LAUTERBACH
DEVELOPMENT TOOLS

RISC-V Debugger

Release 09.2023

MANUAL

RISC-V Debugger

Version 10-Oct-2023

History

18-Aug-23	Marked **SYStem.CONFIG.HARTINDEX** as deprecated command and replaced by SYStem.CONFIG.HART.INDEX.
17-Oct-22	New subchapter 'Vector Extension'.
06-Oct-22	New command SYStem.Mode.StandBy.
06-Oct-22	New subchapter: 'Hart State: Unavailable'.
29-Aug-22	New command: SYStem.Option.DMACTiveRESet.
20-Jul-22	For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.
02-Mar-22	New command: SYStem.Option.KeepAlive.
29-Mar-21	New command: SYStem.CONFIG.IRWIDTH.
04-Sep-17	Initial version.

Introduction

This manual serves as a guideline for debugging one or multiple RISC-V cores via TRACE32.

Please keep in mind that only the **Processor Architecture Manual** (the document you are reading at the moment) is CPU specific, while all other parts of the online help are generic for all CPUs supported by Lauterbach. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Training Basic Debugging"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_*<x>*.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

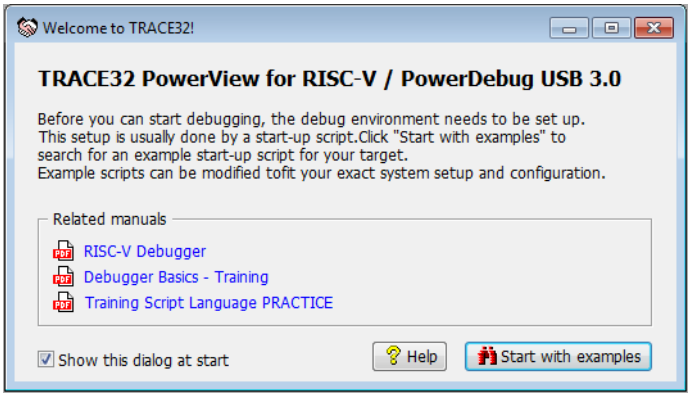
PRACTICE Script Language:

- **"Training Script Language PRACTICE"** (training_practice.pdf)
- **"PRACTICE Script Language Reference Guide"** (practice_ref.pdf)

Video Tutorials:

- **Lauterbach YouTube channel**

To get started with the most important manuals, use the **Welcome to TRACE32!** dialog (**WELCOME.view**):



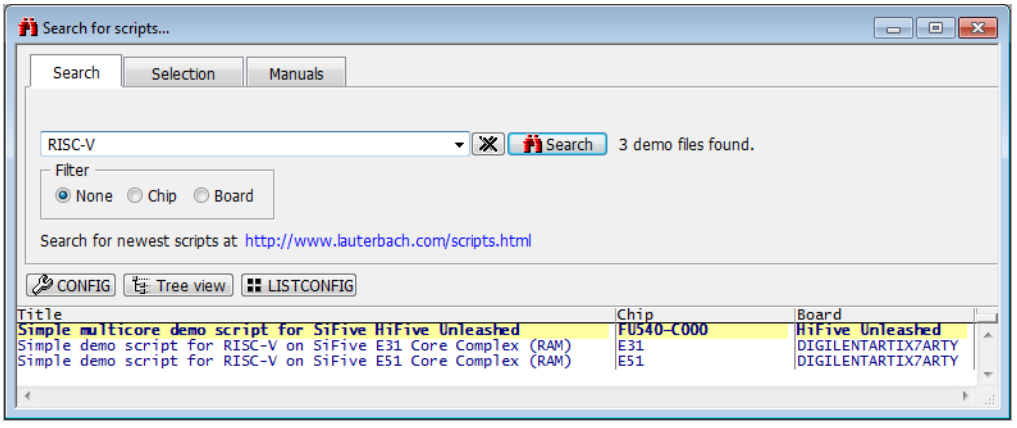
Demo and Start-up Script

Lauterbach provides ready-to-run start-up scripts for known hardware that is based on RISC-V.

To search for PRACTICE scripts, do one of the following in TRACE32 PowerView:

- Type at the command line: **WELCOME.SCRIPTS**
- or choose **File** menu > **Search for Script**.

  You can now search the demo folder and its subdirectories for PRACTICE start-up scripts (*.cmm) and other demo software.



You can also manually navigate in the ~~/demo/riscv/ subfolder of the system directory of TRACE32.

List of Abbreviations and Definitions

CSR	Control and Status Register
DM	Debug Module, as defined by the **RISC-V debug specification**
DTM	Debug Transport Module, as defined by the **RISC-V debug specification**
HART	Hardware thread. A single RISC-V core contains one or multiple hardware threads.
XLEN	The current width of a RISC-V general purpose register in bits.

WARNING:	To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF. Recommendation for the software start: 1. Disconnect the Debug Cable from the target while the target power is off. 2. Connect the host system, the TRACE32 hardware and the Debug Cable. 3. Power ON the TRACE32 hardware. 4. Start the TRACE32 software to load the debugger firmware. 5. Connect the Debug Cable to the target. 6. Switch the target power ON. 7. Configure your debugger e.g. via a start-up script. Power down: 1. Switch off the target power. 2. Disconnect the Debug Cable from the target. 3. Close the TRACE32 software. 4. Power OFF the TRACE32 hardware.

Starting up the debugger is done as follows:

1. **Select the device prompt for the ICD Debugger.**

```
B::
```

The device prompt `B::` is normally already selected in the **TRACE32 command line**. If this is not the case, enter `B::` to set the correct device prompt.

2. **Reset the debugger settings.**

```
RESet
```

The **RESet** command ensures that no debugger setting remains from a former debug session. All settings get set to their default value. **RESet** is not required if you start the debug session directly after starting the TRACE32 development tool. **RESet** does *not* reset the target.

3. **Select the chip or core you intend to debug.**

```
SYStem.CPU <cpu_type>
```

Based on the selected chip the debugger sets the **SYStem.CONFIG** and **SYStem.Option** commands the way which should be most appropriate for debugging this chip. Ideally no further setup is required. Please note that the default configuration is not always the best configuration for your target.

4. **Configure the JTAG interface.**

You can select the JTAG clock frequency, which the Debugger uses to communicate with the target. This can be either done in the JtagClock field in the SYStem window, or by using the command line with the command **SYStem.JtagClock**. The maximum clock frequency might depend on the configuration of your FPGA design. The default clock frequency is 10 MHz.

In case of a JTAG daisy chain use command **SYStem.DETECT SHOWChain** to scan the chain. The result is shown in a window. Double-click on the desired core to tell the debugger which core you'd like to debug.

If the RISC-V Debug Module (DM) is accessible via a JTAG-DTM and the JTAG TAP of that JTAG-DTM is daisy-chained with other TAPs then you can manually configure the JTAG daisy chain with **SYStem.CONFIG.IRPOST**, **SYStem.CONFIG.IRPRE**, **SYStem.CONFIG.DRPOST** and **SYStem.CONFIG.DRPRE**.

If the system has an Arm CoreSight Debug Access Port (Arm DAP) and the JTAG TAP of the DAP is

daisy-chained with other TAPs then you can manually configure hte JTAG daisy chain with **SYStem.CONFIG.DAPIRPOST**, **SYStem.CONFIG.DAPIRPRE**, **SYStem.CONFIG.DAPDRPOST** and **SYStem.CONFIG.DAPDRPRE**.

5. **Configure memory access ports (if available).**

   If the target SoC has an Arm CoreSight debug infrastructure, then the memory access ports need to be configured in order to make their buses accessible via the respective access classes.
   For Arm SoC-400, see **SYStem.CONFIG.APBAP.Port**, **SYStem.CONFIG.AHBAP.Port** and **SYStem.CONFIG.AXIAP.Port** for details.
   For Arm SoC-600, see **SYStem.CONFIG.APBAP.Base**, **SYStem.CONFIG.AHBAP.Base** and **SYStem.CONFIG.AXIAP.Base** for details.

6. **Tell the debugger how to access the RISC-V Debug Module.**

   See chapter **"Quick Start for Debug Module Configuration"** (debugger_riscv.pdf) for details.

7. **Select the reset method.**

```
SYStem.Option.ResetMode <method>
```

   If the debugger is supposed to perform a system reset or core reset while connecting to the target, then the reset method that is most suitable for the target needs to be configured with **SYStem.Option.ResetMode**.

8. **Connect to target.**

```
SYStem.Up
```

   This command establishes the JTAG communication to the target. It resets the processor and enters debug mode (halts the processor; ideally at the reset vector). After this command is executed, it is possible to access memory and registers.

   Some devices can not communicate via JTAG while in reset or you might want to connect to a running program without causing a target reset. In this case use

```
SYStem.Mode Attach
```

   instead. A **Break.direct** will halt the processor.

9. **Load the program you want to debug.**

```
Data.LOAD <file>
```

   This loads the executable to the target and the debug/symbol information to the debugger's host. If the program is already on the target and you just need the debug/symbol information then load with **/NoCODE** option.

   A detailed description of the **Data.LOAD** command and all available options is given in the **"General Commands Reference"**.

LAUTERBACH recommends to prepare a PRACTICE script (*.cmm, ASCII file format) to be able to do all the necessary actions with only one command, such as the command **DO** *<file>*.

The following example shows a system configuration for a **RISC-V system with JTAG-DTM**:

```
RESet                                ; Reset the debugger configuration

SYStem.CPU FU540-C000                ; Select the SoC/CPU/core

SYStem.JtagClock 5.MHz               ; Set JTAG clock frequency

SYStem.CONFIG IRPRE  4.              ; Configure JTAG daisy chain
SYStem.CONFIG IRPOST 0.
SYStem.CONFIG DRPRE  1.
SYStem.CONFIG DRPOST 0.

SYStem.Option.ResetMode NDMRST       ; Select the reset method
```

A typical example for a start sequence (in addition to the above configuration) might look like the following:

```
SYStem.Up                            ; Reset the target, stop the core at
                                     ; the reset vector, enter debug mode

Data.LOAD.Elf riscv_le.elf           ; Load the application

Register.Set PC main                 ; Set the PC to function main

Register.Set X2 0x63FFFFFC           ; Set the stack pointer to
                                     ; address 0x63FFFFFC

Break.Set P:0x1000 /Program          ; Set breakpoint to address P:0x1000


List.Mix                             ; Open source code window

Register.view /SpotLight             ; Open register window

Frame.view /Locals /Caller           ; Open the stack frame with
                                     ; local variables

Var.Watch %SpotLight var1 var2       ; Open watch window for variables

PER.view                             ; Open window for tree view of
                                     ; system peripherals
```

For more details about the configuration of a RISC-V system with **JTAG-DTM**, please see chapter **Debug Module access via JTAG-DTM**.

LAUTERBACH recommends to prepare a PRACTICE script (*.cmm, ASCII file format) to be able to do all the necessary actions with only one command, such as the command **DO** *<file>*.

The following example shows a configuration for a **RISC-V core in an Arm CoreSight SoC-400 system**:

```
RESet                                       ; Reset debugger configuration

SYStem.CPU RV32                             ; Select the SoC/CPU/core

SYStem.JtagClock 5.MHz                      ; Set JTAG clock frequency

SYStem.CONFIG.DAPIRPRE  4.                  ; Configure JTAG daisy chain
SYStem.CONFIG.DAPIRPOST 0.
SYStem.CONFIG.DAPDRPRE  1.
SYStem.CONFIG.DAPDRPOST 0.

SYStem.Option.ResetMode NDMRST              ; Select the reset method

SYStem.CONFIG.APBAP1.Port 4.                ; Configure DAP memory
SYStem.CONFIG.AXIAP1.Port 5.                ; access ports (SoC-400)

SYStem.CONFIG.COREDEBUG.Base APB:0x2000     ; Configure APB base address
                                            ; of RISC-V debug module



SYStem.Up                                   ; Reset the target, stop the
                                            ; core at the reset vector and
                                            ; enter debug mode
```

For additional configuration examples of a RISC-V system integrated into an Arm **CoreSight SoC-400** system, please see chapter “Debug Module Access via Debug Bus”, **subchapter for SoC-400**.

LAUTERBACH recommends to prepare a PRACTICE script (*.cmm, ASCII file format) to be able to do all the necessary actions with only one command, such as the command **DO** *<file>*.

The following example shows a configuration for a **RISC-V core in an Arm CoreSight SoC-600 system**:

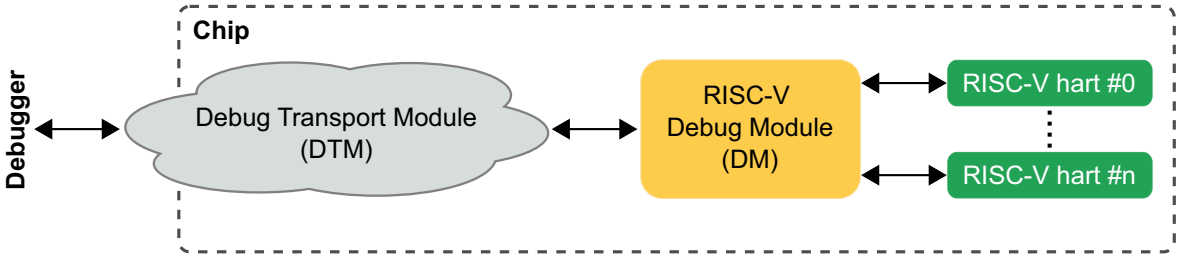
```
RESet                                       ; Reset debugger configuration

SYStem.CPU RV32                             ; Select the SoC/CPU/core

SYStem.JtagClock 5.MHz                      ; Set JTAG clock frequency

SYStem.CONFIG.DAPIRPRE  4.                  ; Configure JTAG daisy chain
SYStem.CONFIG.DAPIRPOST 0.
SYStem.CONFIG.DAPDRPRE  1.
SYStem.CONFIG.DAPDRPOST 0.

SYStem.Option.ResetMode NDMRST              ; Select the reset method

SYStem.CONFIG.APBAP1.Base DP:0x30000        ; Configure memory access port
SYStem.CONFIG.AXIAP1.Base DP:0x70000        ; base addresses (SoC-600)

SYStem.CONFIG.COREDEBUG.Base APB:0x2000     ; Configure APB base address
                                            ; of RISC-V debug module



SYStem.Up                                   ; Reset the target, stop the
                                            ; core at the reset vector and
                                            ; enter debug mode
```

For additional configuration examples of a RISC-V system integrated into an Arm **CoreSight SoC-600** system, please see chapter “Debug Module Access via Debug Bus”, **subchapter for SoC-600**.

Quick Start for Debug Module Configuration

The *RISC-V Debug Module (DM)* is the central IP block that contains the debug registers, which give the debugger access to most RISC-V debug functionalities. Usually all RISC-V hardware threads (harts) in a system are connected to the same DM.

A DM can be integrated into a system in various ways. Any abstract IP block which provides access to the debug registers of the DM is called *Debug Transport Module (DTM)*:



The RISC-V debug specification does *not* specify which interface and implementation the DTM needs to have. In theory, the implementation of the DTM can be completely chip-specific. The RISC-V debug specification does however define one standardized DTM, the so-called **RISC-V JTAG-DTM**.

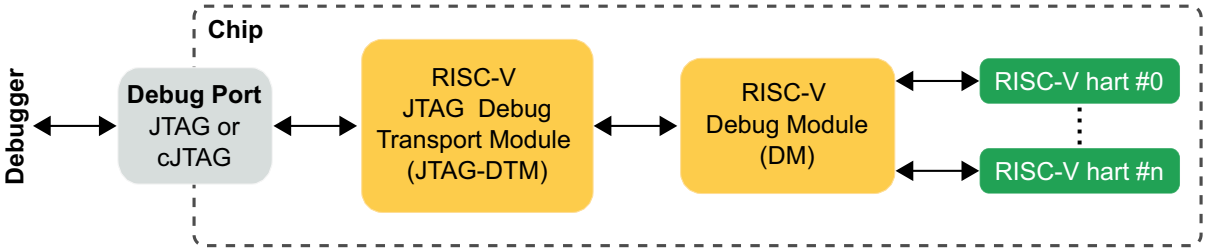
The debugger needs to know how the DM's debug registers can be accessed. That is why this chapter provides a quick start for DM configuration. The following examples cover the most common use cases for DM integration into a system:

- **Example A**: **Debug Module Access via JTAG-DTM**
- **Example B**: **Debug Module Access via Debug Bus**

Debug Module Access via JTAG-DTM

The simplest way to access a RISC-V Debug Module (DM) from an external JTAG interface is via a *JTAG Debug Transport Module* (JTAG-DTM). The JTAG-DTM is specified and standardized by the RISC-V debug specification.

A simple example setup could look as follows:



The RISC-V debugger considers a JTAG-DTM the *default* way to access the DM. This means if no user configuration implies any other way to access the DM then the debugger automatically assumes the existence of a JTAG-DTM.

JTAG-DTM with JTAG port

If the JTAG-DTM does have a normal JTAG port (IEEE 1149.1), then **SYStem.CONFIG.DEBUGPORTTYPE** needs to be set to “JTAG” (default setting).

JTAG-DTM with cJTAG port

However, the RISC-V debugger does also support JTAG-DTMs with a *cJTAG* port (IEEE 1149.7). In this case, **SYStem.CONFIG.DEBUGPORTTYPE** needs to be set to “cJTAG”.

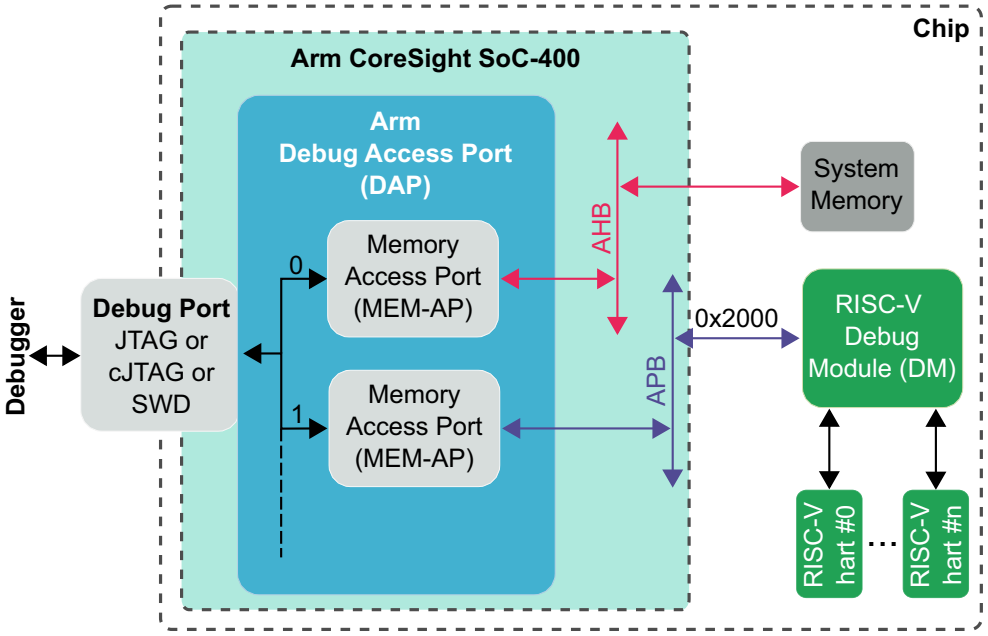
Debug Module Access via Debug Bus

An alternative way to make the RISC-V Debug Module (DM) accessible to a debugger is to map the debug registers of the DM on an existing debug bus.

If the DM debug registers are bus-mapped then the bus type (i.e. the access class) and the base address of the DM must be configured with the command **SYStem.CONFIG COREDEBUG.Base**.

Example: Arm CoreSight SoC-400

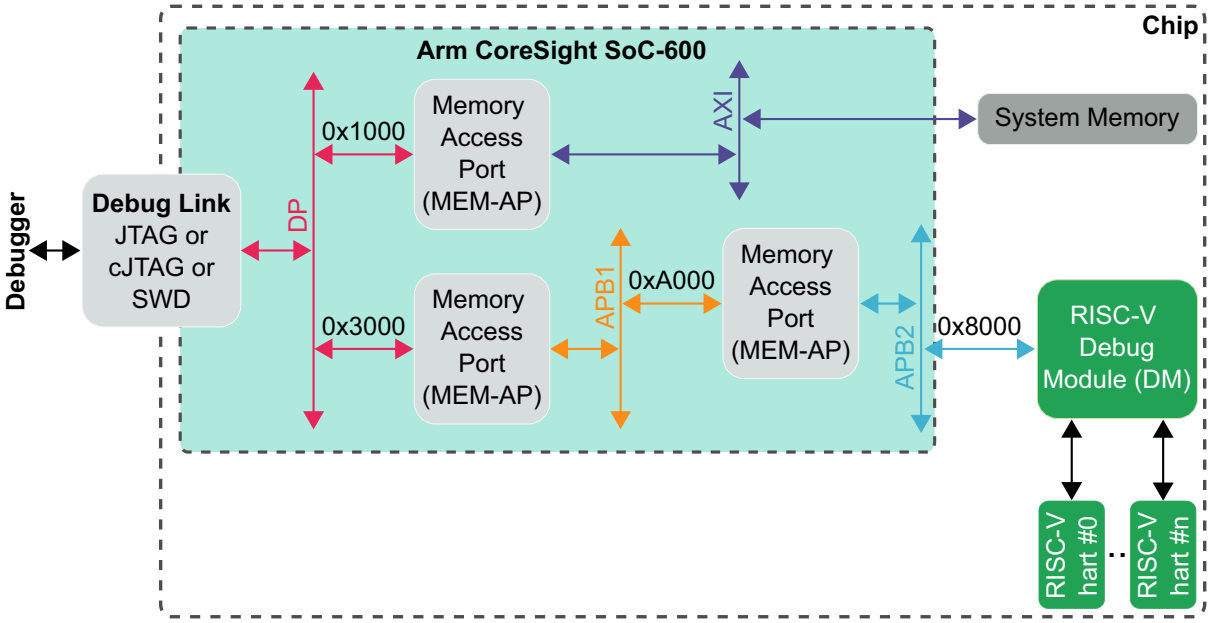
The following example shows a DM that is mapped on a debug bus of an Arm CoreSight SoC-400 system:



TRACE32 configuration:
SYStem.CONFIG AHBAP1.Port 0.
SYStem.CONFIG APBAP1.Port 1.
SYStem.CONFIG COREDEBUG.Base APB:0x2000

The type of the debug port (JTAG, cJTAG or SWD) can be configured via **SYStem.CONFIG.DEBUGPORTTYPE**.

The following example shows a DM that is mapped on a debug bus of an Arm CoreSight SoC-600 system:



TRACE32 configuration:
SYStem.CONFIG AXIAP1.Base DP:0x1000
SYStem.CONFIG APBAP1.Base DP:0x3000
SYStem.CONFIG APBAP2.Base APB1:0xA000
SYStem.CONFIG COREDEBUG.Base APB2:0x8000

The type of the debug port (JTAG, cJTAG or SWD) can be configured via **SYStem.CONFIG.DEBUGPORTTYPE**.

Quick Start for Multicore Debugging

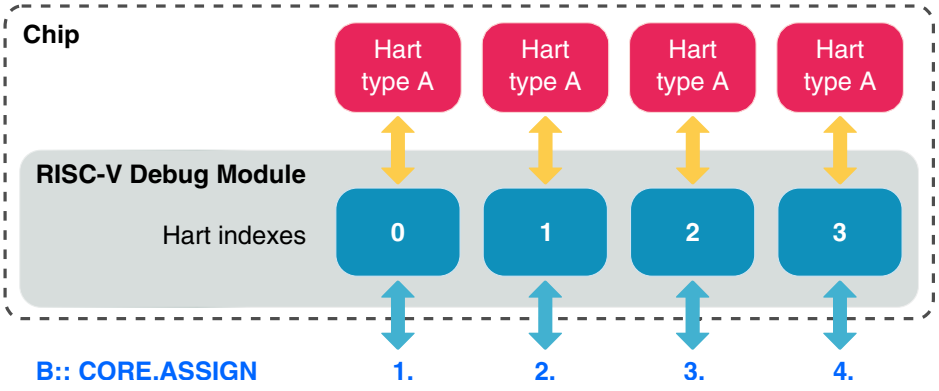
This chapter provides a quick start for multicore processing. The following example scenarios cover the most common use cases for **s**ymmetric **m**ulti**p**rocessing (SMP) and **a**symmetric **m**ulti**p**rocessing (AMP):

- **Example A**: SMP Debugging
- **Example B**: SMP Debugging - Selective
- **Example C**: Homogeneous SMP/AMP Debugging
- **Example D**: Heterogeneous SMP/AMP Debugging

SMP Debugging

This scenario for homogeneous symmetric multiprocessing (SMP) covers the following setup:

4 hardware threads (harts) of the same type are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of the RISC-V Debug Module ranging from 0 to 3. All 4 harts will be debugged simultaneously via SMP.



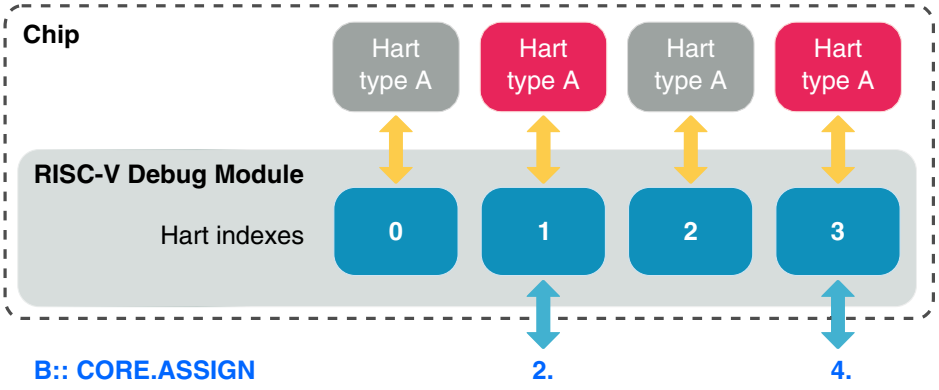
Example A:

```
SYStem.CPU <type_a_cpu>
SYStem.CONFIG CORE 1. 1.                    ; Core group 1 for chip 1
SYStem.CONFIG CoreNumber 4.                 ; 4 harts of type A in total
SYStem.CONFIG HART.INDEX 0. 1. 2. 3.
CORE.ASSIGN 1. 2. 3. 4.                     ; Assign all 4 harts to the
                                            ; SMP session
```

SMP Debugging - Selective

This scenario for homogeneous symmetric multiprocessing (SMP) covers the following setup:

4 hardware threads (harts) of the same type are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of the RISC-V Debug Module ranging from 0 to 3. The harts with hart indexes 1 and 3 will be debugged simultaneously via SMP.



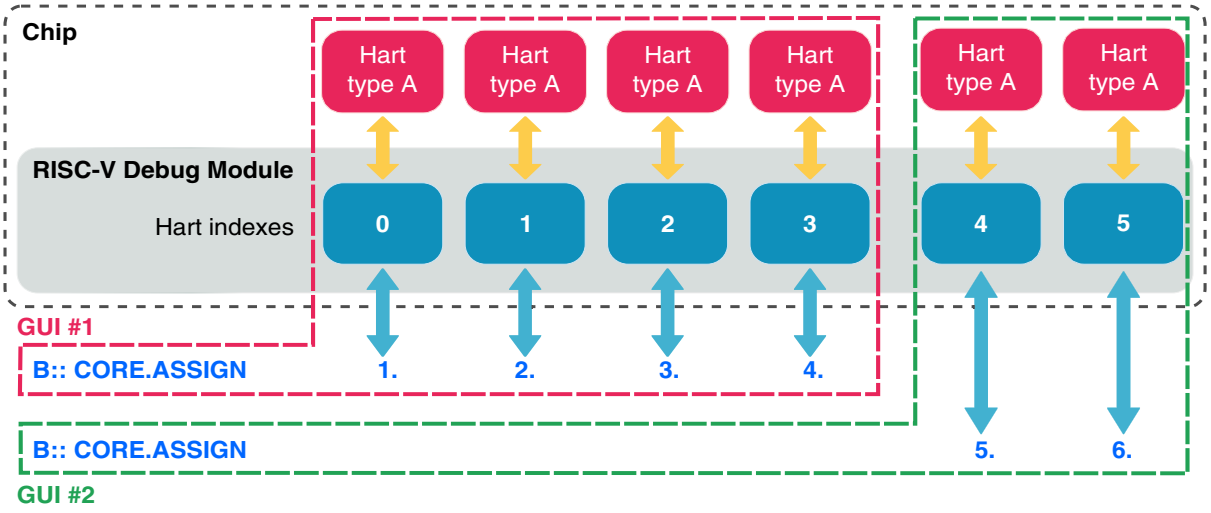
Example B:

```
SYStem.CPU <type_a_cpu>
SYStem.CONFIG CORE 1. 1.                    ; Core group 1 for chip 1
SYStem.CONFIG CoreNumber 4.                 ; 4 harts of type A in total
SYStem.CONFIG HART.INDEX 0. 1. 2. 3.
CORE.ASSIGN 2. 4.                           ; Assign harts with the
                                            ; logical indexes 2 and 4
```

Homogeneous SMP/AMP Debugging

This scenario covers both homogeneous symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).

6 hardware threads (harts) of the same type are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of RISC-V Debug Module ranging from 0 to 5. The first 4 harts will be debugged in an SMP session, and the remaining 2 harts in another SMP session.



Example C:

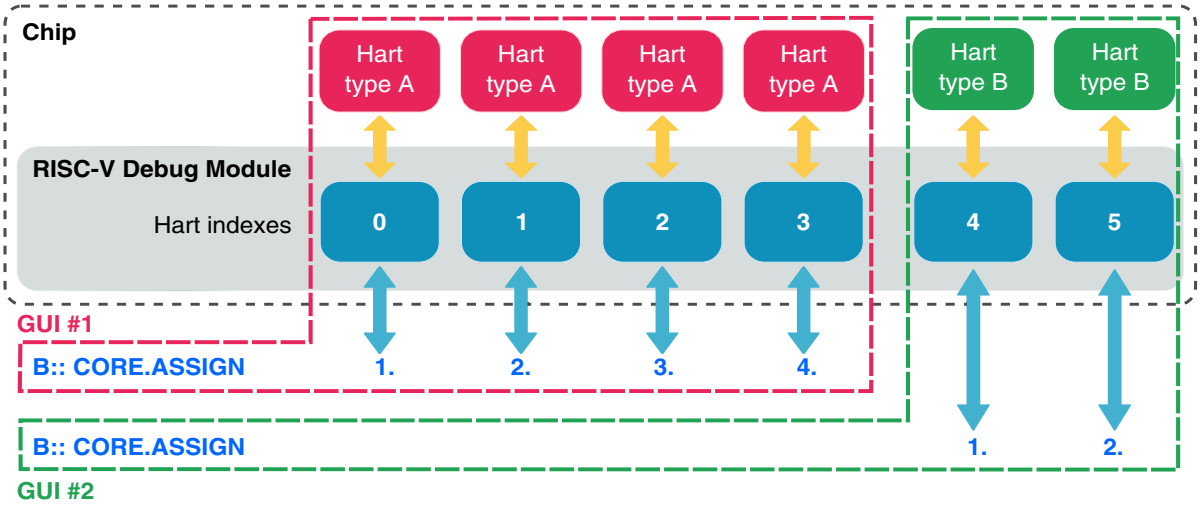
```
; ---- TRACE32 PowerView GUI #1 ----------------------------------------

SYStem.CPU <type_a_cpu>
SYStem.CONFIG CORE 1. 1.                    ; Core group 1 for chip 1
SYStem.CONFIG CoreNumber 6.                 ; 6 harts of type A in total
SYStem.CONFIG HART.INDEX 0. 1. 2. 3. 4. 5.
CORE.ASSIGN 1. 2. 3. 4.                     ; Assign the first 4 harts
```

```
; ---- TRACE32 PowerView GUI #2 ----------------------------------------

SYStem.CPU <type_a_cpu>
SYStem.CONFIG CORE 2. 1.                    ; Core group 2 for chip 1
SYStem.CONFIG CoreNumber 6.                 ; 6 harts of type A in total
SYStem.CONFIG HART.INDEX 0. 1. 2. 3. 4. 5.
CORE.ASSIGN 5. 6.                           ; Assign the last 2 harts
```

Heterogeneous SMP/AMP Debugging

This scenario covers both heterogeneous symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).

6 hardware threads (harts) are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of the RISC-V Debug Module ranging from 0 to 5. The first 4 harts are of type A and will be debugged in an SMP session, and the remaining 2 harts are of type B and will be debugged in another SMP session.



Example D:

```
; ---- TRACE32 PowerView GUI #1 ----------------------------------------

SYStem.CPU <type_a_cpu>
SYStem.CONFIG CORE 1. 1.                    ; Core group 1 for chip 1
SYStem.CONFIG CoreNumber 4.                 ; 4 harts of type A in total
SYStem.CONFIG HART.INDEX 0. 1. 2. 3.        ; Hart indexes of type A
CORE.ASSIGN 1. 2. 3. 4.                     ; Assign all 4 harts of type A
```

```
; ---- TRACE32 PowerView GUI #2 ----------------------------------------

SYStem.CPU <type_b_cpu>
SYStem.CONFIG CORE 2. 1.                    ; Core group 2 for chip 1
SYStem.CONFIG CoreNumber 2.                 ; 2 harts of type B in total
SYStem.CONFIG HART.INDEX 4. 5.              ; Hart indexes of type B
CORE.ASSIGN 1. 2.                           ; Assign all 2 harts of type B
```

Communication between Debugger and Processor cannot be established

Typically **SYStem.Mode Up** or **SYStem.Mode Attach** is the first command of a debug session for which communication with the target board is required. That is why it is the most common command to fail in case that there is any issue with the user configuration, debug connection or target.

NOTE:	In case of any error during the debug session, we highly recommend to open the **AREA.view** window. This window usually contains a list of all recent warnings and error messages, which can be very helpful for error diagnosis.

The error messages in the **AREA.view** window (which can be identified by their red color) usually try to give the user a short error description and a reason for the error. However in some scenarios it can be difficult to deduce the error cause from an error message, because the error message is either too generic or the error message is only the follow-up error of another issue that has nothing to do with the actual error message. In order to still be able to resolve the error in such scenarios, the following lists the most common error causes:

- The target has no power or the debug cable is not connected to the target. This results in the error message “target power fail”.
- You did not select the correct core type via **SYStem.CPU** *<type>*.
- There is an issue with the JTAG interface. See **www.lauterbach.com/adriscv.html** and the manuals or schematic of your target to check the physical and electrical interface. Maybe there is the need to set jumpers on the target to connect the correct signals of the JTAG connector.
- Your RISC-V Debug Module (DM) is mapped on a debug bus, but the base address of the DM is either not configured or incorrect. Check the settings of **SYStem.CONFIG.COREDEBUG.Base**.
- You might have several TAP controllers in your JTAG-chain. Example: The TAP of the JTAG-DTM could be in a chain with other TAPs from other CPUs. In this case you have to check your pre- and post-bit configuration. See for example **SYStem.CONFIG.IRPRE** or **SYStem.CONFIG.DAPIRPRE**.
- The default frequency of the JTAG/SWD/cJTAG debug port is too high, especially if you emulate your core or if you use an FPGA-based target. In this case try **SYStem.JtagClock 50kHz** and optimize the speed when you got it working.
- The target cannot communicate with the debugger while in reset. Try **SYStem.Mode Attach** followed by **Break.direct** instead of **SYStem.Mode Up.**
- The target does not support the configured reset method. Select a different reset method via **SYStem.Option.ResetMode**.
- The target needs a certain setup time during the reset assertion or after the reset release. Try to adapt the reset timing via **SYStem.Option.WaitReset** and/or **SYStem.Option.HoldReset**.
- There is a watchdog which needs to be deactivated.
- There is the need to enable (jumper) the debug features on the target. It will e.g. not work if nTRST signal is directly connected to ground on target side.

- The target is in an unrecoverable state. Re-power your target and try again.
- The core has no power or is kept in reset.
- The core has no clock.

FAQ

Please refer to https://support.lauterbach.com/kb.

Debug Specification for External Debug Support

The Lauterbach debug driver for RISC-V is developed according to the official RISC-V debug specification for external debug support. The latest official version can be found at
https://riscv.org/technical/specifications/

Access Classes

In TRACE32, addresses always consist of two parts:

- An **access class** which defines:
  - What *kind* of memory (or register) to access
  - *How* to perform the access
- A **number** that determines the address of the access

Each access class consists of one or more letters/numbers followed by a colon (:).

Examples:

```
Data.dump D:0x100
Data.dump AXI:0x80000000--0x80000FFF
PRINT Data.Long(CSR:0x300)
```

It is possible to combine individual access classes.

For more background information, see the **chapter about access classes** in the TRACE32 Glossary.

In this section:

Description of the Individual Access Classes

	Description
P	Program memory access. See **SYStem.MemAccessStop** and **SYStem.MemAccess** for the used access method.
D	Data memory access. See **SYStem.MemAccessStop** and **SYStem.MemAccess** for the used access method.
M	Machine privilege level
S	Supervisor privilege level. For debugger memory accesses with this access class, machine privilege level is used.

	Description
U	User privilege level. For debugger memory accesses with this access class, machine privilege level is used.
A	Absolute addressing (physical address) on SoCs with Memory Management Unit (MMU).
C	“Current”. Do not use this access class. It might be shown by the debugger if it is unknown what access class shall be used. The actual used access class is derived from the current processor mode.
CSR	Control and Status Register (CSR) access. The CSR address of this access class does always address data of maximum CSR register width *XLEN*. If a CSR register is smaller than the maximum size, the unused segment gets filled up with zero.
E	Allow memory access while the CPU is running. See **SYStem.MemAccess**, **SYStem.CpuBreak** and **SYStem.CpuSpot**. Any memory access class can be prefixed with **E**, if the memory supports access while the CPU is running.
VM	Virtual Memory (memory on the debug system).
APB	APB bus access. If the APB bus is accessible via an Arm CoreSight DAP (**SoC-400**), see **SYStem.CONFIG APBAP.Port** for details. If the APB bus is accessible via an Arm CoreSight DAP (**SoC-600**), see **SYStem.CONFIG APBAP.Base** for details.
AHB NAHB, ZAHB	AHB bus access. If the AHB bus is accessible via an Arm CoreSight DAP (**SoC-400**), see **SYStem.CONFIG AHBAP.Port** for details. If the AHB bus is accessible via an Arm CoreSight DAP (**SoC-600**), see **SYStem.CONFIG AHBAP.Base** for details.
AXI NAXI, ZAXI	AXI bus access. If the AXI bus is accessible via an Arm CoreSight DAP (**SoC-400**), see **SYStem.CONFIG AXIAP.Port** for details. If the AXI bus is accessible via an Arm CoreSight DAP (**SoC-600**), see **SYStem.CONFIG AXIAP.Base** for details.
SB	System bus access. The memory accesses with this access class are performed via the “System Bus Access block” of the RISC-V Debug Module.

Combination of Several Access Classes

It is possible to combine certain individual access classes for an access. An access class combination can consist of up to five access class specifiers. But any of the five specifiers can also be omitted.

The following **examples** will demonstrate combinations of three access classes:

- **E:** Allow memory access while the CPU is running
- **A:** Physical access, i.e. the MMU is bypassed.
- **D:** Data memory access

Combination of three access class specifiers:
In this example, let's assume...

- You want to view the data memory from the perspective of the CPU:
  Use "D" access class specifier.
- You want to be able to access the data memory independent of whether the CPU is running or halted:
  Use "E" access class specifier.
- You want to make a physical access without any MMU address translation:
  Use "A" access class specifier.

When you put all three access class specifiers together, you will obtain the access class combination "EAD":

```
Data.dump EAD:0x80000000   // Physical data memory access during run-time
```

Combination of two access class specifiers:
In this example, let's assume...

- You want to view the data memory from the perspective of the CPU:
  Use "D" access class specifier.
- You want to be able to access the data memory independent of whether the CPU is running or halted:
  Use "E" access class specifier.
- You want to make a virtual access including MMU address translation:
  Do *not* use "A" access class specifier.

When you put the two access class specifiers together, you will obtain the access class combination "ED":

```
Data.dump ED:0x80000000   // Virtual data memory access during run-time
```

One access class specifier:
In this example, let's assume...

- You want to view the data memory from the perspective of the CPU:
  Use "D" access class specifier.
- You do *not* want to be able to access the data memory while the CPU is running:
  Do *not* use "E" access class specifier.
- You want to make a virtual access including MMU address translation:
  Do *not* use "A" access class specifier.

This means in this case we wo not have a combination of access classes, but instead we simply have the access class "D":

```
Data.dump D:0x80000000  // Virtual data memory access (only when stopped)
```

No access class specifier:
In this example, we will see what happens when you do *not* specify *any* access class at all. In this case the memory access by the debugger will be a virtual access using the *current CPU context*, i.e. the debugger has the same view on memory as the CPU:

```
Data.dump 0x80000000    // Virtual memory access (only when stopped)
```

How to Create Valid Access Class Combinations

There are certain rules on if and how individual access classes can be combined. Only certain access classes can be combined with each other, and they need to be combined in a certain order.

The illustrations below will show you how to combine access class specifiers for frequently-used access class combinations.

Rules to create a valid access class combination:

- From each column of an illustration block, select only one access class specifier.
- You may skip any column - but only if the column in question contains an empty square.
- Do not change the original column order. Recommendation: Put together a valid combination by starting with the left-most column, proceeding to the right.

Memory Access Through CPU (CPU View)

The debugger uses the CPU to access memory, so the CPU carries out the accesses requested by the debugger. This can be either virtual or physical accesses. The accesses can either only happen when the CPU is stopped, or also while the CPU is running.

Example combinations:

ED	Data memory access at run-time
MD	Data memory access with machine privilege level
EMD	Data memory access with machine privilege level at run-time
AP	Physical program memory access

Control and Status Register (CSR) Access

This is used to access the CSRs of a core.

Example combinations:

ECSR	CSR access at run-time

System Bus Access

These accesses grant direct access to system buses, bypassing the CPU.

Example combinations:

EZAXI	Access secure memory location via AXI at run-time
ESB	System bus access of RISC-V debug module at run-time

Breakpoints

For general information about setting breakpoints, refer to the **Break.Set** command.

Software Breakpoints

If a software breakpoint is used, the original instruction at the breakpoint location is temporarily patched by a breakpoint instruction (RISC-V *EBREAK* instruction). There is no restriction in the number of software breakpoints used in a debug session. However, using a software breakpoint requires both read and write access to the respective memory location.

On-chip Breakpoint Resources

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the hardware of the core itself.

For this purpose, a RISC-V core can have generic on-chip triggers that can either be used for on-chip instruction breakpoints or on-chip data breakpoints. These generic triggers are called “address/data match triggers”. The availability of such triggers is optional, and the number of triggers that are available depends on the respective hardware of the core.

This means that on-chip instruction and on-chip data breakpoints share the number of available trigger resources among each other.

One breakpoint can require one or multiple hardware resources, depending on the complexity of the breakpoint.

Example: We have a core with five address/data match trigger resources, and each breakpoint requires exactly one hardware resource. We can either set five on-chip instruction breakpoints, or we could set three instruction breakpoints and two data breakpoints.

On-chip Breakpoints for Instruction Address

On-chip breakpoints for instruction addresses are used to stop the core when an instruction at a certain address is executed.

The resources to set instruction breakpoints are provided by the hardware of the core. For details about the implementation and number of these breakpoints, see chapter **On-chip Breakpoint Resources**.

On-chip instruction breakpoints are particularly useful in scenarios where the program code lies in read-only memory regions such as ROM or flash, as software breakpoints cannot be used in such scenarios. Furthermore breakpoints for instruction address ranges can only be realized with on-chip breakpoints.

On-chip Breakpoints for Data Address

On-chip breakpoints for data addresses are used to stop the core after a read or write access to a memory address.

The resources to set data address breakpoints are provided by the core. For details about the implementation and number of these breakpoints, see chapter **On-chip Breakpoint Resources**.

On-chip data address breakpoints with address range

Some RISC-V on-chip data address breakpoint triggers allow to set triggers for address ranges. Address ranges for on-chip breakpoint of RISC-V can be implemented in two different ways:

- **Address range via address mask**:
  An address range can be expressed with an address mask, if the range matches the following criteria:

  Let the address range be from address A to address B (B inside range), with A < B.
  Let X = A XOR B (infix operator XOR: “exclusive or”).
  Let Y = A AND X (infix operator AND: “logical and”).
  Then all bits in X that equal to one have to be in consecutive order, starting from the least significant bit.
  Then Y has to equal zero.

- **Address range via two addresses**:
  An address range can be expressed with a start address and an end address.

An address range via address mask requires less hardware resources than an address range via two addresses. If the criteria for the address mask are met then the debugger will always automatically choose the mask method, in order to save hardware resources.

Examples:

```
Break.Set 0x0000--0x0FFF /Read        ; Address range suitable for
                                      ; address mask

Break.Set 0x0100--0x01FF /Read        ; Address range suitable for
                                      ; address mask

Break.Set 0x3040--0x307F /Write       ; Address range suitable for
                                      ; address mask

Break.Set 0xA000--0xB0FF /Write       ; Address range suitable for
                                      ; two addresses

Break.Set 0xA000--0xA0FD /Write       ; Address range suitable for
                                      ; two addresses
```

On-chip Data Value Breakpoints

The hardware resources of the core can be used to stop the core when a specific value is read or written:

- **Data Value Breakpoint (Read)**:
  Stop the core when a specific data value is read from a memory address.
- **Data Value Breakpoint (Write)**:
  Stop the core when a specific data value is written to a memory address.

For more information about data value breakpoints, see the **Break.Set** command.

Assume you have a target with

- FLASH from `0x0--0xffff`
- RAM from `0x10000--0x3FFF`

The command **MAP.BOnchip** can be used to inform the debugger for which memory regions breakpoints should only be implemented as on-chip breakpoints. That is why we mark the FLASH region as follows:

```
MAP.BOnchip 0x0--0xffff
```

The following shows examples for setting standard software breakpoints:

```
Break.Set P:0x20100 /Program          ; Software breakpoint on
                                      ; instruction address

Break.Set main /Program               ; Software breakpoint on symbol
```

The following shows examples for setting standard on-chip breakpoints:

```
Break.Set P:0x40 /Program             ; On-chip breakpoint on
                                      ; instruction address.
                                      ; Use on-chip breakpoint because
                                      ; address inside MAP.BOnchip range.

Break.Set P:0x20200 /Program          ; On-chip breakpoint on
                    /Onchip           ; instruction address.
                                      ; Use on-chip breakpoint because
                                      ; of explicit '/Onchip' option.

Break.Set P:0x40--0x48 /Program       ; On-chip breakpoint on
                                      ; instruction address range

Break.Set D:0x1010 /Read              ; On-chip read breakpoint on
                                      ; data address

Break.Set D:0x1020 /Write             ; On-chip write breakpoint on
                                      ; data address

Break.Set D:0x1030 /ReadWrite         ; On-chip read and write breakpoint
                                      ; on data address

Break.Set D:0x1010--0x101F /Read      ; On-chip read breakpoint on
                                      ; data address range

Break.Set D:0x10 /Read                ; On-chip read breakpoint on
                 /DATA.Long 0x123     ; data address, combined with
                                      ; condition for read data value
```

Floating-Point Extensions

The Lauterbach debugger for RISC-V provides support for floating-point extensions of the RISC-V ISA. This covers both the single-precision floating-point extension (“F” extension) and the double-precision floating-point extension (“D” extension).

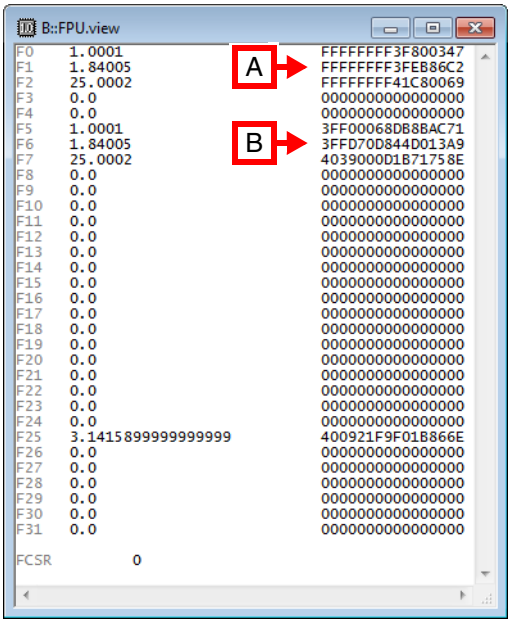
The floating-point features are provided by the **FPU** (Floating-Point Unit) command group.

The **FPU.view** window does display the floating-point registers. Depending on whether the core under debug supports single-precision or double-precision, the **FPU.view** window automatically adjusts its register width.

RISC-V floating-point extensions are compliant with the IEEE 754-2008 arithmetic standard. Cores that support the double-precision extension do automatically support the single-precision extension as well. The RISC-V ISA specification defines that a 32 bit single-precision value is stored in a 64 bit double-precision floating-point register by filling up the upper 32 bits of the register with all 1s (Not a Number (NaN) boxing).

When modifying values with **FPU.Set**, the user can decide in which floating-point precision notation the value is written.

The **FPU.view** window does automatically display register values with NaN boxing in single-precision representation, and register values without NaN boxing in double-precision representation. The following example shows 64 bit floating-point registers that contain the same values in both single-precision and double-precision representation:



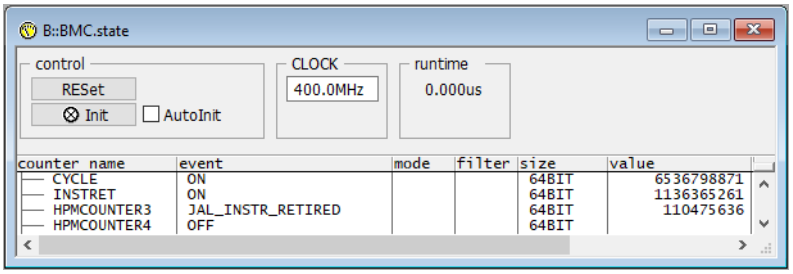
A Single-precision representation

B Double-precision representation

Hardware Performance Monitor

The RISC-V ISA defines a so-called “Hardware Performance Monitor”, which consists of several hardware counters (mcycle, minstret, mhpmcounter, ...). The existence of such a monitor and its counters is optional, so it may not be available in all RISC-V devices.

The Lauterbach **BMC** (**B**ench**M**ark **C**ounter) command group does provide control and usage of these hardware performance counters, if available on the chip:



For information about *architecture-independent* **BMC** commands, refer to **“BMC”** (general_ref_b.pdf).

Hart State: Unavailable

The RISC-V Debug Module can flag a RISC-V hart as “unavailable”, by setting the respective *allunavail/anyunavail* status bits of the *dmstatus* debug register.

The RISC-V debug specification says: “Harts may be unavailable for a variety of reasons including being reset, temporarily powered down, and not being plugged into the hardware platform.”

If the debugger detects that a hart is currently flagged as unavailable, then it will display “unavailable” in the bottom-right corner of the TRACE32 state line :

As long as the hart is in this state, it is *not* possible to manually halt the hart via the **Break.direct** command.

NOTE:	In order to poll the state of a hart, the debugger needs to have full access to all debug registers of the RISC-V debug module. This means that even when a RISC-V hart is for example in reset or power-down, then the debug IP such as RISC-V Debug Module, Debug Module Interface (DMI), etc should still be active and available to the debugger.If however the debug IP, including the JTAG connector, gets powered down as well (which is **not recommended**), then please refer to SYStem.Mode.StandBy.

Semihosting

Semihosting is a technique for application programs running on a RISC-V processor to communicate with the host computer of the debugger. This way the application can use the I/O facilities of the host computer like keyboard input, screen output, and file I/O. This is especially useful if the target platform does not yet provide these I/O facilities or in order to output additional debug information in printf() style.

The RISC-V semihosting is based on the "Semihosting for AArch32 and AArch64: Release 2.0" specification available here: http://documentation-service.arm.com/static/5f905528f86e16515cdc1d25

A RISC-V semihosting call is invoked by the following **semihosting trap instruction sequence**:

```
slli x0, x0, 0x1f     # 0x01f01013   Entry NOP
ebreak                # 0x00100073   Break to debugger
srai x0, x0, 7        # 0x40705013   NOP encoding the semihosting call #7
```

Semihosting register definitions:

- Operation number register: a0
- Parameter register: a1
- Return register: a0
- Data block field size: 32bits for RV32, 64bits for RV64

There is no need to set any additional breakpoints since the ebreak instruction itself will stop the core. The debugger will restart the core after the semihosting data is processed.

Semihosting for RISC-V is enabled by **TERM.METHOD RISCVSWI** and by opening a **TERM.GATE** window for the semihosting screen output. The handling of the semihosting requests is only active when the **TERM.GATE** window does exist.

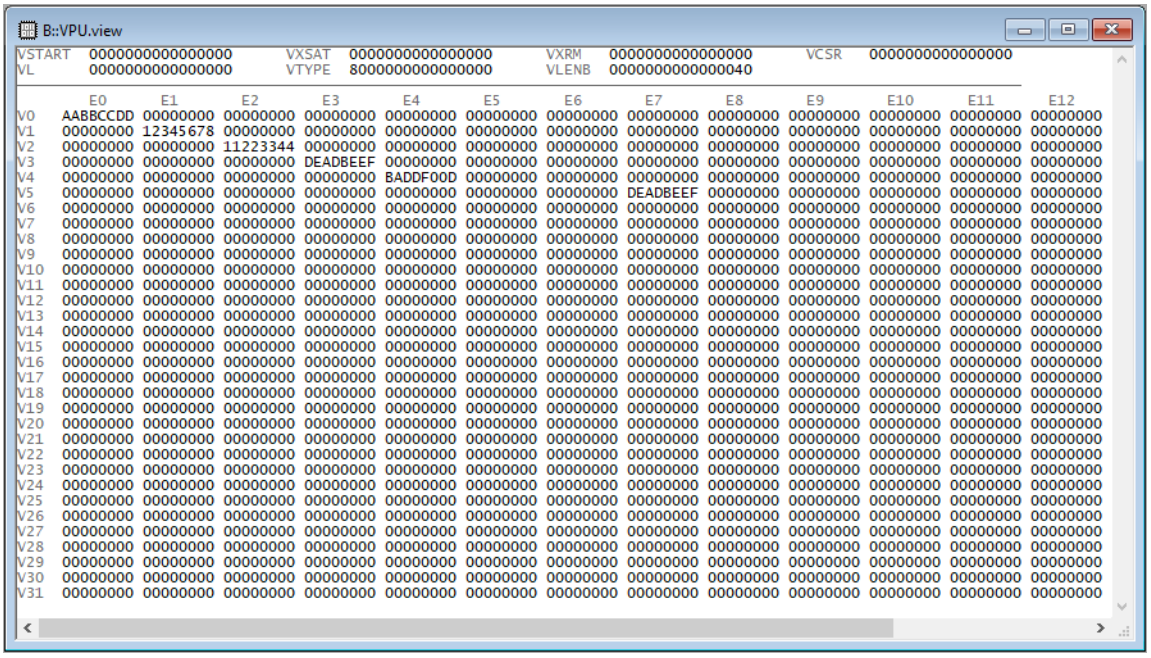
Vector Extension

The Lauterbach debugger for RISC-V provides support for the vector register extension (“V” extension) of the RISC-V ISA.

The vector features are provided by the **VPU** (Vector Processing Unit) command group. This command group is only unlocked if the RISC-V target does support the vector extension.

The **VPU.view** window does display the vector registers. The vector register width *VLEN* is automatically detected by the debugger, and the width of the vector registers in the **VPU.view** window is adjusted accordingly.

As RISC-V vector registers (*v0 - v31*) can be quite large, the debugger displays them in sub-elements (*E0 - En*), with a width of 32bits for each sub-element.



When modifying values with **VPU.Set**, the user can write to each sub-element (*E0 - En*) of a vector register individually.

Example:

```
VPU.Set V8_E2 0x12345678   ; Write to vector register v8, sub-element #2
```

SETUP.DIS Disassembler configuration

Format:	**SETUP.DIS** [*<fields>*] [*<bar>*] [*<constants>*]
<constants>:	[**RegNames** | **AbiNames**] [*<other_constants>*]

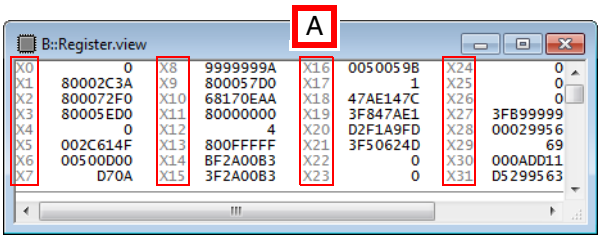
Sets **default values** for configuring the disassembler output of **newly opened windows**. Affected windows and commands are **List.Asm**, **Register.view**, and **Register.Set**.

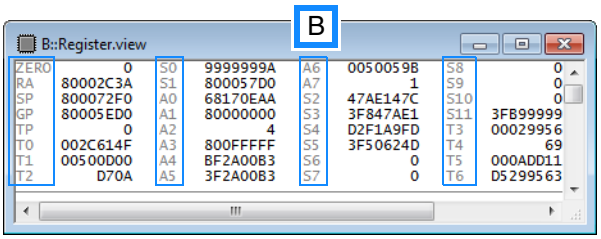
The command does **not affect existing windows** containing disassembler output.

<fields>, *<bar>*, *<constants>*	For a description of the generic arguments, see **SETUP.DIS** in **general_ref_s.pdf**.
AbiNames	Use the *ABI* (application binary interface) naming scheme for the names of the RISC-V general purpose registers.
RegNames (default naming scheme)	Use the *register number* (x0, x1, …, x31) naming scheme for the names of the RISC-V general purpose registers.

Example 1: The changed naming scheme takes immediate effect in the **Register.view** window.

```
SETUP.DIS RegNames    ;by default, the register number naming scheme is
                      ;used for the general purpose registers
Register.view         ;let's open a register window

;... your code

SETUP.DIS AbiNames    ;let's now switch the naming scheme of the general
                      ;purpose registers to the ABI naming scheme
```



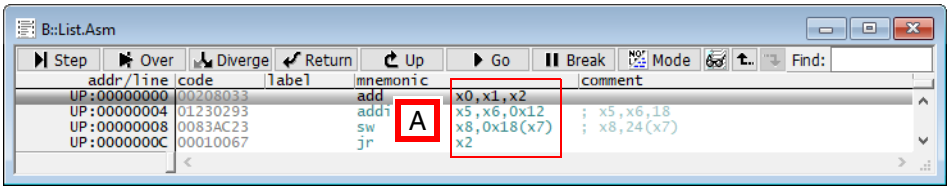


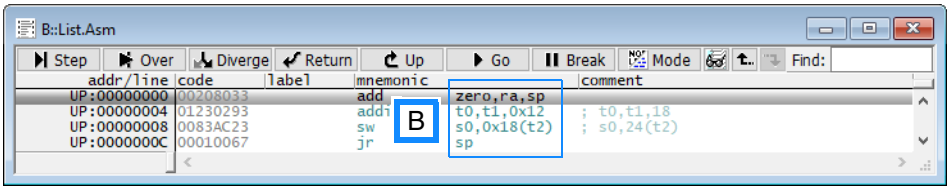
A Register number naming scheme.

B ABI naming scheme. The ABI names are also available as aliases in **Register.Set**.

Example 2: The changed naming scheme does **not affect an existing List.Asm** window. You need to open another **List.Asm** window to view the changed naming scheme.

```
SETUP.DIS RegNames
List.Asm

;... your code

SETUP.DIS AbiNames
List.Asm              ;open another disassembler output window
```





A Register number naming scheme (default naming scheme).

B ABI naming scheme. The ABI names are also available as aliases in **Register.Set**.

SYStem.CONFIG.state — Display target configuration

Format:	**SYStem.CONFIG.state** [**/***<tab>*]
<tab>:	**DebugPort** | **Jtag** | **AccessPorts** | **COmponents**

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger's operations.

Alternatively, you can modify the target configuration settings via the TRACE32 command line with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<tab>	Opens the **SYStem.CONFIG.state** window on the specified tab. For tab descriptions, see below.
DebugPort (default)	The **DebugPort** tab informs the debugger about the debug connector type and the communication protocol it shall use. For descriptions of the commands on the **DebugPort** tab, see **DebugPort**.
Jtag	The **Jtag** tab informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip. For descriptions of the commands on the **Jtag** tab, see **Jtag**.
AccessPorts	This tab informs the debugger about an Arm CoreSight Access Port (AP) and about how to control the AP to access chip-internal memory busses (AHB, APB, AXI) or chip-internal JTAG interfaces. For a descriptions of a corresponding commands, refer to **AP**.

COmponents	The **COmponents** tab informs the debugger (a) about the existence and interconnection of on-chip debug and trace modules and (b) informs the debugger on which memory bus and at which base address the debugger can find the control registers of the modules. For descriptions of the commands on the **COmponents** tab, see **COmponents**.

SYStem.CONFIG Configure debugger according to target topology

Format: **SYStem.CONFIG** *<parameter>*

<parameter>: **(DebugPort)**
CJTAGFLAGS *<flags>*
CJTAGTCA *<value>*
CONNECTOR [**MIPI34** | **MIPI20T**]
CORE *<core>* *<chip>*
CoreNumber *<number>*
DEBUGPORT [**DebugCable0**]
DEBUGPORTTYPE [**JTAG** | **CJTAG** | **SWD**]
Slave [**ON** | **OFF**]
SWDP [**ON** | **OFF**]
SWDPIdleHigh [**ON** | **OFF**]
SWDPTargetSel *<value>*
TriState [**ON** | **OFF**]

<parameter>: **(JTAG)**
DAPDRPOST *<bits>*
DAPDRPRE *<bits>*
DAPIRPOST *<bits>*
DAPIRPRE *<bits>*
DRPOST *<bits>*
DRPRE *<bits>*
IRPOST *<bits>*
IRPRE *<bits>*
IRWIDTH *<bits>*
Slave [**ON** | **OFF**]
TAPState *<state>*
TCKLevel *<level>*
TriState [**ON** | **OFF**]

<parameter>: **(AccessPorts)**
AHBAPn.HPROT [*<value>* | *<name>*]
AHBAPn.RESet
AHBAPn.view
AHBAPn.XtorName *<name>*

<parameter>: (AccessPorts cont.)	**APBAPn.RESet** **APBAPn.view** **APBAPn.XtorName** *<name>*
	AXIAPn.ACEEnable [**ON** | **OFF**] **AXIAPn.CacheFlags** *<value>* **AXIAPn.HPROT** [*<value>* | *<name>*] **AXIAPn.RESet** **AXIAPn.view** **AXIAPn.XtorName** *<name>*
	JTAGAPn.RESet **JTAGAPn.view** **JTAGAPn.XtorName** *<name>*
<parameter>: **(AccessPorts SoC-400)**	**AHBAP.Port** *<port>* **APBAP.Port** *<port>* **AXIAP.Port** *<port>* **JTAGAPn.Port** *<port>* **JTAGAPn.CorePort** *<port>*
<parameter>: **(AccessPorts SoC-600)**	**AHBAP.Base** *<address>* **APBAP.Base** *<address>* **AXIAP.Base** *<address>* **JTAGAPn.Base** *<address>*
<parameter>: **(COmponents)**	**COREDEBUG.Base** <address> **COREDEBUG.RESet** **COREDEBUG.view**
	ETR.CATUBase *<address>*
<parameter>: **(Tessent Embedded Analytics)**	**COREJPAM** *<name>*

The **SYStem.CONFIG** commands inform the debugger about the available on-chip debug and trace components and how to access them.

The **SYStem.CONFIG** command information shall be provided after the **SYStem.CPU** command, which might be a precondition to enter certain **SYStem.CONFIG** commands, and before you start up the debug session, e.g. by **SYStem.Up**.

Syntax Remarks

The commands are not case sensitive. Capital letters show how the command can be shortened.
Example: “SYStem.CONFIG.TriState ON” -> “SYStem.CONFIG.TS ON”

The dots after “SYStem.CONFIG” can alternatively be a blank.
Example:
“SYStem.CONFIG.TriState ON” or “SYStem.CONFIG TriState ON”

CJTAGFLAGS *<flags>*

Activates workarounds for incomplete or buggy cJTAG (IEEE 1149.7) implementations.
Bit 0: Disable scanning of cJTAG ID (TCA-scanning).
Bit 1: Target has no "keeper". Use TRACE32 pseudo keeper.
Bit 2: Inverted meaning of SREDGE register.
Bit 3: Old command opcodes (cJTAG < 1.14).
Bit 4: APFC unlock required.
Bit 5: OAC required

Default: 0

CJTAGTCA *<value>*

Selects the TCA (TAP Controller Address) to address a device in a cJTAG (IEEE 1149.7) Star-2 configuration. The Star-2 configuration requires a unique TCA for each device on the debug port.

CONNECTOR [**MIPI34** | **MIPI20T**]

Specifies the connector "MIPI34" or "MIPI20T" on the target. This is mainly needed in order to notify the trace pin location.

Default: MIPI34 if CombiProbe is used, MIPI20T if µTrace (MicroTrace) is used.

CORE *<core>* *<chip>*

The command helps to identify debug and trace resources which are commonly used by different cores. The command might be required in a multicore environment if you use multiple debugger instances (multiple TRACE32 PowerView GUIs) to simultaneously debug different cores on the same target system.

Because of the default setting of this command

debugger#1: *<core>*=1 *<chip>*=1
debugger#2: *<core>*=1 *<chip>*=2
...

each debugger instance assumes that all notified debug and trace resources can exclusively be used.

But some target systems have shared resources for different cores, for example a common trace port. The default setting causes that each debugger instance controls the same trace port. Sometimes it does not hurt if such a module is controlled twice. But sometimes it is a must to tell the debugger that these cores share resources on the same *<chip>*. Whereby the "chip" does not need to be identical with the device on your target board:

debugger#1: *<core>*=1 *<chip>*=1
debugger#2: *<core>*=2 *<chip>*=1

CORE *<core>* *<chip>*

(cont.)

For cores on the same *<chip>,* the debugger assumes that the cores share the same resource if the control registers of the resource have the same address.

Default:
<core> depends on CPU selection, usually 1.
<chip> derives from the `CORE=` parameter in the configuration file (config.t32), usually 1. If you start multiple debugger instances with the help of t32start.exe, you will get ascending values (1, 2, 3,...).

CoreNumber *<number>*

Number of cores to be considered in an SMP (symmetric multiprocessing) debug session. There are RISC-V core types which can be used as a single core processor or as a scalable multicore processor of the same type. If you intend to debug more than one such core in an SMP debug session you need to specify the number of cores you intend to debug.

Default: 1.

DEBUGPORT [**DebugCable0**]

It specifies which probe cable shall be used e.g. “DebugCable0”. At the moment only the CombiProbe allows to connect more than one probe cable.

Default: depends on detection.

DEBUGPORTTYPE [**JTAG** | **CJTAG** | **SWD**]

It specifies the used debug port type “JTAG”, “CJTAG” or “SWD”. It assumes the selected type is supported by the target.

Default: JTAG.

Slave [**ON** | **OFF**]

If several TRACE32 debugger GUIs share the same debug port (AMP debugging), all GUIs except one must have this option set to **ON**.

JTAG: Only one debugger GUI - the “master GUI” - is allowed to control the signals nTRST and nSRST (nRESET), and perform a test-logic-reset or system reset. Only this master GUI must have **Slave OFF**.

All other debugger GUIs are considered “slave GUIs” and need to have the setting **Slave ON**.

Default: OFF.
Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).

SWDPIdleHigh [**ON** | **OFF**]

Keep SWDIO line high when idle. Only for Serialwire Debug mode. Usually the debugger will pull the SWDIO data line low, when no operation is in progress, so while the clock on the SWCLK line is stopped (kept low).

You can configure the debugger to pull the SWDIO data line high, when no operation is in progress by using **SYStem.CONFIG SWDPIdleHigh ON**

Default: OFF.

SWDPTargetSel *<value>*

Device address in case of a multidrop serial wire debug port.

Default: none set (any address accepted).

TriState [**ON** | **OFF**]

TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode.
Please note:

- nTRST must have a pull-up resistor on the target.
- TCK can have a pull-up or pull-down resistor.
- Other trigger inputs need to be kept in inactive state.

Default: OFF.

With the JTAG interface you can access a Test Access Port controller (TAP) which has implemented a state machine to provide a mechanism to read and write data to an Instruction Register (IR) and a Data Register (DR) in the TAP. The JTAG interface will be controlled by 5 signals:

- nTRST (reset)
- TCK (clock)
- TMS (state machine control)
- TDI (data input)
- TDO (data output)

Multiple TAPs can be controlled by one JTAG interface by daisy-chaining the TAPs (serial connection). If you want to talk to one TAP in the chain, you need to send a BYPASS pattern (all ones) to all other TAPs. For this case the debugger needs to know the position of the TAP it wants to talk to.

The width of the JTAG instruction register of the TAP of a **RISC-V JTAG Debug Transport Module (JTAG-DTM)** can be defined with **IRWIDTH**.

The TAP position of a **RISC-V JTAG Debug Transport Module (JTAG-DTM)** can be defined with the commands **IRPRE**, **IRPOST**, **DRPRE**, and **DRPOST**.

The TAP position of an **Arm CoreSight Debug Access Port (Arm DAP)** can be defined with the commands **DAPIRPRE**, **DAPIRPOST**, **DAPDRPRE**, and **DAPDRPOST**.

DRPOST *<bits>*
Defines the TAP position of the RISC-V JTAG-DTM in a JTAG scan chain. Number of TAPs in the JTAG chain between the TDI signal and the TAP you are describing. In BYPASS mode, each TAP contributes one data register bit. See example below.

Default: 0.

DRPRE *<bits>*
Defines the TAP position of the RISC-V JTAG-DTM in a JTAG scan chain. Number of TAPs in the JTAG chain between the TAP you are describing and the TDO signal. In BYPASS mode, each TAP contributes one data register bit. See example below.

Default: 0.

IRPOST *<bits>*
Defines the TAP position of the RISC-V JTAG-DTM in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between TDI signal and the TAP you are describing. See example below.

Default: 0.

IRPRE *<bits>*

Defines the TAP position of the RISC-V JTAG-DTM in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the TAP you are describing and the TDO signal. See example below.

Default: 0.

IRWIDTH *<bits>*

Defines the JTAG Instruction Register (IR) width of the JTAG TAP of the RISC-V JTAG-DTM. See example below.

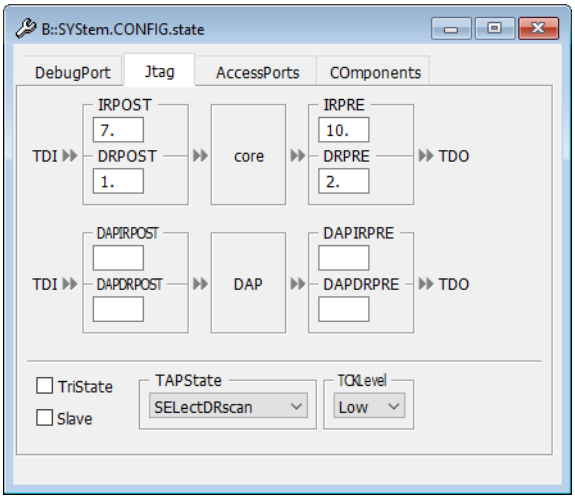
Default: 5.

NOTE: If you are not sure about your settings concerning **IRPRE**, **IRPOST**, **DRPRE**, and **DRPOST**, you can try to detect the settings automatically with the **SYStem.DETECT.DaisyChain** or **SYStem.DETECT.SHOWChain** command.

Example:

This example shows four TAPs in a JTAG daisy chain. The relevant TAP for RISC-V debugging is the *JTAG Debug Transport Module* (JTAG-DTM) TAP. In order to address this TAP, the following settings are necessary:

```
SYStem.CONFIG IRWIDTH 5.
SYStem.CONFIG IRPRE  10.
SYStem.CONFIG IRPOST  7.
SYStem.CONFIG DRPRE   2.
SYStem.CONFIG DRPOST  1.
```



If your system contains an **Arm CoreSight Debug Access Port (DAP)** and the DAP is accessible via JTAG, then the DAP's JTAG Test Access Port controller (TAP) may be inside a JTAG daisy-chain together with other TAPs. To tell the debugger the exact position of the DAP's TAP within the JTAG daisy-chain, you will require the commands DAPIRPRE, DAPIRPOST, DAPDRPRE, and DAPDRPOST. These settings are especially important if the CoreSight DAP is not only used to access memory, but also to access the debug registers of the RISC-V Debug Module.

DAPDRPOST *<bits>*
(default: 0) *<number>* of TAPs in the JTAG chain between the DAP and the TDO signal of the debugger.

DAPDRPRE *<bits>*
(default: 0) *<number>* of TAPs in the JTAG chain between the TDI signal of the debugger and the DAP.

DAPIRPOST *<bits>*

(default: 0) *<number>* of instruction register bits in the JTAG chain between the DAP and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the DAP and the TDO signal of the debugger.

DAPIRPRE *<bits>*

(default: 0) *<number>* of instruction register bits in the JTAG chain between the TDI signal and the DAP. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the DAP.

Slave [**ON** | **OFF**]

If several debuggers share the same debug port, all except one must have this option active.

JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**.

Default: OFF.
Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).

TAPState *<state>*

This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.

During an AMP debug session, this parameter must be set to the same value in all TRACE32 instances.

0 Exit2-DR
1 Exit1-DR
2 Shift-DR
3 Pause-DR
4 Select-IR-Scan
5 Update-DR
6 Capture-DR
7 Select-DR-Scan
8 Exit2-IR
9 Exit1-IR
10 Shift-IR
11 Pause-IR
12 Run-Test/Idle
13 Update-IR
14 Capture-IR
15 Test-Logic-Reset

Default: 7 = Select-DR-Scan.

TCKLevel *<level>*

Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target.

Default: 0.

TriState [**ON** | **OFF**]

TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode.
Please note:

- nTRST must have a pull-up resistor on the target.
- TCK can have a pull-up or pull-down resistor.
- Other trigger inputs need to be kept in inactive state.

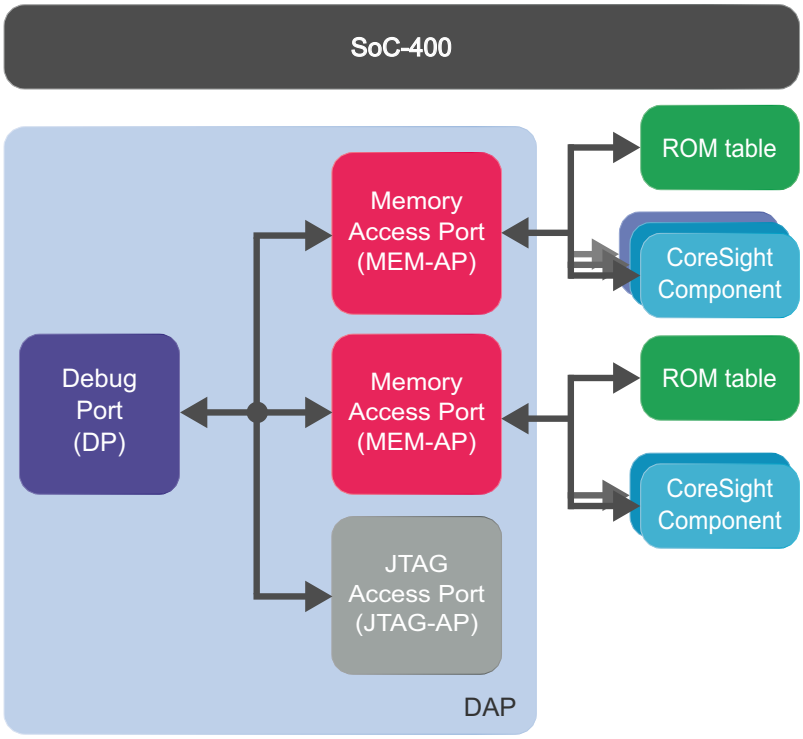
Default: OFF.

An Access Port (AP) is a CoreSight module from Arm which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

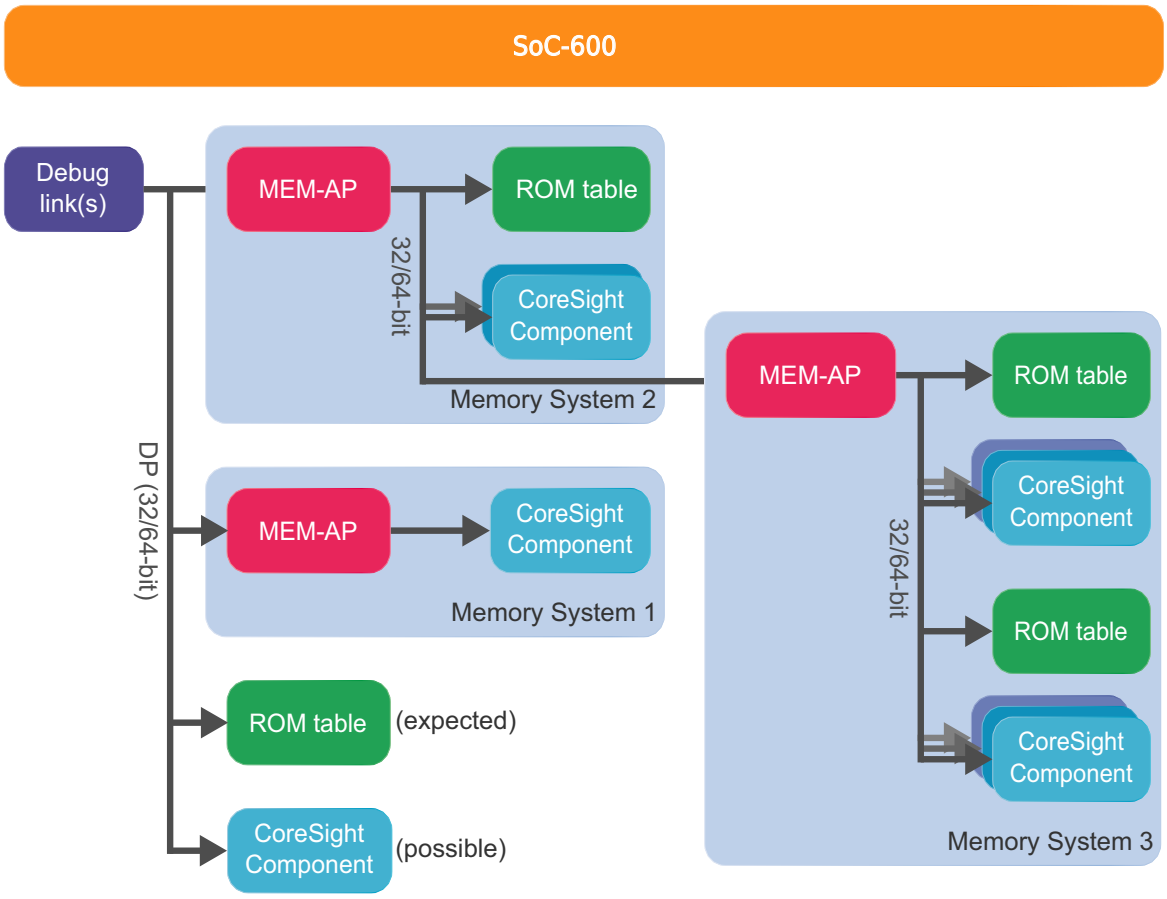
1. **Memory busses** (AHB, APB, AXI). This is especially important if the on-chip debug register needs to be accessed this way. You can access the memory buses by using certain access classes with the debugger commands: “AHB:”, “APB:”, “AXI:. The interface to these buses is called Memory Access Port (MEM-AP).
   The debug registers of some cores are accessible via such a memory bus (mostly APB).

2. A transactor name for virtual connections to AMBA bus level transactors can be configured by the property **SYStem.CONFIG.*APn.XtorName** *<name>*. A JTAG or SWD transactor must be configured for virtual connections to use the property “Port” or “Base” (with “DP:” access) in case XtorName remains empty.

Example 1: SoC-400



Example 2: SoC-600



AHBAPn.HPROT [*<value>* | *<name>*]
SYStem.Option.AHBH-PROT [*<value>* | *<name>*] (deprecated)

Default: 0.
Selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight AHB Access Port, when using the AHB: memory class.

AXIAPn.HPROT [*<value>* | *<name>*]
SYStem.Option.AXIHPROT [*<value>* | *<name>*] (deprecated)

Default: 0.
This option selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight AXI Access Port, when using the AXI: memory class.

AXIAPn.ACEEnable [**ON** | **OFF**]
SYStem.Option.AXIACEEnable [**ON** | **OFF**] (deprecated)

Default: OFF.
Enables ACE transactions on the AXI-AP, including barriers. This does only work if the debug logic of the target CPU implements coherent accesses. Otherwise this option will be without effect.

AXIAPn.CacheFlags *<value>*
SYStem.Option.AXICACHEFLAGS *<value>* (deprecated)

Default: DeviceSYStem (=0x30: Domain=0x3, Cache=0x0).
This option configures the value used for the Cache and Domain bits in the Control Status Word (CSW[27:24]->Cache, CSW[14:13]->Domain) of an Access Port, when using the AXI: memory class.

The below offered selection options are all non-bufferable. Alternatively you can enter a <value>, where value[5:4] determines the Domain bits and value[3:0] the Cache bits.

<name>	Description
DeviceSYStem	=0x30: Domain=0x3, Cache=0x0
NonCacheableSYStem	=0x32: Domain=0x3, Cache=0x2
ReadAllocateNonShareable	=0x06: Domain=0x0, Cache=0x6
ReadAllocateInnerShareable	=0x16: Domain=0x1, Cache=0x6
ReadAllocateOuterShareable	=0x26: Domain=0x2, Cache=0x6
WriteAllocateNonShareable	=0x0A: Domain=0x0, Cache=0xA
WriteAllocateInnerShareable	=0x1A: Domain=0x1, Cache=0xA
WriteAllocateOuterShareable	=0x2A: Domain=0x2, Cache=0xA
ReadWriteAllocateNonShareable	=0x0E: Domain=0x0, Cache=0xE
ReadWriteAllocateInnerShareable	=0x1E: Domain=0x1, Cache=0xE
ReadWriteAllocateOuterShareable	=0x2E: Domain=0x2, Cache=0xE

... .RESet

Undo the configuration for this access port. This does not cause a physical reset for the access port on the chip.

... .view

Opens a window showing the current configuration of the access port.

AHBAPn.XtorName *<name>*
AHB bus transactor name that shall be used for “AHBn:” access class.

APBAPn.XtorName *<name>*
APB bus transactor name that shall be used for “APBn:” access class.

AXIAPn.XtorName *<name>*
AXI bus transactor name that shall be used for “AXIn:” access class.

SoC-400 Specific Commands

In an **Arm SoC-400** system, the following **SYStem.CONFIG** commands configure the port-number for the memory busses:

AHBAPn.Port *<port>*
AHBACCESSPORT *<port>* (deprecated)
Access Port Number (0-255) of a SoC-400 system which shall be used for “AHBn:” access class.
Default: port not available.

APBAPn.Port *<port>*
APBACCESSPORT *<port>* (deprecated)
Access Port Number (0-255) of a SoC-400 system which shall be used for “APBn:” access class.
Default: port not available.

AXIAPn.Port *<port>*
AXIACCESSPORT *<port>* (deprecated)
Access Port Number (0-255) of a SoC-400 system which shall be used for “AXIn:” access class.
Default: port not available.

JTAGAPn.CorePort *<port>*
COREJTAGPORT *<port>* (deprecated)
JTAG-AP port number (0-7) connected to the core which shall be debugged.

JTAGAPn.Port *<port>*
JTAGACCESSPORT *<port>* (deprecated)
Access port number (0-255) of a SoC-400 system of the JTAG Access Port.

AHBAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “AHBAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.AHBAP1.Base DP:0x80002000
Meaning: The control register block of the AHB access ports starts at address 0x80002000.

APBAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “APBAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.APBAP1.Base DP:0x80003000
Meaning: The control register block of the APB access ports starts at address 0x80003000.

AXIAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “AXIAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.AXIAP1.Base DP:0x80004000
Meaning: The control register block of the AXI access ports starts at address 0x80004000.

JTAGAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “JTAGAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.JTAGAP1.Base DP:0x80005000
Meaning: The control register block of the JTAG access ports starts at address 0x80005000.

It is possible to configure multiple Arm SoC-600 buses of one type (e.g. multiple APB buses). This is only necessary if all these buses need to be accessed from within the same TRACE32 PowerView GUI (i.e. from the same SMP session). To do so, each bus can be given its individual bus index.

If no explicit bus index is specified during configuration or use of an access class, then the debugger will automatically imply and assume the index value 1.

Example:

```
SYStem.CONFIG.APBAP1.Base DP:0x1000000  ; first  APB AP: index 1
SYStem.CONFIG.APBAP2.Base DP:0x2000000  ; second APB AP: index 2
SYStem.CONFIG.AXIAP.Base  DP:0x3000000  ; first AXI AP: index 1 (implied)

Data.dump APB:0x80000000            ; use access class of first APB AP
Data.dump APB2:0x90000000           ; use access class of second APB AP
Data.dump AXI:0x30000000            ; use access class of first AXI AP
```

On the **Components** tab in the **SYStem.CONFIG.state** window, you can comfortably add the debug and trace components your chip includes and which you intend to use with the debugger's help.

Components and Available Commands

SYStem.CONFIG.COREDEBUG.Base <address>
SYStem.CONFIG.COREDEBUG.RESet
SYStem.CONFIG.COREDEBUG.view

RISC-V Debug Module: bus type and base address of bus-mapped debug registers.

In some systems the debug registers of the RISC-V Debug Module (DM) are mapped on a debug bus (*without* the use of a JTAG-DTM). In that case this command configures the bus type and the base address of the DM register address space.

Example:

RISC-V DM debug registers mapped on APB bus with base address 0x80000000:

```
SYStem.CONFIG.COREDEBUG.Base APB:0x80000000
```

For further examples, see **"Debug Module Access via Debug Bus"**, page 17.

… **.Base** *<address>*	Configure the start address of the debug register block of the RISC-V debug module. Configuring this address also notifies the debugger that the debug registers are directly accessible for the debugger via memory (and not e.g. via a JTAG-DTM).
… **.RESet**	Undo the configuration of the start address of the debug register block of the RISC-V debug module. This does not cause a physical reset for the component on the chip.
… **.view**	Opens a window showing the current configuration of this command.

<parameters> describing Tessent Embedded Analytics details

For more information on the usage of Tessent Embedded Analytics with TRACE32, see **"Tessent Embedded Analytics Debugger Setup"** manual.

… **.COREJPAM** *<name>*

Configure a Tessent Embedded Analytics **J**TAG **P**rocessor **A**nalytic **M**odule to access the RISC-V JTAG-DTM for debugging. Use the name of an already configured JPAM.

Format:	**SYStem.CONFIG.HART.INDEX** *<index>* **SYStem.CONFIG.HARTINDEX** *<index>* (deprecated)
<index>:	**0.** | **1.** … *n*

Default: 0.

Configures the hardware thread index (hart index) that is used by the RISC-V Debug Module to interact with a specific hart.

The command requires a hart index for each hart that is covered by **SYStem.CONFIG.CoreNumber**.

Example:

```
SYStem.CONFIG.CoreNumber 5.
SYStem.CONFIG.HART.INDEX 3. 4. 5. 6. 7.
```

The Debug Module “hart index” should not be confused with other values such as the “hart ID” of the *mhartid* CSR.

For further examples, see **“Quick Start for Multicore Debugging”**, page 19.

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**RV32** | **RV64** | …

Selects the target core / CPU / SoC / chip to be debugged.

<cpu>	For a list of supported cores/CPUs/SoCs/chips, use the command "`SYStem.CPU *`" or refer to the chip search on the Lauterbach website.

NOTE:	**RV32** and **RV64** are *default* entries for 32-bit and 64-bit RISC-V cores, respectively. These entries should only be selected if there is no dedicated *<cpu>* entry available that matches the target. If RV32/RV64 is selected then all chip-specific configuration needs to be made manually by the user.

NOTE:	All core entries have a prefix that is specific to the core vendor, in order to prevent naming collisions. Example: the E21 core from SiFive has the name "SF-E21".

Format: **SYStem.JtagClock** [<*frequency*> | **RTCK** | **ARTCK** <*frequency*> | **CTCK** <*frequency*> | **CRTCK** <*frequency*>]
SYStem.BdmClock <*frequency*> (deprecated)

<*frequency*>: **10000.** … **40000000.**

Default frequency: 10 MHz.

Selects the JTAG port frequency (TCK) used by the debugger to communicate with the processor. The frequency affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the JTAG lines or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer.

<*frequency*> The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window.
Besides a decimal number like “100000.” short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.

RTCK The JTAG clock is controlled by the RTCK signal (**R**eturned **TCK**)**.** The debugger does not progress to the next TCK edge until after an RTCK edge is received. This mode is not recommended for this debugger since it is not needed here.

ARTCK Accelerated method to control the JTAG clock by the RTCK signal (Accelerated Returned TCK). In ARTCK mode the debugger uses a fixed JTAG frequency for TCK, independent of the RTCK signal. This frequency must be specified by the user. TDI and TMS will be delayed by 1/2 TCK clock cycle. TDO will be sampled with RTCK. This mode is not recommended for this debugger since it is not needed here.

CTCK With this option higher JTAG speeds can be reached. The TDO signal will be sampled by a signal which derives from TCK, but which is timely compensated regarding the debugger-internal driver propagation delays (**C**ompensation by **TCK**).

CRTCK With this option higher JTAG speeds can be reached. The TDO signal will be sampled by the RTCK signal. This compensates the debugger-internal driver propagation delays, the delays on the cable and on the target (**C**ompensation by **RTCK**). This feature requires that the target sends back the TCK signal onto the RTCK signal. In contrast to the **RTCK** option, the TCK is always output with the selected, fixed frequency.

Format: **SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked, no access to the JTAG port will be performed by the debugger. While locked the JTAG connector of the debugger is tristated. The intention of the **SYStem.LOCK** command is, for example, to give JTAG access to another tool. The process can also be automated, see **SYStem.CONFIG TriState**.

It must be ensured that the state of the RISC-V DTM JTAG state machine remains unchanged while the system is locked. To ensure correct hand-over, the options **SYStem.CONFIG TAPState** and **SYStem.CONFIG TCKLevel** must be set properly. They define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

Format: **SYStem.MemAccess** *<method>*

<method>: **Denied**
SB
StopAndGo

Default: Denied.

This command defines if and how memory can be accessed with the **"D:"** and **"P:"** **access classes** while the CPU is **running**.

NOTE: This command only takes effect while the CPU is *running*. For memory access while the CPU is stopped, see **SYStem.MemAccessStop**.

A prerequisite for run-time access with the "D:" and "P:" **access classes** is that they are combined with the **access class prefix "E"**.
An "ED:" or "EP:" access can make a run-time access according to the setting of this command.
A "D:" or "P:" access (without "E" prefix") however will always *deny* run-time access, independent of the setting of this command.

Although the CPU is not halted, run-time memory access creates an additional load on the CPU's internal data bus.

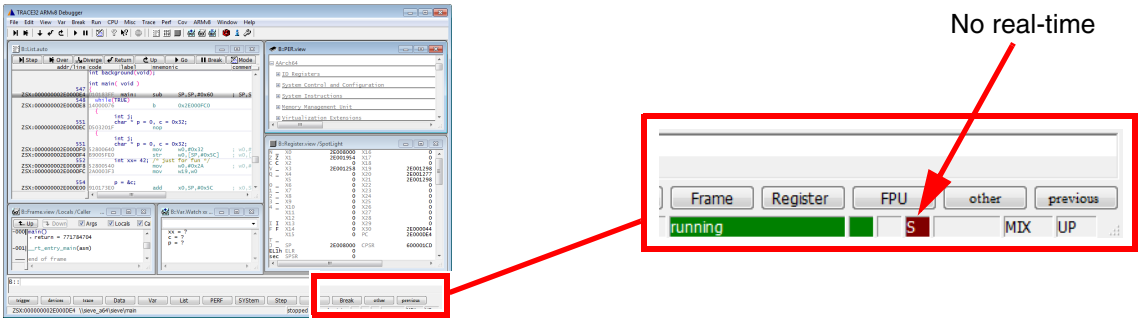
If **SYStem.MemAccess** is not **Denied**, it is possible to read from memory, to write to memory and to set software breakpoints while the CPU is running. For more information, see **SYStem.CpuBreak** and **SYStem.CpuSpot**.

AHB, AXI, SB, ... Depending on which memory buses are available on the chip, the run-time memory access is done through the specified bus.

Denied No memory access is possible while the CPU is running.

StopAndGo Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.
For more information, see below.

If **SYStem.MemAccess StopAndGo** is set, it is possible to read from memory, to write to memory and to set software breakpoints while the CPU is executing the program. To make this possible, the program execution is shortly stopped by the debugger. Each stop takes some time depending on the speed of the JTAG port and the operations that should be performed. A white S against a red background in the TRACE32 state line warns you that the program is no longer running in real-time:



To update specific windows that display memory or variables while the program is running, select the memory class **E:** or the format option **%E**.

```
Data.dump E:0x100

Var.View %E first
```

Format:	**SYStem.MemAccessStop** *<method>*
<method>:	**AUTO** **AAM** **PROGBUF** **SB**

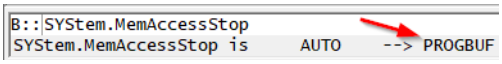
Default: AUTO.

This command defines the memory access method with the **"D:"** and **"P:"** **access classes** while the CPU is **stopped**.

NOTE:	This command only takes effect while the CPU is *stopped*. For memory access while the CPU is running, see **SYStem.MemAccess**.

AUTO	Automatically choose the most suitable memory access method among the methods that are supported by the target.
AAM	Use the 'access memory' abstract command of the RISC-V Debug Module.
PROGBUF	Use program buffer execution via the RISC-V Debug Module.
SB	Use the 'system bus access' block of the RISC-V Debug Module.

If the **AUTO** method is configured, then the method that got automatically selected by the debugger can be seen as soon as the debugger has performed at least one successful memory access. To see the selected method, type "*SYStem.MemAccessStop* " (with whitespace at the end) into the TRACE32 command line. The method should appear in the status line below the command line:



Format: **SYStem.Mode** *<mode>*

SYStem.Attach (alias for SYStem.Mode Attach)
SYStem.Down (alias for SYStem.Mode Down)
SYStem.Up (alias for SYStem.Mode Up)

<mode>:
Down
Prepare
Go
Attach
StandBy
Up

Down (default)
Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tristated.

Prepare
Initializes a debug connection.
The debugger does initialize the debug IP, but it does *not* perform any interaction with the CPU.

This debug mode is used if the CPU shall not be debugged or if it shall be bypassed. The debugger can still access the memory, e.g. via direct system bus access. However, any operation that could alter the CPU state or would require CPU interaction (such as starting or stopping the CPU, accessing memory via the CPU, or accessing GPR/CSR registers) is not possible in this debug mode.

Go
Initializes a debug connection, resets the target (see **SYStem.Option.ResetMode**) and lets the CPU run from its reset vector.

Attach
Initializes a debug connection. The debugger does *not* reset the CPU and does *not* interact with the CPU in any intrusive way. Consequently the CPU stays running if it was running, or stays stopped if it was stopped.

StandBy

Keeps the target in reset via the JTAG reset line and waits until power is detected on the JTAG port. For a reset, the SRST reset line has to be connected to the debug connector.

Once power has been detected, the debugger initializes the debug connection, briefly halts the CPU at the reset vector, restores as many debug registers as possible (e.g. on-chip breakpoints, trace control) and resumes the CPU from the reset vector to start the program execution.

When a CPU power-down is detected, the debugger switches automatically back to the **StandBy** mode. This allows debugging of a power cycle because debug registers will be restored on power-up.

NOTE: Since this mode requires SRST to be asserted right from the beginning of the connect sequence, this mode is only applicable in combination with **SYStem.Option.ResetMode SRST2.**

NOTE: This method only works under very specific circumstances. A more recommended and less error-prone method to handle a core power-down is described in chapter "Hart State: Unavailable".

Up

Initializes a debug connection, resets the target (see **SYStem.Option.ResetMode**) and stops the CPU at its reset vector.

The **SYStem.Option** commands are used to control special features of the debugger or to configure the target. It is recommended to execute the **SYStem.Option** commands **before** the emulation is activated by a **SYStem.Up** or **SYStem.Mode** command.

SYStem.Option.Address32 Define address format display

Format: **SYStem.Option.Address32** [**ON** | **OFF** | **AUTO** | **NARROW**]

Default: AUTO.

Selects the number of displayed address digits in various windows, e.g. **List.auto** or **Data.dump**.

ON	Display all addresses as 32-bit values. 64-bit addresses are truncated.
OFF	Display all addresses as 64-bit values.
AUTO	Number of displayed digits depends on address size.
NARROW	32-bit display with extendible address field.

SYStem.Option.AHBHPROT Select AHB-AP HPROT bits

Format: **SYStem.Option.AHBHPROT** *<value>*

Default: 0

Selects the value used for the HPROT bits in the Control Status Word (CSW) of an AHB Access Port of a DAP, when using the AHB: memory class.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

Format: **SYStem.Option.AXIACEEnable** [**ON** | **OFF**]

Default: OFF.

Enables ACE transactions on the DAP AXI-AP, including barriers. This does only work if the debug logic of the target CPU implements coherent AXI accesses. Otherwise this option will be without effect.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

SYStem.Option.AXICACHEFLAGS Configure AXI-AP cache bits

Format: **SYStem.Option.AXICACHEFLAGS** *<value>*

<value>:
DeviceSYStem
NonCacheableSYStem
ReadAllocateNonShareable
ReadAllocateInnerShareable
ReadAllocateOuterShareable
WriteAllocateNonShareable
WriteAllocateInnerShareable
WriteAllocateOuterShareable
ReadWriteAllocateNonShareable
ReadWriteAllocateInnerShareable
ReadWriteAllocateOuterShareable

Default: DeviceSYStem (=0x30: Domain=0x3, Cache=0x0)

This option configures the value used for the Cache and Domain bits in the Control Status Word (CSW[27:24]->Cache, CSW[14:13]->Domain) of an AXI Access Port of a DAP, when using the AXI: memory class.

The below offered selection options are all non-bufferable. Alternatively you can enter a <value>, where value[5:4] determines the Domain bits and value[3:0] the Cache bits.

DeviceSYStem	=0x30: Domain=0x3, Cache=0x0
NonCacheableSYStem	=0x32: Domain=0x3, Cache=0x2
ReadAllocateNonShareable	=0x06: Domain=0x0, Cache=0x6
ReadAllocateInnerShareable	=0x16: Domain=0x1, Cache=0x6
ReadAllocateOuterShareable	=0x26: Domain=0x2, Cache=0x6

WriteAllocateNonShareable	=0x0A: Domain=0x0, Cache=0xA
WriteAllocateInnerShareable	=0x1A: Domain=0x1, Cache=0xA
WriteAllocateOuterShareable	=0x2A: Domain=0x2, Cache=0xA
ReadWriteAllocateNonShareable	=0x0E: Domain=0x0, Cache=0xE
ReadWriteAllocateInnerShareable	=0x1E: Domain=0x1, Cache=0xE
ReadWriteAllocateOuterShareable	=0x2E: Domain=0x2, Cache=0xE

This option is only meaningful if the chip contains an Arm CoreSight DAP.

SYStem.Option.AXIHPROT Select AXI-AP HPROT bits

Format: **SYStem.Option.AXIHPROT** *<value>*

Default: 0

This option selects the value used for the HPROT bits in the Control Status Word (CSW) of an AXI Access Port of a DAP, when using the AXI: memory class.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

Format: **SYStem.Option.DAPDBGPWRUPREQ** [**ON** | **AlwaysON** | **OFF**]

Default: ON.

This option controls the DBGPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port (DAP) before and after the debug session. Debug power will always be requested by the debugger on a debug session start because debug power is mandatory for debugger operation.

ON Debug power is requested by the debugger on a debug session start, and the control bit is set to 1.
The debug power is released at the end of the debug session, and the control bit is set to 0.

AlwaysON Debug power is requested by the debugger on a debug session start, and the control bit is set to 1.
The debug power is **not** released at the end of the debug session, and the control bit is set to 0.

OFF Only for test purposes: Debug power is **not** requested and **not** checked by the debugger. The control bit is set to 0.

Use case:

Imagine an AMP session consisting of at least of two TRACE32 PowerView GUIs, where one GUI is the master and all other GUIs are slaves. If the master GUI is closed first, it releases the debug power. As a result, a debug port fail error may be displayed in the remaining slave GUIs because they cannot access the debug interface anymore.

To keep the debug interface active, it is recommended that **SYStem.Option.DAPDBGPWRUPREQ** is set to **AlwaysON**.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

SYStem.Option.DAPNOIRCHECK No DAP instruction register check

Format: **SYStem.Option.DAPNOIRCHECK** [**ON** | **OFF**]

Default: OFF.

Bug fix for derivatives which do not return the correct pattern on a DAP (Arm CoreSight Debug Access Port) instruction register (IR) scan. When activated, the returned pattern will not be checked by the debugger.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

SYStem.Option.DAPREMAP Rearrange DAP memory map

Format:	**SYStem.Option.DAPREMAP** {*<address_range> <address>*}

The Debug Access Port (DAP) can be used for memory access during runtime. If the mapping on the DAP is different than the processor view, then this re-mapping command can be used

NOTE:	Up to 16 *<address_range>*/*<address>* pairs are possible. Each pair has to contain an address range followed by a single address.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

SYStem.Option.DAPSYSPWRUPREQ Force system power in DAP

Format:	**SYStem.Option.DAPSYSPWRUPREQ** [**AlwaysON** | **ON** | **OFF**]

Default: ON.

This option controls the SYSPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port (DAP) during and after the debug session

AlwaysON	System power is requested by the debugger on a debug session start, and the control bit is set to 1. The system power is **not** released at the end of the debug session, and the control bit remains at 1.
ON	System power is requested by the debugger on a debug session start, and the control bit is set to 1. The system power is released at the end of the debug session, and the control bit is set to 0.
OFF	System power is **not** requested by the debugger on a debug session start, and the control bit is set to 0.

This option is only meaningful if the chip contains an Arm CoreSight DAP.

Format:	**SYStem.Option.DEBUGPORTOptions** *<option>*
<option>:	**SWITCHTOSWD.**[**TryAll** | **None** | **JtagToSwd** | **LuminaryJtagToSwd** | **DormantToSwd** | **JtagToDormantToSwd**] **SWDTRSTKEEP.**[**DEFault** | **LOW** | **HIGH**]

Default: SWITCHTOSWD.TryAll, SWDTRSTKEEP.DEFault.

See Arm CoreSight manuals to understand the used terms and abbreviations and what is going on here.

SWITCHTOSWD tells the debugger what to do in order to switch the debug port to serial wire mode:

TryAll	Try all switching methods in the order they are listed below. This is the default. Normally it does not hurt to try improper switching sequences. Therefore this succeeds in most cases.
None	There is no switching sequence required. The SW-DP is ready after power-up. The debug port of this device can only be used as SW-DP.
JtagToSwd	Switching procedure as it is required on SWJ-DP without a dormant state. The device is in JTAG mode after power-up.
LuminaryJtagToSwd	Switching procedure as it is required on devices from LuminaryMicro. The device is in JTAG mode after power-up.
DormantToSwd	Switching procedure which is required if the device starts up in dormant state. The device has a dormant state but does not support JTAG.
JtagToDormantToSwd	Switching procedure as it is required on SWJ-DP with a dormant state. The device is in JTAG mode after power-up.

SWDTRSTKEEP tells the debugger what to do with the nTRST signal on the debug connector during serial wire operation. This signal is not required for the serial wire mode but might have effect on some target boards, so that it needs to have a certain signal level.

DEFault	Use nTRST the same way as in JTAG mode which is typically a low-pulse on debugger start-up followed by keeping it high.
LOW	Keep nTRST low during serial wire operation.
HIGH	Keep nTRST high during serial wire operation

This option is only meaningful if the chip contains an Arm CoreSight DAP.

Format:	**SYStem.Option.DMACTiveRESet** [**ON** | **OFF**]

Default: ON

If ON, the debugger will reset the RISC-V debug module via its *dmcontrol.dmactive* bit, before using it for the first time. This is usually done while connecting to the target.

If OFF, the debugger will not reset the RISC-V debug module via its *dmcontrol.dmactive* bit, before using it for the first time. Instead, it will only set the bit to high (if it is not high already).

SYStem.Option.EnReset Allow the debugger to drive nRESET (nSRST)

[SYStem.state window> EnReset]

Format:	**SYStem.Option.EnReset** *<sub_cmd>* (removed)
<sub_cmd>:	**ON** (removed) **Use SYStem.Option.ResetMode SRST instead** **OFF** (removed) **Use SYStem.Option.ResetMode NDMRST instead**

NOTE:	Since release R.2021.02 this command is no longer available for the RISC-V debugger. Please refer to its replacement, **SYStem.Option.ResetMode**.

Default: ON.

If this option is OFF the debugger will never drive the nRESET (nSRST) line on the JTAG connector. This is necessary if nRESET (nSRST) is no open collector or tristate signal.
Instead, during a **SYStem.Up**, the debugger will only assert a soft system reset via the “non-debug module reset” bit (*ndmreset*) of the *dmcontrol* register.

Format: **SYStem.Option.HARVARD** [**ON** | **OFF**]

Default: OFF.

This option must be disabled if the RISC-V target does *not* use a Harvard memory model, i.e. if the target does *not* have physically separate storage and signal pathways for program and data memory.

This option must be enabled if the RISC-V target does use a Harvard memory model.

SYStem.Option.HoldReset Set reset duration time

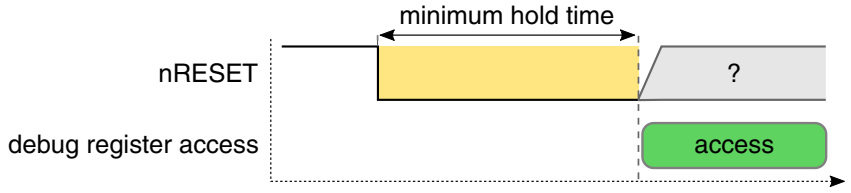
Format: **SYStem.Option.HoldReset** *<time>*

<time>: **1us... 10s**

Default: 50ms.

Set the minimum time the debugger holds the reset active, before either deasserting the reset or continuing with other operations such as debug register accesses (whichever occurs first).

This affects the sequences of **SYStem.Up** and **SYStem.Mode.Go**.



In case of **SYStem.Mode.StandBy,** this command affects the wait time starting after detection of the target power-on (during which SRST is already asserted).

Format: **SYStem.Option.IMASKASM** [**ON** | **OFF**]

Default: OFF.

If enabled, the ‘Step Interrupt Enable Bit’ (dcsr.stepie) will be cleared during assembler single-step operations (**Step.Asm**). No interrupt routines will be executed during assembler single-step operations.

If disabled, the ‘Step Interrupt Enable Bit’ (dcsr.stepie) will be set during assembler single-step operations (**Step.Asm**). Interrupt routines will be executed during assembler single-step operations.

NOTE: Some RISC-V hardware implementations may have hardwired the dcsr.stepie bit to zero.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

Format: **SYStem.Option.IMASKASM** [**ON** | **OFF**]

Default: OFF.

If enabled, the interrupt enable bits of the CPU (mstatus.MIE/SIE/UIE) will be cleared during HLL single-step operations (**Step.Hll** or **Step.Over**). No interrupt routines will be executed during HLL single-step operations. After the HLL single-step, the interrupt enable bits are restored to their original values before the step.

If disabled, the debugger does not modify the interrupt enable bits of the CPU during HLL single-step operations.

SYStem.Option.KeepAlive Keep hart available for debugger

[build 145331 - DVD 09/2022]

Format: **SYStem.Option.KeepAlive** [**ON** | **OFF**]

Default: ON.

Sets or clears the **KEEPALIVE** bit of the RISC-V hart.

The ‘keepalive’ bit suggests that the hardware should attempt to keep the hardware thread (hart) available for the debugger, e.g. by keeping it from entering a low-power state once powered on. Even if the bit is implemented by the hardware, the hardware might not be able to keep a hart available.

SYStem.Option.MMUSPACES Separate address spaces by space IDs

Format: **SYStem.Option.MMUSPACES** [**ON** | **OFF**]
SYStem.Option.MMUspaces [**ON** | **OFF**] (deprecated)
SYStem.Option.MMU [**ON** | **OFF**] (deprecated)

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **“TRACE32 Concepts”** (trace32_concepts.pdf).

NOTE: **SYStem.Option.MMUSPACES** should not be set to **ON** if only one translation table is used on the target.

If a debug session requires space IDs, you must observe the following sequence of steps:

1. Activate **SYStem.Option.MMUSPACES**.

2. Load the symbols with **Data.LOAD**.

Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

```
;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x012A:
Data.dump D:0x012A:0xC00208A

;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x0203:
Data.dump D:0x0203:0xC00208A
```

SYStem.Option.ResetDetection Choose method to detect a target reset

Format: **SYStem.Option.ResetDetection** *<method>*

<method>: **nSRST** | **None**

Default: nSRST

Selects the method how an external target reset can be detected by the debugger.

nSRST Detects a reset if nSRST (nRESET) line on the debug connector is pulled low.

None Detection of external resets is disabled.

SYStem.Option.ResetMode Select reset method

Format: **SYStem.Option.ResetMode** *<method>*

<method>: **SRST**
SRST2
NDMRST
HartRST

Default: SRST.

Configures the reset method used by **SYStem.Up** and **SYStem.Mode Go**.

SRST System reset via the SRST signal of the JTAG connector. This signal is sometimes also called nSRST, RST or RESET.
SRST is asserted directly before the halt request. See paragraph **ResetMode SRST** for details.

SRST2 System reset via the SRST signal of the JTAG connector. This signal is sometimes also called nSRST, RST or RESET.
SRST is asserted before the first debug register access. See paragraph **ResetMode SRST2** for details.

NDMRST System reset via the ‘ndmreset’ bit of the ‘dmcontrol’ debug register in the RISC-V Debug Module.
See paragraph **ResetMode NDMRST** for details.

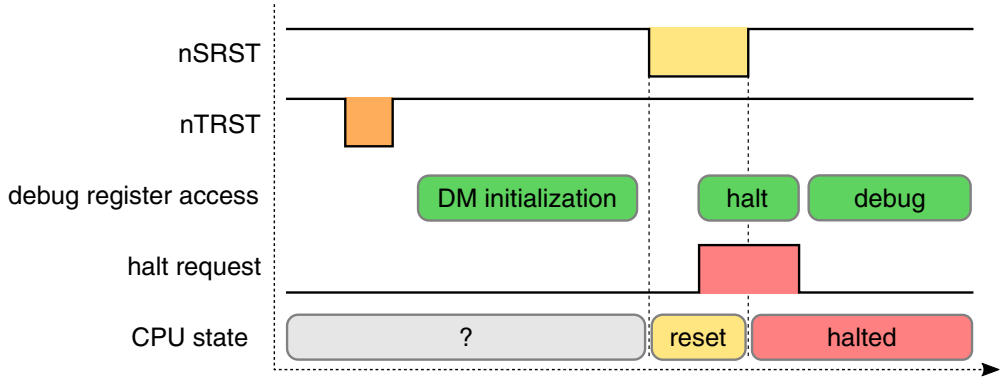
HartRST Hardware thread (hart) reset via the ‘hartreset’ bit of the ‘dmcontrol’ debug register in the RISC-V Debug Module.
Resets all harts that are currently selected via **CORE.ASSIGN**.
See paragraph **ResetMode HartRST** for details.

The behavior of the respective reset method can be further influenced by the following configuration options:

- **SYStem.Option.HoldReset**
- **SYStem.Option.WaitReset**

ResetMode SRST

The sequence of SYStem.Up with SYStem.Option.ResetMode SRST looks as follows:

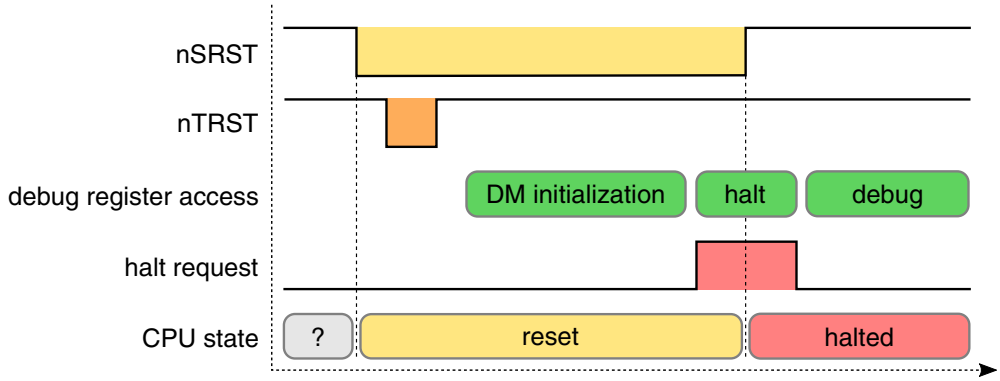


The above debug register access sequence labeled with ‘halt’ does only contain accesses to the ‘dmcontrol’ debug register.

The test-logic-reset via nTRST can be configured by SYStem.Option.TRST.

ResetMode SRST2

The sequence of SYStem.Up with SYStem.Option.ResetMode SRST2 looks as follows:

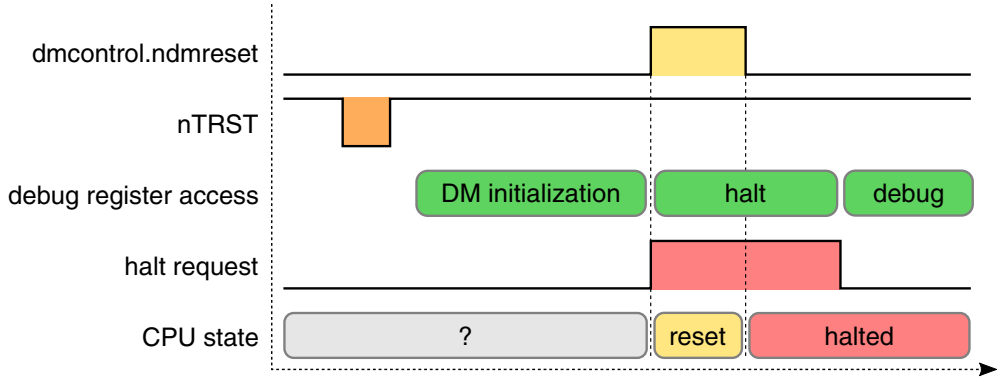


The above debug register access sequence labeled with 'DM initialization' (Debug Module initialization) can contain accesses to any arbitrary debug register. That is why this sequence should only be used if the target allows debugger access to all debug registers while SRST is asserted.

The test-logic-reset via nTRST can be configured by SYStem.Option.TRST.

ResetMode NDMRST

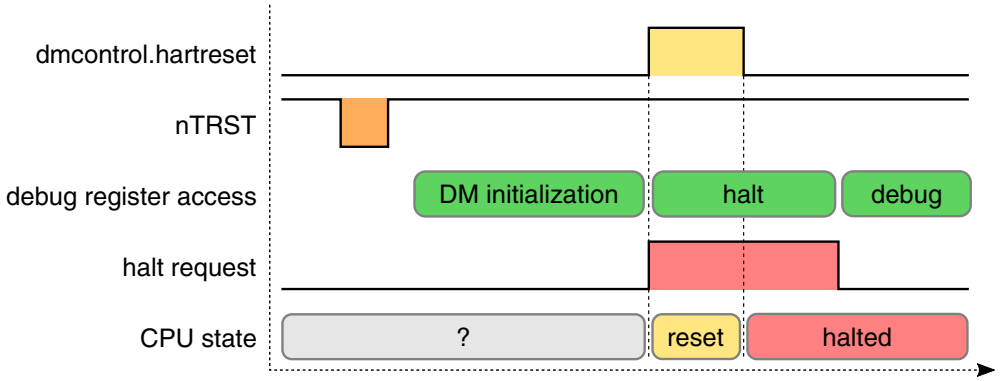
The sequence of SYStem.Up with SYStem.Option.ResetMode NDMRST looks as follows:



The above debug register access sequence labeled with 'halt' does only contain accesses to the 'dmcontrol' debug register.

The test-logic-reset via nTRST can be configured by SYStem.Option.TRST.

The sequence of SYStem.Up with SYStem.Option.ResetMode HartRST looks as follows:



The above debug register access sequence labeled with 'halt' does only contain accesses to the 'dmcontrol' debug register.

The test-logic-reset via nTRST can be configured by SYStem.Option.TRST.

SYStem.Option.SOFTLONG Use 32-bit access to set SW breakpoints

Format:	**SYStem.Option.SOFTLONG** [**ON** | **OFF**]

Default: OFF.

Instructs the debugger to only use 32-bit accesses to patch the code of software breakpoints.

NOTE:	If the debugger should be restricted to only use 32-bit accesses for any kind of memory access (not only for software breakpoints), then please see the command **MAP.BUS32** instead.

Format: **SYStem.Option.SYSDownACTion** *<action>*

<action>: **NONE**
DCSRRST

Default: NONE.

Defines the action that shall be taken when a **SYStem.Down** is performed.

The respective action, however, will *not* be executed if the debugger performs an automated **SYStem.Down** after an error situation.

NONE No action.

DCSRRST Reset certain bits of the *Debug Control and Status Register* (DCSR) of the core under debug to their respective default values. This does not affect the *dcsr.prv* bits or bitfields with implementation-specific reset values (“preset reset values”).
This action can be intrusive, as it may be necessary to temporarily halt the core in order to access the register.

SYStem.Option.TRST Allow debugger to drive TRST

[SYStem.state window > TRST]

Format: **SYStem.Option.TRST** [**ON** | **OFF**]

Default: ON.

If this option is disabled, the nTRST line is never driven by the debugger (permanent high).

If this option is enabled, the debugger *may* make a test-logic-reset via nTRST during the connect sequence to the target.

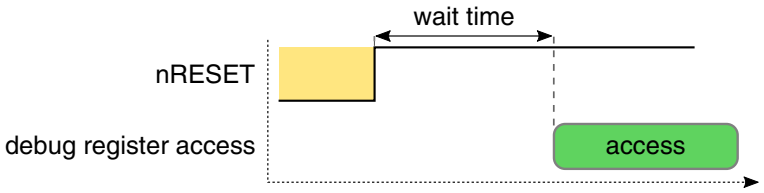
Independent of whether this option is ON or OFF, the debugger’s connect sequence may contain other mechanisms for test-logic-reset or test-logic initialization (for example in case of JTAG debugging, the sequence may additionally contain five consecutive TMS pulses to reset the JTAG TAP).

Format:	**SYStem.Option.WaitReset** *<time>*
<time>:	**1us... 10s**

Default: 50ms.

Set the time that the debugger will wait after deassertion of a reset, e.g. during **SYStem.Up** or **SYStem.Mode.Go**.

Before the wait time is over, the debugger will *not* perform any other target interactions such as JTAG shifts or debug register accesses.



[Example]

Format: **SYStem.Option.ZoneSPACES** [**ON** | **OFF**]

Default: OFF.

The **SYStem.Option.ZoneSPACES** command must be set to **ON** if separate symbol sets are used for the following RISC-V modes:

- Machine mode (access classes M:, MD:, and MP:)
- Supervisor mode (S:, SD:, and SP:) and
- User mode (access classes U:, UD,: and UP:)

RISC-V has two CPU mode dependent address spaces. Within TRACE32, these two CPU mode dependent address spaces are referred to as zones:

- In Machine mode, no address translation is performed. TRACE32 treats the Machine mode as one zone.
- In Supervisor mode as well as in User mode, addresses are translated by the hardware MMU. Both modes share the same address space because they use the same translation. Thus, TRACE32 treats both Supervisor mode and User mode as one single zone.

Due to the different address translation in these modes, different code and data can be visible on the same logical address.

NOTE: For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **"TRACE32 Concepts"** (trace32_concepts.pdf).

OFF	TRACE32 does not separate symbols by access class. Loading two or more symbol sets with overlapping address ranges will result in unpredictable behavior. Loaded symbols are independent of the CPU mode.
ON	Separate symbol sets can be loaded for each zone, even with overlapping address ranges. Loaded symbols are specific to one of the CPU zones.

SYStem.Option.ZoneSPACES is set to **ON** if the user wants to debug code which is executed in Supervisor or User mode, such as a operating system, and code which is executed in Machine mode, such as exception handlers.

If **SYStem.Option.ZoneSPACES** is **ON**, TRACE32 enforces any memory address specified in a TRACE32 command to have an access class which clearly indicates to which zone the memory address belongs.

If an address specified in a command uses an anonymous access class such as D:, P: or C:, the access class of the current PC context is used to complete the addresses' access class.

If a symbol is referenced by name, the associated access class of its zone will be used automatically, so that the memory access is done within the correct CPU mode context. As a result, the symbol's logical address will be translated to the physical address with the correct MMU translation table.

Example:

```
SYStem.Option.ZoneSPACES ON

; 1. Load a Linux image to Supervisor mode
; (access classes S:, SP: and SD: are used for the symbols of Linux.
;  access classes U:, UP: and UD: are used for User mode applications):
Data.LOAD.ELF vmlinux S:0x0 /NoCODE

; 2. Load a secure driver image to Machine mode:
; (access classes M:, MP: and MD: are used for the symbols):
Data.LOAD.ELF secdriver M:0x0 /NoCODE
```

SYStem.state — Display SYStem.state window

Format: **SYStem.state**

Displays the **SYStem.state** window for system settings that configure debugger and target behavior.

FPU.Set — Write to FPU register

Format:	**FPU.Set** *<register>*[**.***<precision>*] [*<expression>* | *<float>*]
<register>:	**F0** | **F1** … **F31**
<precision>:	**auto** **Single** **Double**

Writes to a floating-point register of the RISC-V core under debug.

auto	Automatic detection of the floating-point precision. The debugger automatically detects whether the current value of *<register>* is single-precision or double-precision, and uses the detected precision for the register write. • If single-precision is detected, **FPU.Set** *<register>***.auto** is equal to **FPU.Set** *<register>***.Single**. • If double-precision is detected, **FPU.Set** *<register>***.auto** is equal to **FPU.Set <***register***>.Double**.
Single	Uses single-precision floating-point representation for the register write.
Double	Uses double-precision floating-point representation for the register write.
<float>	**Parameter Type**: Float.
<expression>	**Parameter Type**: Decimal or hex.

Example:

```
FPU.Set F4.auto   1.4     ; Write to register with
                          ; automatic detection of precision
FPU.Set F4.Single 2.7     ; Write to register with single-precision
FPU.Set F4.Double 3.2     ; Write to register with double-precision

FPU.Set F6.Single 0xABCD  ; Write to register with single-precision
                          ; in hexadecimal notation
FPU.Set F6.Double 12.     ; Write to register with double-precision
                          ; in decimal notation
```

MMU.DUMP Page wise display of MMU translation table

Format:	**MMU.DUMP** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.dump** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** *<cpu_specific_tables>*

Displays the contents of the CPU specific MMU translation table.

- If called without parameters, the complete table will be displayed.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Displays the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: displays the translation table of the specified process • else, this command displays the table the CPU currently uses for MMU translation.

KernelPageTable	Displays the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and displays its table entries.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Displays the MMU translation table entries of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

CPU specific Tables in MMU.DUMP <table>

none.

Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.List** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**

Lists the address translation of the CPU-specific MMU table.

- If called without address or range parameters, the complete table will be displayed.
- If called without a table specifier, this command shows the debugger-internal translation table. See **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, table entries will only be displayed if their **logical** address matches with the given parameter.

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Lists the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: list the translation table of the specified process • else, this command lists the table the CPU currently uses for MMU translation.
KernelPageTable	Lists the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and lists its address translation.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Lists the MMU translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and lists its address translation. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** [**Clear**] *<cpu_specific_tables>*

Loads the CPU-specific MMU translation table from the CPU to the debugger-internal static translation table.

- If called without parameters, the complete page table will be loaded. The list of static address translations can be viewed with **TRANSlation.List**.
- If the command is called with either an address range or an explicit address, page table entries will only be loaded if their **logical** address matches with the given parameter.

Use this command to make the translation information available for the debugger even when the program execution is running and the debugger has no access to the page tables and TLBs. This is required for the real-time memory access. Use the command **TRANSlation.ON** to enable the debugger-internal MMU table.

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID: loads the translation table of the specified process • else, this command loads the table the CPU currently uses for MMU translation.

KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.
ALL [**Clear**]	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**. **Clear:** This option allows to clear the static translations list before reading it from all page translation tables.

<range>	The address range of the page table which will be scanned for valid entries.
<address>	The start address from which the page table will be scanned for valid entries. The end address for the scan is *<address>* + *<scan_range>* *<scan_range>* is explained below.

If neither *<range>* nor *<address>* are specified, the page table will be scanned from 0 to *<scan_range>*

<scan_range> depends on the selected or auto-detected **MMU format**.

- MMU format **SV32**: *<scan_range>* = 2^{32} - 1
- MMU format **SV39**: *<scan_range>* = 2^{39} - 1
- MMU format **SV48**: *<scan_range>* = 2^{48} - 1

CPU specific TrOnchip Commands

The **TrOnchip** command group is not available for the RISC-V debugger.

Connector Type and Pinout

RISC-V Debug Cable with 20 pin Connector

Adaption for RISC-V Debug Cable: See **www.lauterbach.com/adriscv.html**

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	N/C
TRST-	3	4	GND
TDI	5	6	GND
TMS	7	8	GND
TCK	9	10	GND
RTCK	11	12	GND
TDO	13	14	GND
RESET-	15	16	GND
N/C	17	18	GND
N/C	19	20	GND

Pin 2, pin 17 and pin 19 must under no circumstances be connected on the target side. Otherwise the hardware of the debugger can get damaged.