tcontrol. This solution has the benefit that it only disables triggers during the trap handler.
RVD.5.29	5.1 (p.48)	R	A user setting M mode triggers that cause breakpoint exceptions will have to be aware of any problems that might come up with the particular system they are working on.
RVD.5.30	5.2 (p.48)	H	Trigger Registers
RVD.5.31	5.2 (p.48)	R	These registers are CSRs, accessible using the RISC-V `csr` opcodes and optionally also using abstract debug commands.
RVD.5.32	5.2 (p.48)	I	Most trigger functionality is optional.
RVD.5.33	5.2 (p.48)	R	All `tdata` registers follow write-any-read-legal semantics.
RVD.5.34	5.2 (p.48)	R	If a debugger writes an unsupported configuration, the register will read back a value that is supported (which may simply be a disabled trigger).
RVD.5.35	5.2 (p.48)	R	This means that a debugger must always read back values it writes to `tdata` registers, unless it already knows already what is supported.
RVD.5.36	5.2 (p.48)	R	Writes to one `tdata` register may not modify the contents of other `tdata` registers, nor the configuration of any trigger besides the one that is currently selected.
RVD.5.37	5.2 (p.48)	R	The trigger registers are only accessible in machine and Debug Mode to prevent untrusted user code from causing entry into Debug Mode without the OS's permission.
RVD.5.38	5.2 (p.48)	R	In this section XLEN means MXLEN when in M-mode, and DXLEN when in Debug Mode.
RVD.5.39	5.2 (p.48)	R	Note that this makes several of the fields in `tdata1` move around based on the current execution mode and value of MXLEN.

ID	REFERENCE	TYPE	DEFINITION
RVD.5.40	5.2 (p.49) Table 5.1	R	action encoding

Value	Description
0	Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.)
1	Enter Debug Mode. (Only supported when the trigger's dmode is 1.)
2 - 5	Reserved for use by the trace specification.
other	Reserved for future use.

RVD.5.41	5.2 (p.49) Table 5.2	R	Trigger Registers

Address	Name
`0x7a0`	Trigger Select (`tselect`)
`0x7a1`	Trigger Data 1 (`tdata1`)
`0x7a1`	Match Control (`mcontrol`)
`0x7a1`	Instruction Count (`icount`)
`0x7a1`	Interrupt Trigger (`itrigger`)
`0x7a1`	Exception Trigger (`etrigger`)
`0x7a2`	Trigger Data 2 (`tdata2`)
`0x7a3`	Trigger Data 3 (`tdata3`)
`0x7a3`	Trigger Extra (RV32) (`textra32`)
`0x7a3`	Trigger Extra (RV64) (`textra64`)
`0x7a4`	Trigger Info (`tinfo`)
`0x7a5`	Trigger Control (`tcontrol`)
`0x7a8`	Machine Context (`mcontext`)
`0x7aa`	Supervisor Context (`scontext`)

RVD.5.42	5.2.1 (p.49)	H	Trigger Select (`tselect`, at 0x7a0)
RVD.5.43	5.2.1 (p.49)	I	This register determines which trigger is accessible through the other trigger registers.
RVD.5.44	5.2.1 (p.49)	R	The set of accessible triggers must start at 0, and be contiguous.
RVD.5.45	5.2.1 (p.49)	R	Writes of values greater than or equal to the number of supported triggers may result in a different value in this register than what was written.
RVD.5.46	5.2.1 (p.49)	R	To verify that what they wrote is a valid index, debuggers can read back the value and check that `tselect` holds what they wrote.
RVD.5.47	5.2.1 (p.49)	R	Since triggers can be used both by Debug Mode and M-mode, the debugger must restore this register if it modifies it.
RVD.5.48	5.2.1 (p.49)	R	

XLEN-1	...	0
index		

RVD.5.49	5.2.2 (p.50)	H	Trigger Data 1 (`tdata1`, at 0x7a1)
RVD.5.50	5.2.2 (p.50)	R	

XLEN-1	...	XLEN-4	XLEN-5	XLEN-6	...	0
type			dmode	data		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.51	5.2.2 (p.50)	R	

Field	Description	Access	Reset
type	0: There is no trigger at this `tselect`. 1: The trigger is a legacy SiFive address match trigger. These should not be implemented and aren't further documented here. 2: The trigger is an address/data match trigger. The remaining bits in this register act as described in `mcontrol`. 3: The trigger is an instruction count trigger. The remaining bits in this register act as described in `icount`. 4: The trigger is an interrupt trigger. The remaining bits in this register act as described in `itrigger`. 5: The trigger is an exception trigger. The remaining bits in this register act as described in `etrigger`. 15: This trigger exists (so enumeration shouldn't terminate), but is not currently available. Other values are reserved for future use.	R/W	Preset
dmode	0: Both Debug and M-mode can write the `tdata` registers at the selected `tselect`. 1: Only Debug Mode can write the `tdata` registers at the selected `tselect`. Writes from other modes are ignored. This bit is only writable from Debug Mode.	R/W	0
data	Trigger-specific data.	R/W	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.5.52	5.2.3 (p.50)	H	Trigger Data 2 (`tdata2`, at 0x7a2)
RVD.5.53	5.2.3 (p.50)	I	Trigger-specific data.
RVD.5.54	5.2.3 (p.50)	R	If XLEN is less than DXLEN, writes to this register are sign-extended.
RVD.5.55	5.2.3 (p.50)	R	

XLEN-1	...	0
data		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.56	5.2.4 (p.51)	H	Trigger Data 3 (`tdata3`, at 0x7a3)
RVD.5.57	5.2.4 (p.51)	I	Trigger-specific data.
RVD.5.58	5.2.4 (p.51)	R	If XLEN is less than DXLEN, writes to this register are sign-extended.
RVD.5.59	5.2.4 (p.51)	R	

XLEN-1	...	0
data		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.60	5.2.5 (p.51)	H	Trigger Info (`tinfo`, at 0x7a4)
RVD.5.61	5.2.5 (p.51)	R	This entire register is read-only.
RVD.5.62	5.2.5 (p.51)	R	

XLEN-1	...	16	15	...	0
0			info		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.63	5.2.5 (p.51)	R	

Field	Description	Access	Reset
info	One bit for each possible type enumerated in `tdata1`. Bit N corresponds to type N. If the bit is set, then that type is supported by the currently selected trigger. If the currently selected trigger doesn't exist, this field contains 1. If type is not writable, this register may be unimplemented, in which case reading it causes an illegal instruction exception. In this case the debugger can read the only supported type from `tdata1`.	R	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.5.64	5.2.6 (p.51)	H	Trigger Control (`tcontrol`, at 0x7a5)

ID	REFERENCE	TYPE	DEFINITION
RVD.5.65	5.2.6 (p.51)	R	This optional register is one solution to a problem regarding triggers with action=0 firing in M-mode trap handlers.
RVD.5.66	5.2.6 (p.51)	I	See Section 5.1 for more details.
RVD.5.67	5.2.6 (p.51)	R	

XLEN-1	...	8	7	6	...	4	3	2	...	0
0			mpte	0			mte	0		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.68	5.2.6 (p.52)	R	

Field	Description	Access	Reset
mpte	M-mode previous trigger enable field. When a trap into M-mode is taken, mpte is set to the value of mte.	R/W	0
mte	M-mode trigger enable field. 0: Triggers with action=0 do not match/fire while the hart is in M-mode. 1: Triggers do match/fire while the hart is in M-mode. When a trap into M-mode is taken, mte is set to 0. When **mret** is executed, mte is set to the value of mpte.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.5.69	5.2.7 (p.52)	H	Machine Context (`mcontext`, at 0x7a8)
RVD.5.70	5.2.7 (p.52)	R	This register is only writable in M mode and Debug Mode.
RVD.5.71	5.2.7 (p.52)	R	

XLEN-1	...	0
mcontext		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.72	5.2.7 (p.52)	R	

Field	Description	Access	Reset
mcontext	Machine mode software can write a context number to this register, which can be used to set triggers that only fire in that specific context. An implementation may tie any number of upper bits in this field to 0. It's recommended to implement no more than 6 bits on RV32, and 13 on RV64.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.5.73	5.2.8 (p.52)	H	Supervisor Context (`scontext`, at 0x7aa)
RVD.5.74	5.2.8 (p.52)	R	This register is only writable in S mode, M mode and Debug Mode.
RVD.5.75	5.2.8 (p.52)	R	

XLEN-1	...	0
data		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.76	5.2.8 (p.53)	R	

Field	Description	Access	Reset
data	Supervisor mode software can write a context number to this register, which can be used to set triggers that only fire in that specific context. An implementation may tie any number of high bits in this field to 0. It's recommended to implement no more than 16 bits on RV32, and 34 on RV64.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.5.77	5.2.9 (p.53)	H	Match Control (`mcontrol`, at 0x7a1)
RVD.5.78	5.2.9 (p.53)	R	This register is accessible as `tdata1` when type is 2.
RVD.5.79	5.2.9 (p.53)	I	Address and data trigger implementation are heavily dependent on how the processor core is implemented.
RVD.5.80	5.2.9 (p.53)	O	To accommodate various implementations, execute, load, and store address/data triggers may fire at whatever point in time is most convenient for the implementation.
RVD.5.81	5.2.9 (p.53)	R	The debugger may request specific timings as described in timing.

ID	REFERENCE	TYPE	DEFINITION
RVD.5.82	5.2.9 (p.53) Table 5.8	O	The table below suggests timings for the best user experience. Suggested Breakpoint Timings

Match Type	Suggested Trigger Timing
Execute Address	Before
Execute Instruction	Before
Execute Address+Instruction	Before
Load Address	Before
Load Data	After
Load Address+Data	After
Store Address	Before
Store Data	Before
Store Address+Data	Before

ID	REFERENCE	TYPE	DEFINITION
RVD.5.83	5.2.9 (p.53)	O	This trigger type may be limited to address comparisons (select is always 0) only.
RVD.5.84	5.2.9 (p.53)	R	If that is the case, then `tdata2` must be able to hold all valid virtual addresses but it need not be capable of holding other values.
RVD.5.85	5.2.9 (p.53)	R	

XLEN-1	...	XLEN-4	XLEN-5	XLEN-6	...	XLEN-11	XLEN-12	...	23
type			dmode	maskmax			0		

22	21	20	19	18	17	16
sizehi		hit	select	timing	sizelo	

15	...	12	11	10	...	7	6	5	4	3	2	1	0
action			chain	match			m	0	s	u	execute	store	load

ID	REFERENCE	TYPE	DEFINITION
RVD.5.86	5.2.9 (p.54)	R	

Field	Description	Access	Reset
maskmax	Specifies the largest naturally aligned powers-of-two (NAPOT) range supported by the hardware when match is 1. The value is the logarithm base 2 of the number of bytes in that range. A value of 0 indicates that only exact value matches are supported (one byte range). A value of 63 corresponds to the maximum NAPOT range, which is 2^{63} bytes in size.	R	Preset
sizehi	This field only exists if XLEN is greater than 32. In that case it extends size. If it does not exist then hardware operates as if the field contains 0.	R/W	0
hit	If this optional bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	R/W	0
select	0: Perform a match on the virtual address. 1: Perform a match on the data value loaded or stored, or the instruction executed.	R/W	0

ID **REFERENCE** **TYPE** **DEFINITION**

RVD.5.87 5.2.9 (p.55) R

Field	Description	Access	Reset
timing	0: The action for this trigger will be taken just before the instruction that triggered it is executed, but after all preceding instructions are committed. 1: The action for this trigger will be taken after the instruction that triggered it is executed. It should be taken before the next instruction is executed, but it is better to implement triggers and not implement that suggestion than to not implement them at all. Most hardware will only implement one timing or the other, possibly dependent on select, execute, load, and store. This bit primarily exists for the hardware to communicate to the debugger what will happen. Hardware may implement the bit fully writable, in which case the debugger has a little more control. Data load triggers with timing of 0 will result in the same load happening again when the debugger lets the hart run. For data load triggers, debuggers must first attempt to set the breakpoint with timing of 1. A chain of triggers that don't all have the same timing value will never fire (unless consecutive instructions match the appropriate triggers). If a trigger with timing of 0 matches, it is implementation-dependent whether that prevents a trigger with timing of 1 matching as well.	R/W	0

RVD.5.88 5.2.9 (p.56) R

Field	Description	Access	Reset
sizelo	This field contains the 2 low bits of size. The high bits come from sizehi. The combined value is interpreted as follows: 0: The trigger will attempt to match against an access of any size. The behavior is only well defined if select = 0, or if the access size is XLEN. 1: The trigger will only match against 8-bit memory accesses. 2: The trigger will only match against 16-bit memory accesses or execution of 16-bit instructions. 3: The trigger will only match against 32-bit memory accesses or execution of 32-bit instructions. 4: The trigger will only match against execution of 48-bit instructions. 5: The trigger will only match against 64-bit memory accesses or execution of 64-bit instructions. 6: The trigger will only match against execution of 80-bit instructions. 7: The trigger will only match against execution of 96-bit instructions. 8: The trigger will only match against execution of 112-bit instructions. 9: The trigger will only match against 128-bit memory accesses or execution of 128-bit instructions.	R/W	0
action	The action to take when the trigger fires. The values are explained in Table 5.1.	R/W	0

ID **REFERENCE** **TYPE** **DEFINITION**

RVD.5.89 5.2.9 (p.57) R

Field	Description	Access	Reset
chain	0: When this trigger matches, the configured action is taken. 1: While this trigger does not match, it prevents the trigger with the next index from matching. A trigger chain starts on the first trigger with chain = 1 after a trigger with chain = 0, or simply on the first trigger if that has chain = 1. It ends on the first trigger after that which has chain = 0. This final trigger is part of the chain. The action on all but the final trigger is ignored. The action on that final trigger will be taken if and only if all the triggers in the chain match at the same time. Because chain affects the next trigger, hardware must zero it in writes to `mcontrol` that set dmode to 0 if the next trigger has dmode of 1. In addition hardware should ignore writes to `mcontrol` that set dmode to 1 if the previous trigger has both dmode of 0 and chain of 1. Debuggers must avoid the latter case by checking chain on the previous trigger if they're writing `mcontrol`. Implementations that wish to limit the maximum length of a trigger chain (eg. to meet timing requirements) may do so by zeroing chain in writes to `mcontrol` that would make the chain too long.	R/W	0
match	0: Matches when the value equals `tdata2`. 1: Matches when the top M bits of the value match the top M bits of `tdata2`. M is XLEN-1 minus the index of the least-significant bit containing 0 in `tdata2`. 2: Matches when the value is greater than (unsigned) or equal to `tdata2`. 3: Matches when the value is less than (unsigned) `tdata2`. 4: Matches when the lower half of the value equals the lower half of `tdata2` after the lower half of the value is ANDed with the upper half of `tdata2`. 5: Matches when the upper half of the value equals the lower half of `tdata2` after the upper half of the value is ANDed with the upper half of `tdata2`. Other values are reserved for future use.	R/W	0
m	When set, enable this trigger in M-mode.	R/W	0
s	When set, enable this trigger in S-mode.	R/W	0
u	When set, enable this trigger in U-mode.	R/W	0

RVD.5.90 5.2.9 (p.58) R

Field	Description	Access	Reset
execute	When set, the trigger fires on the virtual address or opcode of an instruction that is executed.	R/W	0
store	When set, the trigger fires on the virtual address or data of a store.	R/W	0
load	When set, the trigger fires on the virtual address or data of a load.	R/W	0

RVD.5.91 5.2.10 (p.58) H Instruction Count (`icount`, at 0x7a1)

RVD.5.92 5.2.10 (p.58) R This register is accessible as `tdata1` when type is 3.

ID	REFERENCE	TYPE	DEFINITION
RVD.5.93	5.2.10 (p.58)	C	This trigger type is intended to be used as a single step that's useful both for external debuggers and for software monitor programs. For that case it is not necessary to support count greater than 1. The only two combinations of the mode bits that are useful in those scenarios are u by itself, or m, s, and u all set.
RVD.5.94	5.2.10 (p.58)	C	If the hardware limits count to 1, and changes mode bits instead of decrementing count, this register can be implemented with just 2 bits. One for u, and one for m and s tied together. If only the external debugger or only a software monitor needs to be supported, a single bit is enough.
RVD.5.95	5.2.10 (p.58)	R	

XLEN-1	...	XLEN-4	XLEN-5	XLEN-6	...	25	24	23	...	10
type			dmode	0			hit	count		

9	8	7	6	5	...	0
m	0	s	u	action		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.96	5.2.10 (p.58, p.59)	R	

Field	Description	Access	Reset
hit	If this optional bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	R/W	0
count	When count is decremented to 0, the trigger fires. Instead of changing count from 1 to 0, it is also acceptable for hardware to clear m, s, and u. This allows count to be hard-wired to 1 if this register just exists for single step.	R/W	1
m	When set, every instruction completed or exception taken in M-mode decrements count by 1.	R/W	0
s	When set, every instruction completed or exception taken in S-mode decrements count by 1.	R/W	0
u	When set, every instruction completed or exception taken in U-mode decrements count by 1.	R/W	0
action	The action to take when the trigger fires. The values are explained in Table 5.1.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.5.97	5.2.11 (p.59)	H	Interrupt Trigger (`itrigger`, at 0x7a1)
RVD.5.98	5.2.11 (p.59)	R	This register is accessible as `tdata1` when type is 4.
RVD.5.99	5.2.11 (p.59)	R	This trigger may fire on any of the interrupts configurable in `mie` (described in the Privileged Spec).
RVD.5.100	5.2.11 (p.59)	R	The interrupts to fire on are configured by setting the same bit in `tdata2` as would be set in `mie` to enable the interrupt.
RVD.5.101	5.2.11 (p.59)	O	Hardware may only support a subset of interrupts for this trigger.
RVD.5.102	5.2.11 (p.59)	R	A debugger must read back `tdata2` after writing it to confirm the requested functionality is actually supported.
RVD.5.103	5.2.11 (p.59)	R	The trigger only fires if the hart takes a trap because of the interrupt. (E.g. it does not fire when a timer interrupt occurs but that interrupt is not enabled in `mie`.)
RVD.5.104	5.2.11 (p.59)	R	When the trigger fires, all CSRs are updated as defined by the Privileged Spec, and the requested action is taken just before the first instruction of the interrupt/exception handler is executed.

ID	REFERENCE	TYPE	DEFINITION
RVD.5.105	5.2.11 (p.59)	R	

XLEN-1	...	XLEN-4	XLEN-5	XLEN-6	XLEN-7	...	10	9	8	7	6	5	...	0
type			dmode	hit	0			m	0	s	u	action		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.106	5.2.11 (p.59)	R	

Field	Description	Access	Reset
hit	If this optional bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	R/W	0
m	When set, enable this trigger for interrupts that are taken from M mode.	R/W	0
s	When set, enable this trigger for interrupts that are taken from S mode.	R/W	0
u	When set, enable this trigger for interrupts that are taken from U mode.	R/W	0
action	The action to take when the trigger fires. The values are explained in Table 5.1.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.5.107	5.2.12 (p.60)	H	Exception Trigger (`etrigger`, at 0x7a1)
RVD.5.108	5.2.12 (p.60)	R	This register is accessible as `tdata1` when type is 5.
RVD.5.109	5.2.12 (p.60)	R	This trigger may fire on up to XLEN of the Exception Codes defined in `mcause` (described in the Privileged Spec, with Interrupt=0).
RVD.5.110	5.2.12 (p.60)	R	Those causes are configured by writing the corresponding bit in `tdata2`. (E.g. to trap on an illegal instruction, the debugger sets bit 2 in `tdata2`.)
RVD.5.111	5.2.12 (p.60)	R	Hardware may support only a subset of exceptions.
RVD.5.112	5.2.12 (p.60)	R	A debugger must read back `tdata2` after writing it to confirm the requested functionality is actually supported.
RVD.5.113	5.2.12 (p.60)	R	When the trigger fires, all CSRs are updated as defined by the Privileged Spec, and the requested action is taken just before the first instruction of the interrupt/exception handler is executed.
RVD.5.114	5.2.12 (p.60)	R	

XLEN-1	...	XLEN-4	XLEN-5	XLEN-6	XLEN-7	...	10	9	8	7	6	5	...	0
type			dmode	hit	0			m	0	s	u	action		

ID	REFERENCE	TYPE	DEFINITION
RVD.5.115	5.2.12 (p.60)	R	

Field	Description	Access	Reset
hit	If this optional bit is implemented, the hardware sets it when this trigger matches. The trigger's user can set or clear it at any time. It is used to determine which trigger(s) matched. If the bit is not implemented, it is always 0 and writing it has no effect.	R/W	0
m	When set, enable this trigger for exceptions that are taken from M mode.	R/W	0
s	When set, enable this trigger for exceptions that are taken from S mode.	R/W	0
u	When set, enable this trigger for exceptions that are taken from U mode.	R/W	0
action	The action to take when the trigger fires. The values are explained in Table 5.1.	R/W	0

ID	REFERENCE	TYPE	DEFINITION
RVD.5.116	5.2.13 (p.60)	H	Trigger Extra (RV32) (`textra32`, at 0x7a3)
RVD.5.117	5.2.13 (p.60)	R	This register is accessible as `tdata3` when type is 2, 3, 4, or 5.
RVD.5.118	5.2.13 (p.60)	O	All functionality in this register is optional.
RVD.5.119	5.2.13 (p.60)	R	The value bits may tie any number of upper bits to 0.

ID	REFERENCE	TYPE	DEFINITION
RVD.5.120	5.2.13 (p.60)	R	The select bits may only support 0 (ignore).
RVD.5.121	5.2.13 (p.60)	R	(register layout below)
RVD.5.122	5.2.13 (p.61)	R	(field table below)
RVD.5.123	5.2.14 (p.61)	H	Trigger Extra (RV64) (`textra64`, at 0x7a3)
RVD.5.124	5.2.14 (p.61)	R	This is the layout of `textra` if XLEN is 64. The fields are defined above, in `textra32`.
RVD.5.125	5.2.14 (p.61)	R	(register layout below)

RVD.5.121:

31	...	26	25	24	...	18	17	...	2	1	0
mvalue			mselect	0			svalue			sselect	

RVD.5.122:

Field	Description	Access	Reset
mvalue	Data used together with mselect.	R/W	0
mselect	0: Ignore mvalue. 1: This trigger will only match if the low bits of `mcontext` equal mvalue.	WARL	0
svalue	Data used together with sselect.	R/W	0
sselect	0: Ignore svalue. 1: This trigger will only match if the low bits of `scontext` equal svalue. 2: This trigger will only match if ASID in `satp` equals the lower ASIDMAX (defined in the Privileged Spec) bits of svalue.	WARL	0

RVD.5.125:

63	...	51	50	49	...	36	35	...	2	1	0
mvalue			mselect	0			svalue			sselect	

CHAPTER 6 Debug Transport Module (DTM)

ID	REFERENCE	TYPE	DEFINITION
RVD.6.1	6.0 (p.62)	H	Debug Transport Module (DTM)
RVD.6.2	6.0 (p.62)	R	Debug Transport Modules provide access to the DM over one or more transports (e.g. JTAG or USB).
RVD.6.3	6.0 (p.62)	R	There may be multiple DTMs in a single platform.
RVD.6.4	6.0 (p.62)	R	Ideally every component that communicates with the outside world includes a DTM, allowing a platform to be debugged through every transport it supports.
RVD.6.5	6.0 (p.62)	I	For instance a USB component could include a DTM. This would trivially allow any platform to be debugged over USB.
RVD.6.6	6.0 (p.62)	I	All that is required is that the USB module already in use also has access to the Debug Module Interface.
RVD.6.7	6.0 (p.62)	R	Using multiple DTMs at the same time is not supported.
RVD.6.8	6.0 (p.62)	R	It is left to the user to ensure this does not happen.
RVD.6.9	6.0 (p.62)	I	This specification defines a JTAG DTM in Section 6.1.
RVD.6.10	6.0 (p.62)	I	Additional DTMs may be added in future versions of this specification.
RVD.6.11	6.0 (p.62)	I	An implementation can be compliant with this specification without implementing any of this section.
RVD.6.12	6.0 (p.62)	I	In that case it must be advertised as conforming to “RISC-V Debug Specification 0.13.2, with custom DTM."
RVD.6.13	6.0 (p.62)	I	If the JTAG DTM described here is implemented, it must be advertised as conforming to the “RISC-V Debug Specification 0.13.2, with JTAG DTM."
RVD.6.14	6.1 (p.62)	H	JTAG Debug Transport Module
RVD.6.15	6.1 (p.62)	I	This Debug Transport Module is based around a normal JTAG Test Access Port (TAP).
RVD.6.16	6.1 (p.62)	R	The JTAG TAP allows access to arbitrary JTAG registers by first selecting one using the JTAG instruction register (IR), and then accessing it through the JTAG data register (DR).
RVD.6.17	6.1.1 (p.62)	H	JTAG Background
RVD.6.18	6.1.1 (p.62)	I	JTAG refers to IEEE Std 1149.1-2013.
RVD.6.19	6.1.1 (p.62, p.63)	I	It is a standard that defines test logic that can be included in an integrated circuit to test the interconnections between integrated circuits, test the integrated circuit itself, and observe or modify circuit activity during the components normal operation.
RVD.6.20	6.1.1 (p.63)	R	This specification uses the latter functionality.
RVD.6.21	6.1.1 (p.63)	R	The JTAG standard defines a Test Access Port (TAP) that can be used to read and write a few custom registers, which can be used to communicate with debug hardware in a component.
RVD.6.22	6.1.2 (p.63)	H	JTAG DTM Registers
RVD.6.23	6.1.2 (p.63)	R	JTAG TAPs used as a DTM must have an IR of at least 5 bits.

ID	REFERENCE	TYPE	DEFINITION
RVD.6.24	6.1.2 (p.63)	R	When the TAP is reset, IR must default to 00001, selecting the IDCODE instruction.
RVD.6.25	6.1.2 (p.63) Table 6.1	R	A full list of JTAG registers along with their encoding is in the table below. JTAG DTM TAP Registers

Address	Name	Description
`0x00`	BYPASS	JTAG recommends this encoding
`0x01`	IDCODE	JTAG recommends this encoding
`0x10`	DTM Control and Status (`dtmcs`)	For Debugging
`0x11`	Debug Module Interface Access (`dmi`)	For Debugging
`0x12`	Reserved (BYPASS)	Reserved for future RISC-V debugging
`0x13`	Reserved (BYPASS)	Reserved for future RISC-V debugging
`0x14`	Reserved (BYPASS)	Reserved for future RISC-V debugging
`0x15`	Reserved (BYPASS)	Reserved for future RISC-V debugging
`0x16`	Reserved (BYPASS)	Reserved for future RISC-V debugging
`0x17`	Reserved (BYPASS)	Reserved for future RISC-V debugging
`0x1f`	BYPASS	JTAG requires this encoding

ID	REFERENCE	TYPE	DEFINITION
RVD.6.26	6.1.2 (p.63)	R	If the IR actually has more than 5 bits, then the encodings in Table 6.1 should be extended with 0's in their most significant bits.
RVD.6.27	6.1.2 (p.63)	R	The only regular JTAG registers a debugger might use are BYPASS and IDCODE, but this specification leaves IR space for many other standard JTAG instructions.
RVD.6.28	6.1.2 (p.63)	R	Unimplemented instructions must select the BYPASS register.
RVD.6.29	6.1.3 (p.63)	H	IDCODE (at 0x01)
RVD.6.30	6.1.3 (p.63)	R	This register is selected (in IR) when the TAP state machine is reset. Its definition is exactly as defined in IEEE Std 1149.1-2013.
RVD.6.31	6.1.3 (p.63)	R	This entire register is read-only.
RVD.6.32	6.1.3 (p.63)	R	

31	...	28	27	...	12	11	...	1	0
Version			PartNumber			ManufId			1

ID	REFERENCE	TYPE	DEFINITION
RVD.6.33	6.1.3 (p.63, p.64)	R	

Field	Description	Access	Reset
Version	Identifies the release version of this part. R Preset	R	Preset
PartNumber	Identifies the designer's part number of this part.	R	Preset
ManufId	Identifies the designer/manufacturer of this part. Bits 6:0 must be bits 6:0 of the designer/manufacturer's Identification Code as assigned by JEDEC Standard JEP106. Bits 10:7 contain the modulo-16 count of the number of continuation characters (0x7f) in that same Identification Code.	R	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.6.34	6.1.4 (p.64)	H	DTM Control and Status (`dtmcs`, at 0x10)
RVD.6.35	6.1.4 (p.64)	R	The size of this register will remain constant in future versions so that a debugger can always determine the version of the DTM.
RVD.6.36	6.1.4 (p.64)	R	

31	...	18	17	16	15	14	...	12	11	10	9	...	4	3	...	0
0			dmihardreset	dmireset	0	idle			dmistat		abits			version		

ID	REFERENCE	TYPE	DEFINITION
RVD.6.37	6.1.4 (p.64)	R	

Field	Description	Access	Reset
dmihardreset	Writing 1 to this bit does a hard reset of the DTM, causing the DTM to forget about any outstanding DMI transactions. In general this should only be used when the Debugger has reason to expect that the outstanding DMI transaction will never complete (e.g. a reset condition caused an inflight DMI transaction to be cancelled).	W1	-
dmireset	Writing 1 to this bit clears the sticky error state and allows the DTM to retry or complete the previous transaction.	W1	-
idle	This is a hint to the debugger of the minimum number of cycles a debugger should spend in Run-Test/Idle after every DMI scan to avoid a `busy' return code (dmistat of 3). A debugger must still check dmistat when necessary. 0: It is not necessary to enter Run-Test/Idle at all. 1: Enter Run-Test/Idle and leave it immediately. 2: Enter Run-Test/Idle and stay there for 1 cycle before leaving. And so on.	R	Preset

ID	REFERENCE	TYPE	DEFINITION
RVD.6.38	6.1.4 (p.65)	R	

Field	Description	Access	Reset
dmistat	0: No error. 1: Reserved. Interpret the same as 2. 2: An operation failed (resulted in op of 2). 3: An operation was attempted while a DMI access was still in progress (resulted in op of 3).	R	0
abits	The size of address in `dmi`.	R	Preset
version	0: Version described in spec version 0.11. 1: Version described in spec version 0.13. 15: Version not described in any available version of this spec.	R	1

ID	REFERENCE	TYPE	DEFINITION
RVD.6.39	6.1.5 (p.65)	H	Debug Module Interface Access (`dmi`, at 0x11)
RVD.6.40	6.1.5 (p.65)	R	This register allows access to the Debug Module Interface (DMI).
RVD.6.41	6.1.5 (p.65)	R	In Update-DR, the DTM starts the operation specified in op unless the current status reported in op is sticky.
RVD.6.42	6.1.5 (p.65)	R	In Capture-DR, the DTM updates data with the result from that operation, updating op if the current op isn't sticky.
RVD.6.43	6.1.5 (p.65)	R	See Section B.1 and Table ?? (original text) for examples of how this is used.
RVD.6.44	6.1.5 (p.65)	C	The still-in-progress status is sticky to accommodate debuggers that batch together a number of scans, which must all be executed or stop as soon as there's a problem. For instance a series of scans may write a Debug Program and execute it. If one of the writes fails but the execution continues, then the Debug Program may hang or have other unexpected side effects.
RVD.6.45	6.1.5 (p.65)	R	

abits+33	...	34	33	...	2	1	0
address			data			op	

ID	REFERENCE	TYPE	DEFINITION
RVD.6.46	6.1.5 (p.65, p.66)	R	

Field	Description	Access	Reset
address	Address used for DMI access. In Update-DR this value is used to access the DM over the DMI.	R/W	0
data	The data to send to the DM over the DMI during Update-DR, and the data returned from the DM as a result of the previous operation.	R/W	0
op	When the debugger writes this field, it has the following meaning: 0: Ignore data and address. (nop) Don't send anything over the DMI during Update-DR. This operation should never result in a busy or error response. The address and data reported in the following Capture-DR are undefined. 1: Read from address. (read) 2: Write data to address. (write) 3: Reserved. When the debugger reads this field, it means the following: 0: The previous operation completed successfully. 1: Reserved. 2: A previous operation failed. The data scanned into `dmi` in this access will be ignored. This status is sticky and can be cleared by writing dmireset in `dtmcs`. This indicates that the DM itself responded with an error. There are no specified cases in which the DM would respond with an error, and DMI is not required to support returning errors. 3: An operation was attempted while a DMI request is still in progress. The data scanned into `dmi` in this access will be ignored. This status is sticky and can be cleared by writing `dmireset` in `dtmcs`. If a debugger sees this status, it needs to give the target more TCK edges between Update-DR and Capture-DR. The simplest way to do that is to add extra transitions in Run-Test/Idle.	R/W	0

RVD.6.47	6.1.6 (p.66)	H	BYPASS (at 0x1f)
RVD.6.48	6.1.6 (p.66)	R	1-bit register that has no effect.
RVD.6.49	6.1.6 (p.66)	R	It is used when a debugger does not want to communicate with this TAP.
RVD.6.50	6.1.6 (p.66)	R	This entire register is read-only.
RVD.6.51	6.1.6 (p.66)	R	0 0
RVD.6.52	6.1.7 (p.67)	H	Recommended JTAG Connector
RVD.6.53	6.1.7 (p.67)	R	To make it easy to acquire debug hardware, this spec recommends a connector that is compatible with the MIPI-10 .05 inch connector specification, as described in the MIPI Alliance Recommendation for Debug and Trace Connectors, Version 1.10.00, 16 March 2011.
RVD.6.54	6.1.7 (p.67)	R	The connector has .05 inch spacing, gold-plated male header with .016 inch thick hardened copper or beryllium bronze square posts (SAMTEC FTSH or equivalent).
RVD.6.55	6.1.7 (p.67)	R	Female connectors are compatible 20μm gold connectors.

ID	**REFERENCE**	**TYPE**	**DEFINITION**
RVD.6.56	6.1.7 (p.67) Table 6.5	R	Viewing the male header from above (the pins pointing at your eye), a target's connector looks as it does in Table 6.5. MIPI-10 Connector Diagram

VREF DEBUG	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
GND or KEY	7	8	TDI
GND	9	10	nRESET

RVD.6.57	6.1.7 (p.67)	R	If a platform requires nTRST then it is permissible to reuse the nRESET pin as the nTRST signal.
RVD.6.58	6.1.7 (p.67)	R	If a platform requires both system reset and TAP reset, the MIPI-20 connector should be used.
RVD.6.59	6.1.7 (p.67)	R	Its physical connector is virtually identical to MIPI-10, except that it's twice as long, supporting twice as many pins.
RVD.6.60	6.1.7 (p.67) Table 6.6	R	Its connector is show in Table 6.6. MIPI-20 Connector Diagram

VREF DEBUG	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
GND or KEY	7	8	TDI
GND	9	10	nRESET
GND	11	12	RTCK
GND	13	14	nTRST_PD
GND	15	16	nTRST
GND	17	18	DBGRQ
GND	19	20	DBGACK

RVD.6.61	6.1.7 (p.67)	R	The same connectors can be used for 2-wire cJTAG.
RVD.6.62	6.1.7 (p.67)	R	In that case TMS is used for TMSC, and TCK is used for TCKC.

ID	REFERENCE	TYPE	DEFINITION
RVD.6.63	6.1.7 (p.67) Table 6.7	R	The function of each pin is described in Table 6.7. JTAG Connector Pinout

1	VREF DEBUG	Reference voltage for logic high.
2	TMS	JTAG TMS signal, driven by the debug adapter.
4	TCK	JTAG TCK signal, driven by the debug adapter.
6	TDO	JTAG TDO signal, driven by the target.
7	GND or KEY	This pin may be cut on the male and plugged on the female header to ensure the header is always plugged in correctly. It is, however, recommended to use this pin as an additional ground, to allow for fastest TCK speeds. A shrouded connector should be used to prevent the cable from being plugged in incorrectly.
8	TDI	JTAG TDI signal, driven by the debug adapter.
10	nRESET	Active-low reset signal, driven by the debug adapter. Asserting reset should reset any RISC-V cores as well as any other peripherals on the PCB. It should not reset the debug logic. This pin is optional but strongly encouraged. If necessary, this pin could be used as nTRST instead. nRESET should never be connected to the TAP reset, otherwise the debugger might not be able to debug through a reset to discover the cause of a crash or to maintain execution control after the reset.
12	RTCK	Return test clock, driven by the target. A target may relay the TCK signal here once it has processed it, allowing a debugger to adjust its TCK frequency in response.
14	nTRST_PD	Test reset pull-down (optional), driven by the debug adapter. Same function as nTRST, but with pull-down resistor on target.
16	nTRST	Test reset (optional), driven by the debug adapter. Used to reset the JTAG TAP Controller.
18	DBGRQ	Not used, driven low by the debug adapter.
20	DBGACK	Not used, driven by the target.

Appendix A Hardware Implementations

ID	REFERENCE	TYPE	DEFINITION
RVD.A.1	A.0 (p.69)	H	Hardware Implementations
RVD.A.2	A.0 (p.69)	I	Below are two possible implementations. A designer could choose one, mix and match, or come up with their own design.
RVD.A.3	A.1 (p.69)	H	Abstract Command Based
RVD.A.4	A.1 (p.69)	I	Halting happens by stalling the hart execution pipeline.
RVD.A.5	A.1 (p.69)	I	Muxes on the register file(s) allow for accessing GPRs and CSRs using the Access Register abstract command.
RVD.A.6	A.1 (p.69)	I	Memory is accessed using the Abstract Access Memory command or through System Bus Access.
RVD.A.7	A.1 (p.69)	I	This implementation could allow a debugger to collect information from the hart even when that hart is unable to execute instructions.
RVD.A.8	A.1 (p.69)	H	Execution Based
RVD.A.9	A.2 (p.69)	I	This implementation only implements the Access Register abstract command for GPRs on a halted hart, and relies on the Program Buffer for all other operations. It uses the hart's existing pipeline and ability to execute from arbitrary memory locations to avoid modifications to a hart's datapath.
RVD.A.10	A.2 (p.69)	I	When the halt request bit is set, the Debug Module raises a special interrupt to the selected harts. This interrupt causes each hart to enter Debug Mode and jump to a defined memory region that is serviced by the DM. When taking this exception, `pc` is saved to `dpc` and `cause` is updated in `dcsr`.
RVD.A.11	A.2 (p.69)	I	The code in the Debug Module causes the hart to execute a “park loop." In the park loop the hart writes its mhartid to a memory location within the Debug Module to indicate that it is halted. To allow the DM to individually control one out of several halted harts, each hart polls for flags in a DM-controlled memory location to determine whether the debugger wants it to execute the Program Buffer or perform a resume.
RVD.A.12	A.2 (p.70)	I	To execute an abstract command, the DM first populates some internal words of program buffer according to `command`. When transfer is set, the DM populates these words with **lw <gpr>, 0x400(zero)** or **sw 0x400(zero), <gpr>**. 64- and 128-bit accesses use **ld**/**sd** and **lq**/**sq** respectively. If `transfer` is not set, the DM populates these instructions as nops. If execute is set, execution continues to the debugger-controlled Program Buffer, otherwise the DM causes a **ebreak** to execute immediately.
RVD.A.13	A.2 (p.70)	I	When **ebreak** is executed (indicating the end of the Program Buffer code) the hart returns to its park loop. If an exception is encountered, the hart jumps to a debug exception address within the Debug Module. The code at that address causes the hart to write to an address in the Debug Module which indicates exception. This address is considered I/O for fence instructions (see #9 on page 39). Then the hart jumps back to the park loop. The DM infers from the write that there was an exception, and sets cmderr appropriately.

ID	**REFERENCE**	**TYPE**	**DEFINITION**
RVD.A.14	A.2 (p.70)	I	To resume execution, the debug module sets a flag which causes the hart to execute a `dret`. When `dret` is executed, `pc` is restored from `dpc` and normal execution resumes at the privilege set by prv.
RVD.A.15	A.2 (p.70)	I	`data0` etc. are mapped into regular memory at an address relative to zero with only a 12-bit `imm`. The exact address is an implementation detail that a debugger must not rely on. For example, the data registers might be mapped to 0x400.
RVD.A.16	A.2 (p.70)	I	For additional flexibility, `progbuf0`, etc. are mapped into regular memory immediately preceding `data0`, in order to form a contiguous region of memory which can be used for either program execution or data transfer.

Appendix B Debugger Implementation

ID	REFERENCE	TYPE	DEFINITION
RVD.B.1	B.0 (p.71)	H	Debugger Implementation
RVD.B.2	B.0 (p.71)	I	This section details how an external debugger might use the described debug interface to perform some common operations on RISC-V cores using the JTAG DTM described in Section 6.1. All these examples assume a 32-bit core but it should be easy to adapt the examples to 64- or 128-bit cores.
RVD.B.3	B.0 (p.71)	I	To keep the examples readable, they all assume that everything succeeds, and that they complete faster than the debugger can perform the next access. This will be the case in a typical JTAG setup. However, the debugger must always check the sticky error status bits after performing a sequence of actions. If it sees any that are set, then it should attempt the same actions again, possibly while adding in some delay, or explicit checks for status bits.
RVD.B.4	B.1 (p.71)	H	Debug Module Interface Access
RVD.B.5	B.1 (p.71)	I	To read an arbitrary Debug Module register, select `dmi`, and scan in a value with op set to 1, and address set to the desired register address. In Update-DR the operation will start, and in Capture-DR its results will be captured into `data`. If the operation didn't complete in time, op will be 3 and the value in `data` must be ignored. The busy condition must be cleared by writing dmireset in `dtmcs`, and then the second scan scan must be performed again. This process must be repeated until op returns 0. In later operations the debugger should allow for more time between Capture-DR and Update-DR.
RVD.B.6	B.1 (p.71)	I	To write an arbitrary Debug Bus register, select `dmi`, and scan in a value with op set to 2, and address and data set to the desired register address and data respectively. From then on everything happens exactly as with a read, except that a write is performed instead of the read.
RVD.B.7	B.1 (p.71)	I	It should almost never be necessary to scan IR, avoiding a big part of the ineffciency in typical JTAG use.
RVD.B.8	B.2 (p.72)	H	Checking for Halted Harts
RVD.B.9	B.2 (p.72)	I	A user will want to know as quickly as possible when a hart is halted (e.g. due to a breakpoint). To effciently determine which harts are halted when there are many harts, the debugger uses the `haltsum` registers. Assuming the maximum number of harts exist, first it checks `haltsum3`. For each bit set there, it writes hartsel, and checks `haltsum2`. This process repeats through `haltsum1` and `haltsum0`. Depending on how many harts exist, the process should start at one of the lower haltsum registers.
RVD.B.10	B.3 (p.72)	H	Halting
RVD.B.11	B.3 (p.72)	I	To halt one or more harts, the debugger selects them, sets haltreq, and then waits for allhalted to indicate the harts are halted. Then it can clear haltreq to 0, or leave it high to catch a hart that resets while halted.
RVD.B.12	B.4 (p.72)	H	Running

ID	REFERENCE	TYPE	DEFINITION
RVD.B.13	B.4 (p.72)	I	First, the debugger should restore any registers that it has overwritten. Then it can let the selected harts run by setting resumereq. Once allresumeack is set, the debugger knows the hart has resumed, and it can clear resumereq. Harts might halt very quickly after resuming (e.g. by hitting a software breakpoint) so the debugger cannot use allhalted/anyhalted to check whether the hart resumed.
RVD.B.14	B.5 (p.72)	H	Single Step
RVD.B.15	B.5 (p.72)	I	Using the hardware single step feature is almost the same as regular running. The debugger just sets step in `dcsr` before letting the hart run. The hart behaves exactly as in the running case, except that interrupts may be disabled (depending on stepie) and it only fetches and executes a single instruction before re-entering Debug Mode.
RVD.B.16	B.6 (p.72)	H	Accessing Registers
RVD.B.17	B.6.1 (p.72)	H	Using Abstract Command
RVD.B.18	B.6.1 (p.72)	I	Read s0 using abstract command:

Op	Address	Value	Comment
Write	`command`	aarsize = 2, transfer, regno = 0x1008	Read s0
Read	`data0`	-	Returns value that was in s0

ID	REFERENCE	TYPE	DEFINITION
RVD.B.19	B.6.1 (p.73)	I	Write mstatus using abstract command:

Op	Address	Value	Comment
Write	`data0`	New value	
Write	`command`	aarsize = 2, transfer, write, regno = 0x300	Write `mstatus`

ID	REFERENCE	TYPE	DEFINITION
RVD.B.20	B.6.2 (p.73)	H	Using Program Buffer
RVD.B.21	B.6.2 (p.73)	I	Abstract commands are used to exchange data with GPRs. Using this mechanism, other registers can be accessed by moving their value into/out of GPRs.
RVD.B.22	B.6.2 (p.73)	I	Write mstatus using program buffer:

Op	Address	Value	Comment
Write	`progbuf0`	csrw s0, MSTATUS	
Write	`progbuf1`	ebreak	
Write	`data0`	new value	
Write	`command`	aarsize = 2, postexec, transfer, write, regno = 0x1008	Write s0, then execute program buffer

ID	REFERENCE	TYPE	DEFINITION
RVD.B.23	B.6.2 (p.73)	I	Read f1 using program buffer:

Op	Address	Value	Comment
Write	`progbuf0`	fmv.x.s s0, f1	
Write	`progbuf1`	ebreak	
Write	`command`	postexec	Execute program buffer
Write	`command`	transfer, regno = 0x1008	read s0
Read	`data0`	-	Returns the value that was in f1

ID	REFERENCE	TYPE	DEFINITION
RVD.B.24	B.7 (p.73)	H	Reading Memory

ID	REFERENCE	TYPE	DEFINITION
RVD.B.25	B.7.1 (p.73)	H	Using System Bus Access
RVD.B.26	B.7.1 (p.73)	I	With system bus access, addresses are physical system bus addresses.
RVD.B.27	B.7.1 (p.73)	I	Read a word from memory using system bus access:

Op	Address	Value	Comment
Write	`sbcs`	Sbaccess = 2, sbreadonaddr	Setup
Write	`sbaddress0`	address	
Read	`sbdata0`	-	Value read from memory

ID	REFERENCE	TYPE	DEFINITION
RVD.B.28	B.7.1 (p.73, p.74)	I	Read block of memory using system bus access:

Op	Address	Value	Comment
Write	`sbcs`	sbaccess = 2, sbreadonaddr, sbreadondata, sbautoincrement	Turn on autoread and autoincrement
Write	`sbaddress0`	address	Writing address triggers read and increment
Read	`sbdata0`	-	Value read from memory
Read	`sbdata0`	-	Next value read from memory
...	`...`	...	...
Write	`sbcs`	0	Disable autoread
Read	`sbdata0`	-	Get last value read from memory.

ID	REFERENCE	TYPE	DEFINITION
RVD.B.29	B.7.2 (p.74)	H	Using Program Buffer
RVD.B.30	B.7.2 (p.74)	I	Through the Program Buffer, the hart performs the memory accesses. Addresses are physical or virtual (depending on mprven and other system configuration).
RVD.B.31	B.7.2 (p.74)	I	Read block of memory using system bus access:

Op	Address	Value	Comment
Write	`progbuf0`	lw s0, 0(s0)	
Write	`progbuf1`	ebreak	
Write	`data0`	address	
Write	`command`	write, postexec, regno = 0x1008	Write s0, then execute program buffer
Write	`command`	regno = 0x1008	Read s0
Read	`data0`	-	Value read from memory

ID	REFERENCE	TYPE	DEFINITION
RVD.B.32	B.7.2 (p.74, p.75)	I	Read block of memory using program buffer:

Op	Address	Value	Comment
Write	`progbuf0`	lw s1, 0(s0)	
Write	`progbuf1`	addi s0, s0, 4	
Write	`progbuf2`	ebreak	
Write	`data0`	address	
Write	`command`	write, postexec, regno = 0x1008	Write s0, then execute program buffer
Write	`command`	postexec, regno = 0x1009	Read s1, then execute program buffer
Write	`abstractauto`	autoexecdata [0]	Set autoexecdata [0]
Read	`data0`	-	Get value read from memory, then execute program buffer
Read	`data0`	-	Get next value read from memory, then execute program buffer
...	`...`	...	...
Write	`abstractauto`	0	Clear autoexecdata [0]
Read	`data0`	-	Get last value read from memory.

ID	REFERENCE	TYPE	DEFINITION
RVD.B.33	B.7.3 (p.75)	H	Using Abstract Memory Access
RVD.B.34	B.7.3 (p.75)	I	Abstract memory accesses act as if they are performed by the hart, although the actual implementation may differ.
RVD.B.35	B.7.3 (p.75)	I	Read block of memory using system bus access:

Op	Address	Value	Comment
Write	`data1`	address	
Write	`command`	cmdtype=2, aamsize =2	
Read	`data0`	-	Value read from memory

ID	REFERENCE	TYPE	DEFINITION
RVD.B.36	B.7.3 (p.75)	I	Read block of memory using abstract memory access:

Op	Address	Value	Comment
Write	`abstractauto`	1	Re-execute the command when data0 is accessed
Write	`data1`	address	
Write	`command`	cmdtype=2, aamsize=2, aampostincrement =1	
Read	`data0`	-	Read value, and trigger reading of next address
...	`...`	...	...
Write	`abstractauto`	0	Disable auto-exec
Read	`data0`	-	Get last value read from memory.

ID	REFERENCE	TYPE	DEFINITION
RVD.B.37	B.8 (p.76)	H	Writing Memory
RVD.B.38	B.8.1 (p.76)	H	Using System Bus Access
RVD.B.39	B.8.1 (p.76)	I	With system bus access, addresses are physical system bus addresses.

ID	REFERENCE	TYPE	DEFINITION
RVD.B.40	B.8.1 (p.76)	I	Write a word to memory using system bus access:

Op	Address	Value	Comment
Write	`sbaddress0`	address	
Write	`sbdata0`	value	

ID	REFERENCE	TYPE	DEFINITION
RVD.B.41	B.8.1 (p.76)	I	Write a block of memory using system bus access:

Op	Address	Value	Comment
Write	`sbcs`	sbaccess = 2, sbautoincrement	Turn on autoincrement
Write	`sbaddress0`	address	
Write	`sbdata0`	value0	
Write	`sbdata0`	value1	
...	`...`	...	...
Write	`sbdata0`	valueN	

ID	REFERENCE	TYPE	DEFINITION
RVD.B.42	B.8.2 (p.76)	H	Using Program Buffer
RVD.B.43	B.8.2 (p.76)	I	Through the Program Buffer, the hart performs the memory accesses. Addresses are physical or virtual (depending on `mprven` and other system configuration).
RVD.B.44	B.8.2 (p.76)	I	Write a word to memory using program buffer:

Op	Address	Value	Comment
Write	`progbuf0`	sw s1, 0(s0)	
Write	`progbuf1`	ebreak	
Write	`data0`	address	
Write	`command`	write, regno = 0x1008	Write s0
Write	`data0`	value	
Write	`command`	write, postexec, regno = 0x1009	Write s1, then execute program buffer

ID	REFERENCE	TYPE	DEFINITION
RVD.B.45	B.8.2 (p.76, p.77)	I	Write block of memory using program buffer:

Op	Address	Value	Comment
Write	`progbuf0`	sw s1, 0(s0)	
Write	`progbuf1`	addi s0, s0, 4	
Write	`progbuf2`	ebreak	
Write	`data0`	address	
Write	`command`	write, regno = 0x1008	Write s0
Write	`data0`	value0	
Write	`command`	write, postexec, regno = 0x1009	Write s1, then execute program buffer
Write	`abstractauto`	autoexecdata [0]	Set autoexecdata [0]
Write	`data0`	value1	
...	`...`	...	...
Write	`data0`	valueN	
Write	`abstractauto`	0	Clear autoexecdata [0]

ID	REFERENCE	TYPE	DEFINITION
RVD.B.46	B.8.3 (p.77)	H	Using Abstract Memory Access
RVD.B.47	B.8.3 (p.77)	I	Abstract memory accesses act as if they are performed by the hart, although the actual implementation may differ.

ID **REFERENCE** **TYPE** **DEFINITION**

RVD.B.48 B.8.3 (p.77) I Write a word to memory using abstract memory access:

Op	Address	Value	Comment
Write	`data1`	address	
Write	`data0`	value	
Write	`command`	cmdtype=2, aamsize =2, write=1	

RVD.B.49 B.8.3 (p.77) I Write a block of memory using abstract memory access:

Op	Address	Value	Comment
Write	`data1`	address	
Write	`data0`	value	
Write	`command`	cmdtype=2, aamsize=2, write=1, aampostincrement=1	
Write	`abstractauto`	1	Re-execute the command when data0 is accessed
Write	`data0`	value1	
Write	`data0`	value2	
...	`...`		
Write	`data0`	valueN	
Write	`abstractauto`	0	Disable auto-exec

RVD.B.50 B.9 (p.78) H Triggers

RVD.B.51 B.9 (p.78) I A debugger can use hardware triggers to halt a hart when a certain event occurs. Below are some examples, but as there is no requirement on the number of features of the triggers implemented by a hart, these examples may not be applicable to all implementations. When a debugger wants to set a trigger, it writes the desired configuration, and then reads back to see if that configuration is supported.

RVD.B.52 B.9 (p.78) I Enter Debug Mode just before the instruction at 0x80001234 is executed, to be used as an instruction breakpoint in ROM:

`tdata1`	`0x105c`	action=1, match=0, m=1, s=1, u=1, execute=1
`tdata2`	`0x80001234`	address

RVD.B.53 B.9 (p.78) I Enter Debug Mode right after the value at 0x80007f80 is read:

`tdata1`	`0x4159`	timing=1, action=1, match=0, m=1, s=1, u=1, load=1
`tdata2`	`0x80007f80`	address

RVD.B.54 B.9 (p.78) I Enter Debug Mode right before a write to an address between 0x80007c80 and 0x80007cef (inclusive):

`tdata1 0`	`0x195a`	action=1, chain=1, match=2, m=1, s=1, u=1, store=1
`tdata2 0`	`0x80007c80`	start address (inclusive)
`tdata1 1`	`0x11da`	action=1, match=3, m=1, s=1, u=1, store=1
`tdata2 1`	`0x80007cf0`	end address (exclusive)

RVD.B.55 B.9 (p.78) I Enter Debug Mode right before a write to an address between 0x81230000 and 0x8123ffff (inclusive):

`tdata1`	`0x10da`	action=1, match=1, m=1, s=1, u=1, store=1
`tdata2`	`0x81237fff`	16 bits to match exactly, then 0, then all ones.

ID	REFERENCE	TYPE	DEFINITION
RVD.B.56	B.9 (p.78)	I	Enter Debug Mode right after a read from an address between 0x86753090 and 0x8675309f or between 0x96753090 and 0x9675309f (inclusive):

`tdata1 0`	`0x41a59`	timing=1, action=1, chain=1, match=4, m=1, s=1, u=1, load=1
`tdata2 0`	`0xfff03090`	Mask for low half, then match for low half
`tdata1 1`	`0x412d9`	timing=1, action=1, match=5, m=1, s=1, u=1, load=1
`tdata2 1`	`0xefff8675`	Mask for high half, then match for high half

ID	REFERENCE	TYPE	DEFINITION
RVD.B.57	B.10 (p.79)	H	Handling Exceptions
RVD.B.58	B.10 (p.79)	I	Generally the debugger can avoid exceptions by being careful with the programs it writes. Sometimes they are unavoidable though, e.g. if the user asks to access memory or a CSR that is not implemented. A typical debugger will not know enough about the platform to know what's going to happen, and must attempt the access to determine the outcome.
RVD.B.59	B.10 (p.79)	I	When an exception occurs while executing the Program Buffer, `cmderr` becomes set. The debugger can check this eld to see whether a program encountered an exception. If there was an exception, it's left to the debugger to know what must have caused it.
RVD.B.60	B.11 (p.79)	H	Quick Access
RVD.B.61	B.11 (p.79)	I	There are a variety of instructions to transfer data between GPRs and the data registers. They are either loads/stores or CSR reads/writes. The specific addresses also vary. This is all specified in `hartinfo`. The examples here use the pseudo-op **transfer dest**, **src** to represent all these options.
RVD.B.62	B.11 (p.79)	I	Halt the hart for a minimum amount of time to perform a single memory write:

Op	Address	Value	Comment
Write	`progbuf0`	transfer arg2, s0	Save s0
Write	`progbuf1`	transfer s0, arg0	Read rst argument (address)
Write	`progbuf2`	transfer arg0, s1	Save s1
Write	`progbuf3`	transfer s1, arg1	Read second argument (data)
Write	`progbuf4`	sw s1, 0(s0)	
Write	`progbuf5`	transfer s1, arg0	Restore s1
Write	`progbuf6`	transfer s0, arg2	Restore s0
Write	`progbuf7`	ebreak	
Write	`data0`	address	
Write	`data1`	data	
Write	`command`	0x10000000	Perform quick access

ID	REFERENCE	TYPE	DEFINITION
RVD.B.63	B.11 (p.79)	I	This shows an example of setting the m bit in mcontrol to enable a hardware breakpoint in M-mode. Similar quick access instructions could have been used previously to configure the trigger that is being enabled here:

Op	Address	Value	Comment
Write	`progbuf0`	transfer arg0, s0	Save s0
Write	`progbuf1`	li s0, (1 << 6)	Form the mask for m bit
Write	`progbuf2`	csrrs x0, tdata1, s0	Apply the mask to mcontrol
Write	`progbuf3`	transfer s0, arg2	Restore s0
Write	`progbuf4`	ebreak	
Write	`command`	0x10000000	Perform quick access