

Chapter 1

Debug Module (DM)

The Debug Module implements a translation interface between abstract debug operations and their specific implementation. It might support the following operations:

1. Give the debugger necessary information about the implementation. (Required)
2. Allow any individual hart to be halted and resumed. (Required)
3. Provide status on which harts are halted. (Required)
4. Provide abstract read and write access to a halted hart's GPRs. (Required)
5. Provide access to a reset signal that allows debugging from the very first instruction after reset. (Required)
6. Provide a mechanism to allow debugging harts immediately out of reset (regardless of the reset cause). (Optional)
7. Provide abstract access to non-GPR hart registers. (Optional)
8. Provide a Program Buffer to force the hart to execute arbitrary instructions. (Optional)
9. Allow multiple harts to be halted, resumed, and/or reset at the same time. (Optional)
10. Allow memory access from a hart's point of view. (Optional)
11. Allow direct System Bus Access. (Optional)

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 ??.
 - (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 0.13.1”, instead of the “RISC-V Debug Specification 0.13.1”.

A single DM can debug up to 2^{20} harts.

1.1 Debug Module Interface (DMI)

Debug Modules are slaves to a bus called the Debug Module Interface (DMI). The master of the bus is the Debug Transport Module(s). 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.

The DMI uses between 7 and 32 address bits. It supports read and write operations. The bottom of the address space is used for the first (and usually only) DM. Extra space can be used for custom debug devices, other cores, additional DMs, etc. If there are additional DMs on this DMI, the base address of the next DM in the DMI address space is given in `nextdm`.

The Debug Module is controlled via register accesses to its DMI address space.

1.2 Reset Control

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. Exactly what is affected by this reset is implementation dependent, as long as it is possible to debug programs from the first instruction executed. The Debug Module's own state and registers should only be reset at power-up and while `dmactive` in `dmcontrol` is 0. The halt state of harts should be maintained across system reset provided that `dmactive` is 1, although trigger CSRs may be cleared.

Due to clock and power domain crossing issues, it may not be possible to perform arbitrary DMI accesses across system reset. While `ndmreset` or any external reset is asserted, the only supported DM operation is accessing `dmcontrol`. The behavior of other accesses is undefined.

There is no requirement on the duration of the assertion of `ndmreset`. The implementation must ensure that a write of `ndmreset` to 1 followed by a write of `ndmreset` to 0 triggers system reset. The system may take an arbitrarily long time to come out of reset, as reported by `allunavail`, `anyunavail`.

Individual harts (or several at once) can be reset by selecting them, setting and then clearing `hartreset`. In this case an implementation may reset more harts than just the ones that are selected. The debugger can discover which other harts are reset (if any) by selecting them and checking `anyhavereset` and `allhavereset`.

When harts have been reset, they must set a sticky `havereset` state bit. The conceptual `havereset` state bits can be read for selected harts in `anyhavereset` and `allhavereset` in `dmstatus`. These bits must be set regardless of the cause of the reset. The `havereset` bits for the selected harts can be cleared by writing 1 to `ackhavereset` in `dmcontrol`. The `havereset` bits may or may not be cleared when `dmactive` is low.

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.

1.3 Selecting Harts

Up to 2^{20} harts can be connected to a single DM. The debugger selects a hart, and then subsequent halt, resume, reset, and debugging commands are specific to that hart.

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. Then it selects each hart starting from 0 until either `anynonexistent` in `dmstatus` is 1, or the highest index (depending on `HARTSELLEN`) is reached.

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.

1.3.1 Selecting a Single Hart

All debug modules must support selecting a single hart. The debugger can select a hart by writing its index to `hartsel`. Hart indexes start at 0 and are contiguous until the final index.

1.3.2 Selecting Multiple Harts

Debug Modules may implement a Hart Array Mask register to allow selecting multiple harts at once. 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. 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.

The debugger can set bits in the hart array mask register using `hawindowssel` and `hawindow`, then apply actions to all selected harts by setting `hasel`. If this feature is supported, multiple harts can be halted, resumed, and reset simultaneously. The state of the hart array mask register is not affected by setting or clearing `hasel`.

Only the actions initiated by `dmcontrol` can apply to multiple harts at once, Abstract Commands apply only to the hart selected by `hartsel`.

1.4 Hart States

Every hart that can be selected is in exactly one of four states. Which state the selected harts are in is reflected by `allnonexistent`, `anynonexistent`, `allunavail`, `anyunavail`, `allrunning`, `anyrunning`, `allhalted`, and `anyhalted`.

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.

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. Harts may be unavailable for a variety of reasons including being

reset, temporarily powered down, and not being plugged into the system. 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.

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.

Harts are halted when they are in Debug Mode, only performing tasks on behalf of the debugger.

Which states a hart that is reset goes through is implementation dependent. Harts may be unavailable while reset is asserted, and some time after reset is deasserted. They might transition to running for some time after reset is deasserted. Finally they end up either running or halted, depending on `haltreq` and `resethaltreq`.

1.5 Run Control

For every hart, the Debug Module tracks 4 conceptual bits of state: halt request, resume ack, halt-on-reset request, and hart reset. (The hart reset and halt-on-reset request bits are optional.) These 4 bits reset to 0, except for resume ack, which may reset to either 0 or 1. The DM receives halted, running, and havereset signals from each hart. 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`. The state of the other bits cannot be observed directly.

When a debugger writes 1 to `haltreq`, each selected hart's halt request bit is set. 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. Halted harts ignore their halt request bit.

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. 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. These status signals of all selected harts are reflected in `allresumeack`, `anyresumeack`, `allrunning`, and `anyrunning`. Resume requests are ignored by running harts.

When halt or resume is requested, a hart must respond in less than one second, unless it is unavailable. (How this is implemented is not further specified. A few clock cycles will be a more typical latency).

The DM can implement optional halt-on-reset bits for each hart, which it indicates by setting `hasresethaltreq` to 1. This means the DM implements the `setresethaltreq` and `clrresethaltreq` bits. Writing 1 to `setresethaltreq` sets the halt-on-reset request bit for each selected hart. 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. This is true regardless of the reset's cause. 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.

1.6 Abstract Commands

The DM supports a set of abstract commands, most of which are optional. Depending on the implementation, the debugger may be able to perform some abstract commands even when the selected hart is not halted. 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. Commands may be supported with some options set, but not with other options set. If a command has unsupported options set, the DM must set `cmderr` to 2 (not supported).

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 “not supported.”

Debuggers execute abstract commands by writing them to `command`. They can determine whether an abstract command is complete by reading `busy` in `abstractcs`. After completion, `cmderr` indicates whether the command was successful or not. Commands may fail because a hart is not halted, not running, unavailable, or because they encounter an error during execution.

If the command takes arguments, the debugger must write them to the `data` registers before writing to `command`. If a command returns results, the Debug Module must ensure they are placed in the `data` registers before `busy` is cleared. Which `data` registers are used for the arguments is described in Table ???. In all cases the least-significant word is placed in the lowest-numbered `data` register. The argument width depends on the command being executed, and is `DXLEN` where not explicitly specified.

Table 1.1: Use of Data Registers

Argument Width	arg0/return value	arg1	arg2
32	<code>data0</code>	<code>data1</code>	<code>data2</code>
64	<code>data0</code> , <code>data1</code>	<code>data2</code> , <code>data3</code>	<code>data4</code> , <code>data5</code>
128	<code>data0</code> – <code>data3</code>	<code>data4</code> – <code>data7</code>	<code>data8</code> – <code>data11</code>

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.

Before starting an abstract command, a debugger must ensure that `haltreq`, `resumereq`, and `ackhavereset` are all 0.

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`.

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`).

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).

1.6.1 Abstract Command Listing

This section describes each of the different abstract commands and how their fields should be interpreted when they are written to **command**.

Each abstract command is a 32-bit value. The top 8 bits contain **cmdtype** which determines the kind of command. Table ?? lists all commands.

Table 1.2: Meaning of **cmdtype**

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

1.6.1.1 Access Register

This command gives the debugger access to CPU registers and allows it to execute the Program Buffer. 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.

If any of these operations fail, **cmderr** is set and none of the remaining steps are executed. An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure. If the failure is that the requested register does not exist in the hart, **cmderr** must be set to 3 (exception).

Debug Modules must implement this command and must support read and write access to all GPRs when the selected hart is halted. Debug Modules may optionally support accessing other registers, or accessing registers when the hart is running. Each individual register (aside from GPRs) may be supported differently across read, write, and halt status.

*The encoding of **aarsize** was chosen to match **sbaccess** in **sbc**s.*

This command modifies **arg0** only when a register is read. The other **data** registers are not changed.

Table 1.3: Abstract Register Numbers

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

31	24	23	22	20	19	18	17	16	15	0
cmdtype	0	aarsize	aarpostincrement		postexec	transfer	write		regno	
8	1	3	1		1	1	1		16	

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 <code>aarsize</code> specifies a size larger than the register’s actual size, then the access must fail. If a register is accessible, then reads of <code>aarsize</code> less than or equal to the register’s actual size must be supported. This field controls the Argument Width as referenced in Table ??.
aarpostincrement	0: No effect. This variant must be supported. 1: After a successful register access, <code>regno</code> 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 <code>progbuFSIZE</code> 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: Don’t do the operation specified by <code>write</code> . 1: Do the operation specified by <code>write</code> . This bit can be used to just execute the Program Buffer without having to worry about placing valid values into <code>aarsize</code> or <code>regno</code> .
write	When <code>transfer</code> is set: 0: Copy data from the specified register into <code>arg0</code> portion of <code>data</code> . 1: Copy data from <code>arg0</code> portion of <code>data</code> into the specified register.
regno	Number of the register to access, as described in Table ?. <code>dpc</code> may be used as an alias for PC if this command is supported on a non-halted hart.

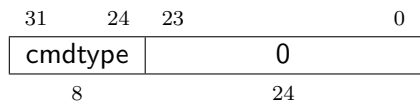
1.6.1.2 Quick Access

Perform the following sequence of operations:

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.

Implementing this command is optional.

This command does not touch the **data** registers.



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