





[TRACE32 Online Help](#)

[TRACE32 Directory](#)

[TRACE32 Index](#)

TRACE32 Documents	
ICD In-Circuit Debugger	
Processor Architecture Manuals	
RISC-V	
RISC-V Debugger	1
History	4
Introduction	5
Brief Overview of Documents for New Users	5
Demo and Start-up Script	6
List of Abbreviations and Definitions	7
Warning	8
Quick Start of the JTAG Debugger	9
Quick Start for Multicore Debugging	11
SMP Debugging	11
SMP Debugging - Selective	12
Homogeneous SMP/AMP Debugging	13
Heterogeneous SMP/AMP Debugging	14
Troubleshooting	15
SYStem.Up Errors	15
FAQ	16
RISC-V Specific Implementations	18
Debug Specification for External Debug Support	18
Floating-Point Extensions	18
Breakpoints	20
Software Breakpoints	20
On-chip Breakpoint Resources	20
On-chip Breakpoints for Instruction Address	20
On-chip Breakpoints for Data Address	20
On-chip Data Value Breakpoints	21
Examples for Standard Breakpoints	22
Access Classes	23

CPU specific SETUP Command	24
SETUP.DIS	Disassembler configuration 24
CPU specific SYStem Commands	26
SYStem.CONFIG.state	Display target configuration 26
SYStem.CONFIG	Configure debugger according to target topology 27
<parameters> describing the “DebugPort”	28
<parameters> describing the “JTAG” scan chain and signal behavior	30
<parameters> describing debug and trace “Components”	33
SYStem.CONFIG.CPUMemAccess	Set CPU memory access method 34
SYStem.CONFIG.HARTINDEX	Set hardware thread index 35
SYStem.CPU	Select the used CPU 35
SYStem.CpuAccess	Run-time memory access (intrusive) 36
SYStem.CpuBreak	Master control to deny stopping the target (long stop) 37
SYStem.CpuSpot	Master control to deny spotting the target (short stop) 38
SYStem.JtagClock	Define JTAG frequency 39
SYStem.LOCK	Tristate the JTAG port 39
SYStem.MemAccess	Run-time memory access (non-intrusive) 40
SYStem.Mode	Establish the communication with the target 42
SYStem.Option	Special setup 43
SYStem.Option Address32	Define address format display 43
SYStem.Option EnReset	Allow the debugger to drive nRESET (nSRST) 43
SYStem.Option HARVARD	Use Harvard memory model 44
SYStem.Option IMASKASM	Disable interrupts while single stepping 44
SYStem.Option MMUSPACES	Separate address spaces by space IDs 44
SYStem.Option ResetDetection	Choose method to detect a target reset 45
SYStem.Option SYSDownACTion	Define action during SYStem.Down 46
SYStem.Option TRST	Allow debugger to drive TRST 46
SYStem.Option ZoneSPACES	Enable symbol management for zones 47
SYStem.state	Display SYStem.state window 48
CPU specific FPU Command	49
FPU.Set	Write to FPU register 49
CPU specific MMU Commands	50
MMU.DUMP	Page wise display of MMU translation table 50
MMU.List	Compact display of MMU translation table 52
MMU.SCAN	Load MMU table from CPU 53
Target Adaption	55
Connector Type and Pinout	55
RISC-V Debug Cable with 20 pin Connector	55
Support	56
Available Tools	56
Compilers	57

Products	58
Product Information	58
Order Information	59

History

- 10-May-19 New command: [SYStem.Option ResetDetection](#).
- 12-Apr-19 New command: [SYStem.Option.ZoneSPACES](#).
- 05-Apr-19 New commands: [MMU.DUMP](#), [MMU.List](#), and [MMU.SCAN](#).
- 08-Jan-19 Added description of the command [SYStem.Option HARVARD](#).
- 07-Dec-18 Added description of the command [SYStem.Option.SYSDownACTion](#).
- 29-Nov-18 Added description and examples for the commands [SETUP.DIS RegNames](#) and [SETUP.DIS AbiNames](#).
- 29-Jun-18 Added description and example for the command [FPU.Set](#).
- 22-Jun-18 New chapter “[Quick Start for Multicore Debugging](#)”
- 11-Apr-18 Added description of the command [SYStem.Option MMUSPACES](#).

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:

- **“Debugger Basics - Training”** (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.

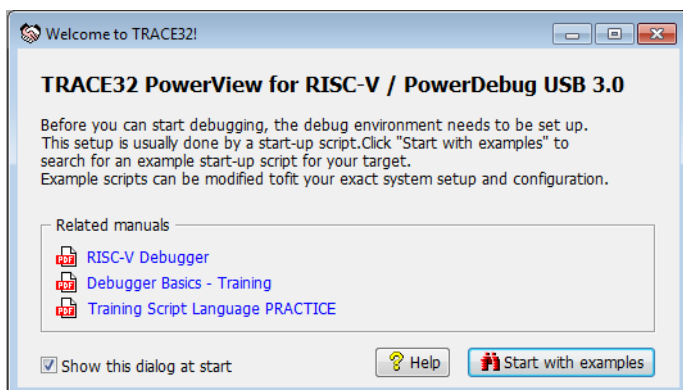
Architecture-specific information:

- **“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)

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



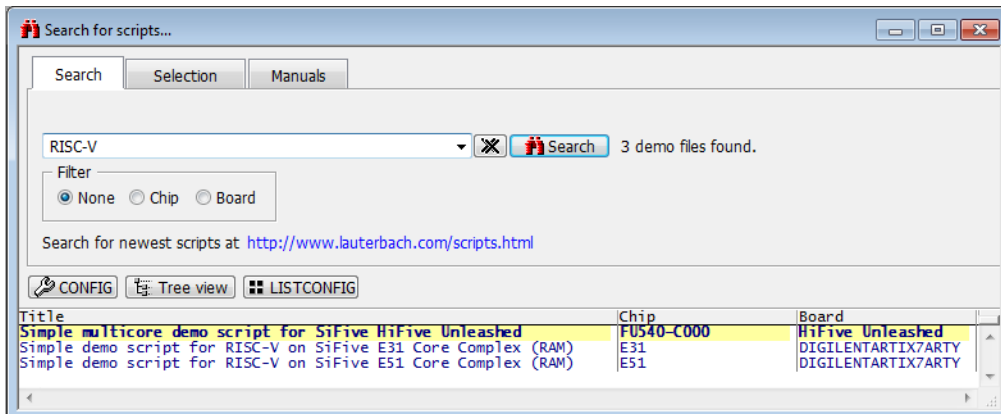
Demo and Start-up Script

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 inspect the demo folder manually in the system directory of TRACE32.

The ~/~/demo/riscv/ folder contains:

hardware/	Ready-to-run debugging and flash programming demos for evaluation boards. Recommended for getting started!
kernel/	Various OS Awareness examples.

List of Abbreviations and Definitions

CSR	Control and Status Register
DM	Debug Module, as defined by the standard RISC-V debug specification
DTM	Debug Transport Module, as defined by the standard 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.

Quick Start of the JTAG Debugger

Starting up the debugger is done as follows:

1. Reset the debugger.

```
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.

2. 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.

3. 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.

4. 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**".

A simple start sequence example is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** <file>.

```
WinCLEAR                ; Clear all windows

SYStem.CPU FU540-C000    ; Select the core type

MAP.BOnchip 0x10000++0xffff ; Specify where FLASH/ROM is

SYStem.Up                ; Reset the target and 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 0x63FFFFFF ; Set the stack pointer to address
                        ; 0x63FFFFFF

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 flags ast ; Open watch window for variables *)

Break.Set 0x1000 /Program  ; Set software breakpoint to address
                        ; 0x1000(address 0x1000 outside of
                        ; BOnchip range)

Break.Set 0x10100 /Program ; Set on-chip breakpoint
                        ; to address 0x10100 (address
                        ; 0x10100 is within BOnchip range)
```

*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

Quick Start for Multicore Debugging

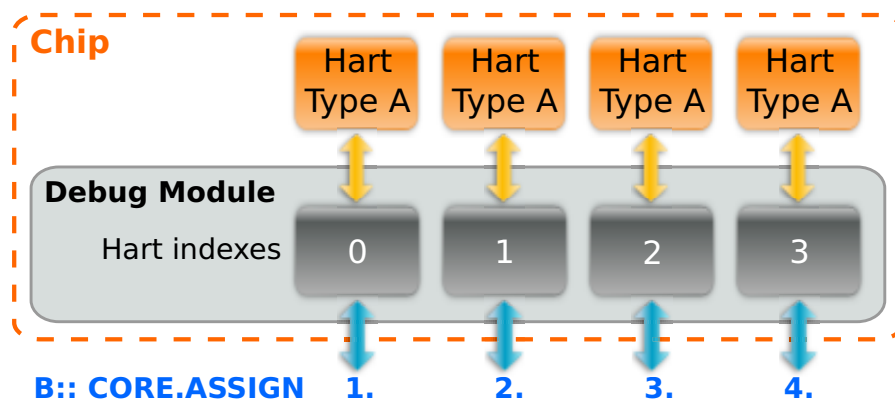
This chapter provides a quick start for multicore processing. The following example scenarios cover the most common use cases:

- **Example A: SMP Debugging** (Symmetric Multiprocessing)
- **Example B: SMP Debugging** (Symmetric Multiprocessing) - 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 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.



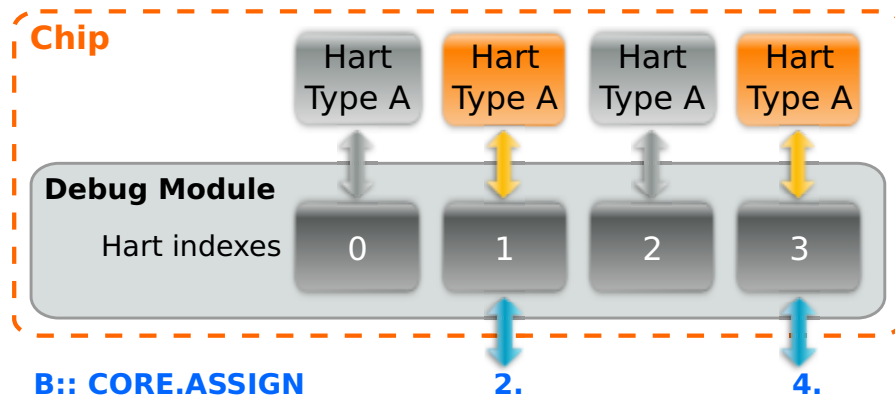
Example A:

```
SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 1. 1.
SYStem.CONFIG.CoreNumber 4. ; 4 harts of type A in total
SYStem.CONFIG.HARTINDEX 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 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.



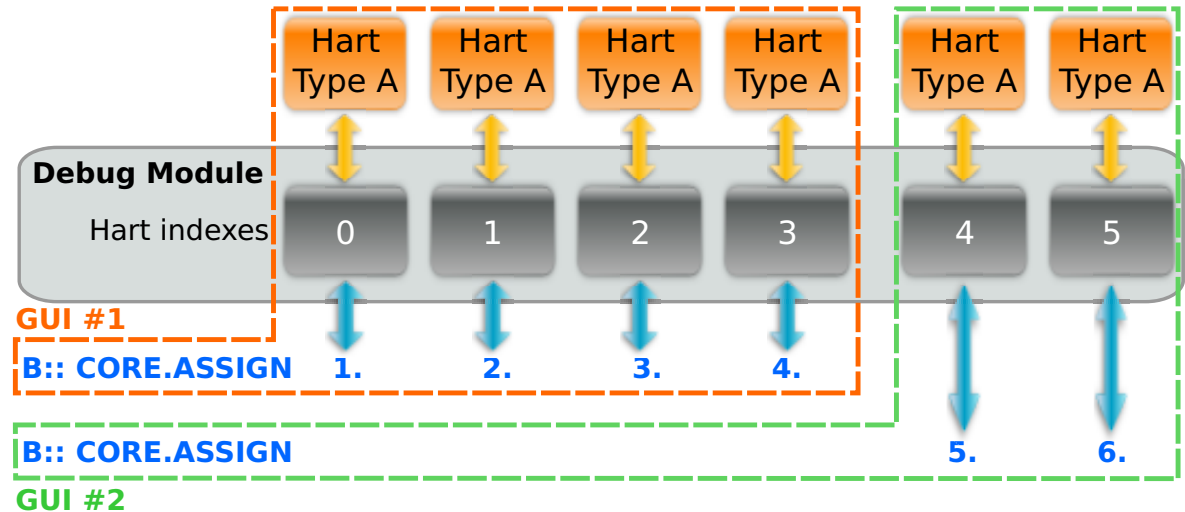
Example B:

```
SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 1. 1.
SYStem.CONFIG.CoreNumber 4. ; 4 harts of type A in total
SYStem.CONFIG.HARTINDEX 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 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:

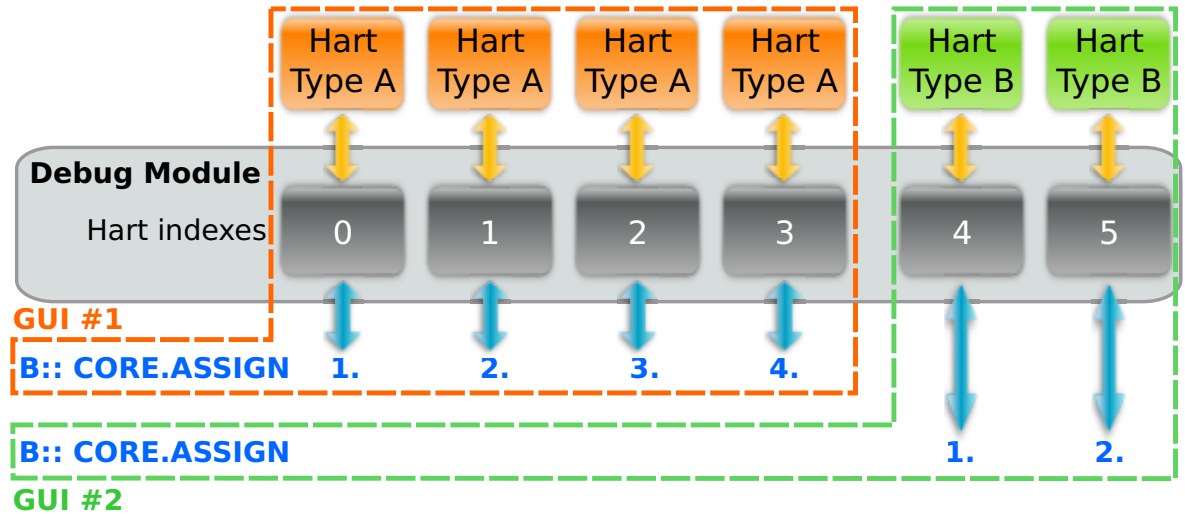
```
; ---- TRACE32 PowerView GUI #1 -----  
  
SYStem.CPU <type_a_cpu>  
SYStem.CONFIG.CORE 1. 1.  
SYStem.CONFIG.CoreNumber 6. ; 6 harts of type A in total  
SYStem.CONFIG.HARTINDEX 0. 1. 2. 3. 4. 5.  
CORE.ASSIGN 1. 2. 3. 4. ; Assign the first 4 harts
```

```
; ---- TRACE32 PowerView GUI #2 -----  
  
SYStem.CPU <type_a_cpu>  
SYStem.CONFIG.CORE 2. 1.  
SYStem.CONFIG.CoreNumber 6. ; 6 harts of type A in total  
SYStem.CONFIG.HARTINDEX 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 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.  
SYStem.CONFIG.CoreNumber 4. ; 4 harts of type A in total  
SYStem.CONFIG.HARTINDEX 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.  
SYStem.CONFIG.CoreNumber 2. ; 2 harts of type B in total  
SYStem.CONFIG.HARTINDEX 4. 5. ; Hart indexes of type B  
CORE.ASSIGN 1. 2. ; Assign all 2 harts of type B
```

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command, this may have the following reasons:

- The target has no power.
- The target is in reset.
- The core is not enabled.
- There is logic added to the JTAG state machine.
- There are additional loads or capacities or serial resistors on the JTAG lines.
- There is a short circuit on at least one of the output lines of the core.
- There are stubs on the signal line.

<p>Debugging via VPN</p> <p>Ref: 0307</p>	<p>The debugger is accessed via Internet/VPN and the performance is very slow. What can be done to improve debug performance?</p> <p>The main cause for bad debug performance via Internet or VPN are low data throughput and high latency. The ways to improve performance by the debugger are limited:</p> <p>In PRACTICE scripts, use "SCREEN.OFF" at the beginning of the script and "SCREEN.ON" at the end. "SCREEN.OFF" will turn off screen updates. Please note that if your program stops (e.g. on error) without executing "SCREEN.OFF", some windows will not be updated.</p> <p>"SYStem.POLLING SLOW" will set a lower frequency for target state checks (e.g. power, reset, jtag state). It will take longer for the debugger to recognize that the core stopped on a breakpoint.</p> <p>"SETUP.URATE 1.s" will set the default update frequency of Data.List/Data.dump/Variable windows to 1 second (the slowest possible setting).</p> <p>prevent unneeded memory accesses using "MAP.UPDATEONCE [address-range]" for RAM and "MAP.CONST [address--range]" for ROM/FLASH. Address ranged with "MAP.UPDATEONCE" will read the specified address range only once after the core stopped at a breakpoint or manual break. "MAP.CONST" will read the specified address range only once per SYStem.Mode command (e.g. SYStem.Up).</p>
---	--

<p>Setting a Software Breakpoint fails</p> <p>Ref: 0276</p>	<p>What can be the reasons why setting a software breakpoint fails?</p> <p>Setting a software breakpoint can fail when the target HW is not able to implement the wanted breakpoint.</p> <p>Possible reasons:</p> <p>The wanted breakpoint needs special features that are only possible to realize by the trigger unit inside the controller.</p> <p>Example: Read, write and access (Read/Write) breakpoints ("type" in Break.Set window). Breakpoints with checking in real-time for data-values ("Data"). Breakpoints with special features ("action") like TriggerTrace, TraceEnable, TraceOn/TraceOFF.</p> <p>TRACE32 can not change the memory.</p> <p>Example: ROM and Flash when no preparation with FLASH.Create, FLASH.TARGET and FLASH.AUTO was made. All type of memory if the memory device is missing the necessary control signals like WriteEnable or settings of registers and SpecialFunctionRegisters (SFR).</p> <p>Contrary settings in TRACE32.</p> <p>Like: MAP.BOnchip for this memory range. Break.SELect.<breakpoint-type> Onchip (HARD is only available for ICE and FIRE).</p> <p>RTOS and MMU:</p> <p>If the memory can be changed by Data.Set but the breakpoint doesn't work it might be a problem of using an MMU on target when setting the breakpoint to a symbolic address that is different than the writable and intended memory location.</p>
---	--

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/specifications/debug-specification/>

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).

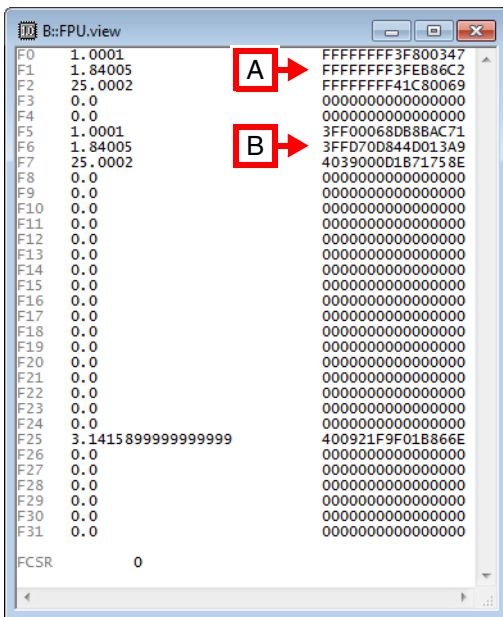
The floating-point features are provided by the **FPU** (Floating-Point Unit) command group.

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

Breakpoints

For general information about setting breakpoints, refer to the [Break.Set](#) command.

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is temporarily patched by a breakpoint code (RISC-V *EBREAK* instruction). There is no restriction in the number of software breakpoints.

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 $X = A \text{ XOR } B$ (infix operator XOR: “exclusive or”).
Let $Y = A \text{ AND } X$ (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 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.

Examples for Standard Breakpoints

Assume you have a target with

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

The command to set up TRACE32 correctly for this configuration is:

```
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

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
```

Access Classes

For background information about the term *access class*, see “[TRACE32 Glossary](#)” (glossary.pdf).

The following RISC-V specific access classes are available.

Access Class	Description
P	Program memory access
D	Data memory access
SB	System bus access The read and write accesses with this class are usually performed via the “System Bus Access block” of the RISC-V Debug Module.
E	Access memory while CPU is running. See SYStem.MemAccess , SYStem.CpuBreak and SYStem.CpuSpot . Any memory access class can be prefixed with E , if the memory access class supports access while the CPU is running.
CSR	Control and Status Register (CSR) access The CSR address of this access class does always address data of maximum CSR register width <i>XLEN</i> . If a CSR register is smaller than the maximum size, the unused segment gets filled up with zero.
M	Machine privilege level
S	Supervisor privilege level For memory read and write accesses with this class, machine privilege level will be used.
U	User privilege level For memory read and write accesses with this class, machine privilege level will be used.

To perform an access with a certain access class, write the class in front of the address.

Example:

```
Data.dump D:0x0--0x3
Data.dump ED:0x0--0x3      ; run-time data memory access
Data.dump SD:0x0--0x3
PRINT Data.Long(CSR:0x300)
```

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](#).

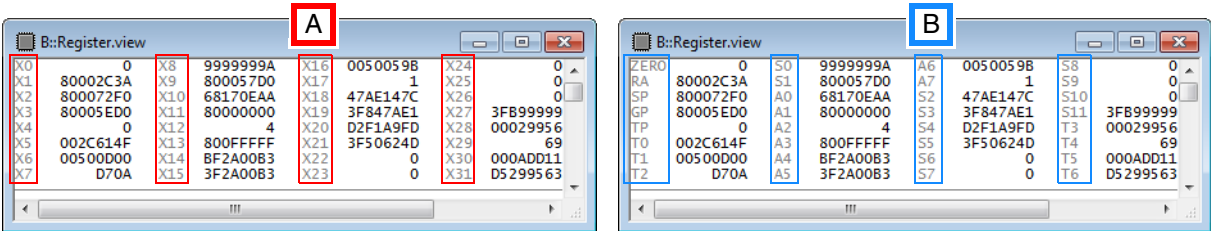
The command does **not affect existing windows** containing disassembler output.

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

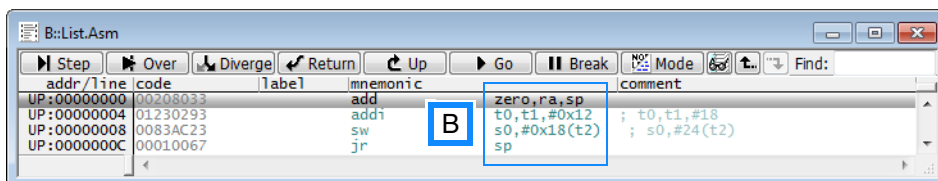
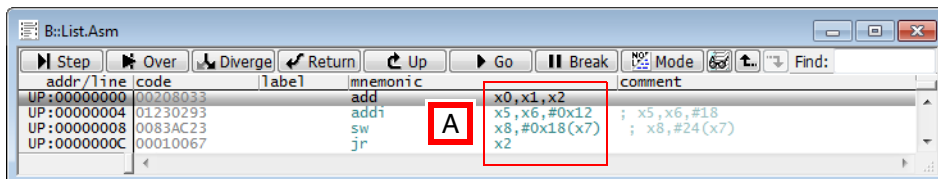
Example 2: The changed naming scheme does **not** affect an existing **List.Asm** window. You need to open another **List.Asm** window to view the changed naming scheme.

```
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**.

Format: **SYStem.CONFIG.state** [/<tab>]

<tab>: **DebugPort** | **Jtag** | **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 .
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 .

Format:	SYStem.CONFIG <i><parameter></i>
<i><parameter></i> : (DebugPort)	CORE <i><core></i> <i><chip></i> CoreNumber <i><number></i> DEBUGPORT [DebugCable0] DEBUGPORTTYPE [JTAG CJTAG SWD] Slave [ON OFF] TriState [ON OFF]
<i><parameter></i> : (JTAG)	DRPOST <i><bits></i> DRPRE <i><bits></i> IRPOST <i><bits></i> IRPRE <i><bits></i> Slave [ON OFF] TAPState <i><state></i> TCKLevel <i><level></i> TriState [ON OFF]
<i><parameter></i> : (Components)	COREDEBUG.Base <i><address></i> COREDEBUG.RESET

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”

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 uTrace 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>

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.

(cont.)

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 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).

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 TAP position can be defined with the first four commands in the table below.

DRPOST <bits> Defines the TAP position 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 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 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 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.

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.

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.

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]

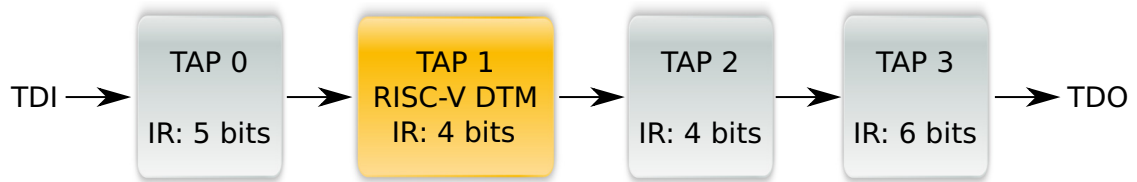
TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

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.

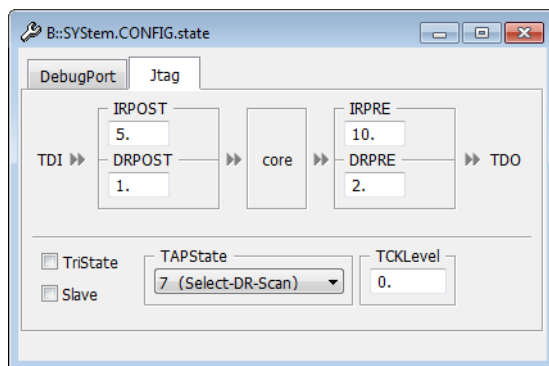
Default: OFF.

Example:



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

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



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

COREDEBUG.Base <address>

COREDEBUG.RESET

RISC-V Debug Module: Debug Registers base address

The RISC-V Debug Module (DM) can either be accessible via a *Debug Transport Module* (DTM, e.g. “JTAG-DTM”), or alternatively by mapping the debug registers of the DM on a system bus.

If the DM debug registers are mapped on a system bus (for example on the memory access port of an Arm CoreSight system), then this command configures the base address of the DM register address space.

Format: **SYStem.CONFIG.CPUMemAccess** *<method>*

<method>: **AUTO | AAM | PROGBUF | SBA**

Default: AUTO.

Configures the method that the debugger uses for memory access from perspective of the CPU.

AUTO	The debugger automatically tries to select the most suitable method among the methods that are supported by the target.
AAM	Memory access via “abstract command ‘access memory’” (aam) of the RISC-V Debug Module.
PROGBUF	Memory access via program buffer execution of the RISC-V Debug Module.
SBA	Memory access via “system bus access” method of the RISC-V Debug Module. Depending on the target system, this memory view can differ from the memory view of the CPU.

Format:	SYStem.CONFIG.HARTINDEX <index>
<index>:	0. 1. ... n

Default: 0.

Configures the hardware thread (hart) index that is used by the Debug Module to address one or more harts.

The command requires an index for each hart that is covered by [SYStem.CONFIG.CoreNumber](#).

Example:

```
SYStem.CONFIG.CoreNumber 5.  
SYStem.CONFIG.HARTINDEX 3. 4. 5. 6. 7.
```

For further examples, see [“Quick Start for Multicore Debugging”](#), page 11.

Format:	SYStem.CPU <cpu>
<cpu>:	RV32 RV64 ...

RV32 and **RV64** are *default* entries for the 32-bit and 64-bit RISC-V cores respectively. These entries should only be selected if the respective debug IP of the target does conform to the official RISC-V debug specification, and if no dedicated <cpu> entry for the respective target CPU or board is available.

<cpu>	For a list of supported CPUs, use the command <code>SYStem.CPU *</code> or refer to the chip search on the Lauterbach website.
-------	--

Format: **SYStem.CpuAccess** *<sub_cmd>* (deprecated)

<sub_cmd>: **Enable** (deprecated)
Use SYStem.MemAccess StopAndGo instead.

Denied (deprecated)
There is no need to use a successor command (default setting).

Nonstop (deprecated)
Use SYStem.CpuBreak Denied instead.

Default: Denied.

Configures how memory access is handled during run-time.

Enable	Allow intrusive run-time memory access.
Denied	Lock intrusive run-time memory access.
Nonstop	Lock all features of the debugger that affect the run-time behavior.

Format:	SYSystem.CpuBreak [<i><mode></i>]
<i><mode></i> :	Enable Denied

Default: Enable.

Enable	Allows stopping the target.
Denied	<p>Denies stopping the target. This includes manual stops and stop breakpoints. However, short stops, such as spot breakpoints, may still be allowed.</p> <p>SYSystem.CpuBreak Denied can be used to protect a target system which does not tolerate that the program execution is stopped for an extended period of time; for example, a motor controller which could damage the motor if the motor control software is stopped.</p> <p>For more information, see SYSystem.CpuSpot, SYSystem.MemAccess.</p>

Example:

```
SYSystem.CpuBreak Denied

Break.Set main           ; stop breakpoint results in an error message

Break.Set main /Spot     ; spot breakpoint may be allowed
```

Format: **SYStem.CpuSpot** [*<mode>*]

<mode>: **Enable | Denied | Target | SINGLE**

Default: Enable.

Spotting is an intrusive way to transfer data periodically or on certain events from the target system to the debugger. As a result, the program is not running in real-time anymore. For more information, see [SYStem.CpuBreak](#) and [SYStem.MemAccess](#).

Enable	Allows spotting the target.
Denied	Denies spotting the target. Stopping the target may still be allowed.
Target	Allows spotting the target controlled by the target. This allows target-stopped FDX and TERM communication. All other spots are denied.
SINGLE	Allows single spots triggered by a command. This includes spotting for changing the breakpoint configuration and the SNOOPPer.PC command. This setting also allows target-stopped FDX and TERM communication. All other spots are denied.

Example:

```
SYStem.CpuSpot Denied
Break.Set main /Spot ; spot breakpoint results in an error message
```

Format: **SYStem.JtagClock** [*<frequency>*]

<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.

SYStem.LOCK

Tristate the JTAG port

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 <mode>
<mode>:	Denied SB AHB AXI ... (Arm CoreSight) StopAndGo

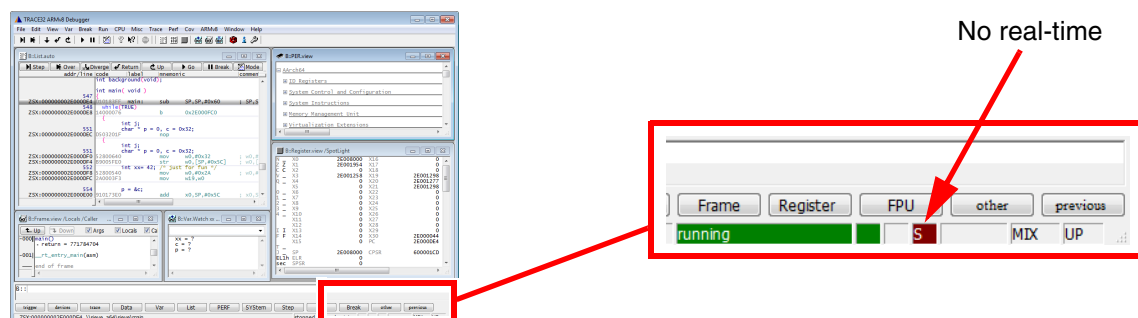
Default: Denied.

This option declares if a **non-intrusive** memory access can take place while the CPU is excuting code. 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 executing code. For more information, see [SYStem.CpuBreak](#) and [SYStem.CpuSpot](#).

Denied	No memory access is possible while the CPU is executing code.
SB	Run-time memory access is done via the “system bus access” method of the RISC-V Debug Module.
AHB, AXI, ...	Run-time memory access is done via the memory access port of an Arm CoreSight DAP. This requires that the respective memory access port is available on the chip and was properly configured (see access classes for details).
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.Mode** <mode>

<mode>:
Down
Prepare
Go
Attach
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 it shall be bypassed. In that case 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 accessing GPR or CSR registers) is not possible in this debug mode.

Go

Initializes a debug connection, resets the target (assertion of system reset) and lets the CPU run from its reset vector.

Attach

Initializes a debug connection. The target is *not* reset (*no* system reset), i.e. the state of the target is *not* changed. Consequently the CPU stays running if it was running, or stays stopped if it was stopped.

Up

Initializes a debug connection, resets the target (assertion of system reset), and stops the CPU at its reset vector.

The **SYStem.Option** commands are used to control special features of the debugger or emulator 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]

Default: Depending on the CPU selected with the **SYStem.CPU** command.

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.

SYStem.Option EnReset

Allow the debugger to drive nRESET (nSRST)

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

Default: ON.

If this option is disabled 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.

If this option is enabled the debugger will perform both reset options (assert nRESET line and set ndmreset bit) during a **SYStem.Up**.

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 IMASKASM

Disable interrupts while single stepping

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

Default: ON.

If enabled, the Step Interrupt Enable Bit will be cleared during assembler single-step operations. The interrupt routine is not executed during single-step operations.

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.

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.

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). Instead five consecutive TCK pulses with TMS high are asserted to reset the TAP controller which have the same effect.

Format:	SYSystem.Option ZoneSPACES [ON OFF]
---------	--

Default: OFF.

The **SYSystem.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 Glossary ” (glossary.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.

SYSystem.Option ZoneSPACES ON

SYSystem.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 **SYSystem.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. <ul style="list-style-type: none">• 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.
PageTable	Displays the current MMU translation table entries of the CPU. This command reads all tables the CPU currently uses for MMU translation and displays the table entries.
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. <ul style="list-style-type: none">• 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.

none.

Format:	MMU.List <i><table></i> [<i><range></i> <i><address></i> <i><range></i> <i><root></i> <i><address></i> <i><root></i>] MMU.<i><table></i>.List (deprecated)
<i><table></i> :	PageTable KernelPageTable TaskPageTable <i><task_magic></i> <i><task_id></i> <i><task_name></i> <i><space_id></i> :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.

<i><root></i>	The <i><root></i> 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.
PageTable	Lists the current MMU translation of the CPU. This command reads all tables the CPU currently uses for MMU translation and lists the address 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 <i><task_magic></i> <i><task_id></i> <i><task_name></i> <i><space_id></i> :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. <ul style="list-style-type: none"> • 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
<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 current MMU address translation of the CPU. This command reads all tables the CPU currently uses for MMU translation, and copies the address translation into the debugger-internal static translation table.
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. <ul style="list-style-type: none"> • 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	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 .

<i><range></i>	The address range of the page table which will be scanned for valid entries.
<i><address></i>	The start address from which the page table will be scanned for valid entries. The end address for the scan is <i><address></i> + <i><scan_range></i> <i><scan_range></i> 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>* = $2^{48} - 1$

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.

Lauterbach technical support is available via www.lauterbach.com/tsupport.html.

For support requests that concern RISC-V, contact bdmriscv-support@lauterbach.com

Available Tools

CPU	ICE	FIRE	ICD DEBUG	ICD MONITOR	ICD TRACE	POWER INTEGRATOR	INSTRUCTION SIMULATOR
BM-310			YES				
E20			YES				
E21			YES				
E24			YES				
E31			YES				
E34			YES				
E76			YES				
FU540-C000			YES				
LINXCORE130			YES				
LINXCORE130S			YES				
LINXCORE131			YES				
N25			YES				
NX25			YES				
S51			YES				
S54			YES				
S76			YES				
SCR1			YES				
SCR3			YES				
U54			YES				
U74			YES				

Compilers

Language	Compiler	Company	Option	Comment
C++	GCC	GNU Compiler Collection	ELF/DWARF	

Product Information

OrderNo Code	Text
LA-2717 JTAG-RISC-V	Debugger for RISC-V (ICD) supports RISC-V MIPI10 / MIPI20 connector on the target required: LA-3770 ARM Converter ARM-20 to MIPI-10/20/34 ALTERA-10/RISCV-10 connector on the target required: LA-3863 Converter ARM-20 to ALTERA-10/RISCV-10 includes software for Windows, Linux and MacOSX requires Power Debug Module
LA-2717A JTAG-RISC-V-A	JTAG Debugger for RISC-V Add. supports RISC-V only suitable for debug cables newer than 07/2008 MIPI10 / MIPI20 connector on the target required: LA-3770 ARM Converter ARM-20 to MIPI-10/20/34 ALTERA-10/RISCV-10 connector on the target required: LA-3863 Converter ARM-20 to ALTERA-10/RISCV-10 includes software for Windows, Linux and MacOSX
LA-2765 CON-ARM20-RISC-V-01	Converter ARM-20 to RISC-V SiFive-12 PIN PMOD Converter from ARM-20 to 12 pin PMOD connectors with the JTAG pinout from SiFive. PMOD is an open standard defined by Digilent for peripherals used with FPGAs or microcontrollers. The PMOD pinout of this adapter does ONLY work with targets that use the pinout that was defined by SiFive. So it is not a general/official RISC-V pinout. In case of FPGA boards, the pinout depends on the used FPGA image.
LA-3770 CONV-ARM20/MIPI34	Converter ARM-20 to MIPI-10/20/34 Arm/RISC-V Converter to connect a 20-pin debug cable to 10/20/34 pin connectors specified by MIPI
LA-3863 CON-ARM-ALTERA	Converter ARM-20 to ALTERA-10/RISCV-10 Converter from ARM 20-pin connector to Altera 10-pin (Byteblaster) connector or RISC-V 10-pin Target connector pitch can be 2.54mm or 1.27mm (half size).

Order No.	Code	Text
LA-2717	JTAG-RISC-V	Debugger for RISC-V (ICD)
LA-2717A	JTAG-RISC-V-A	JTAG Debugger for RISC-V Add.
LA-2765	CON-ARM20-RISC-V-01	Converter ARM-20 to RISC-V SiFive-12 PIN PMOD
LA-3770	CONV-ARM20/MIPI34	Converter ARM-20 to MIPI-10/20/34 Arm/RISC-V
LA-3863	CON-ARM-ALTERA	Converter ARM-20 to ALTERA-10/RISCV-10
Additional Options		
LA-3774A	JTAG-TEAKLITE-III-A	JTAG Debugger for TeakLite III Add. (ICD)
LA-7960X	MULTICORE-LICENSE	License for Multicore Debugging