MicroBlaze Debugger and Trace

Release 02.2026

MicroBlaze Debugger and Trace

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ....

ICD In-Circuit Debugger ....

Processor Architecture Manuals ....

MicroBlaze ....

MicroBlaze Debugger and Trace 1

General Note 5

Introduction 5

Brief Overview of Documents for New Users 5

Demo and Start-up Scripts 6

MicroBlaze Debug and Trace Features Supported by TRACE32 7

ESD Protection 8

Quick Start of the Debugger 9

Quick-Start of the Real-Time Trace 12

Compiling Software with Debug Information 13

Troubleshooting 14

SYStem.Up Errors 14

FAQ 14

Displaying MicroBlaze Core Configuration 15

CPU specific Implementations 16

Memory Accesses Causing Bus Errors 16

Breakpoints 17

Software Breakpoints 17

On-chip Breakpoints 17

Breakpoints in ROM 17

Example for Breakpoints 18

SYStem.Option.AHBHPROT Select AHB-AP HPROT bits 19

SYStem.Option.AXIACEEnable ACE enable flag of the AXI-AP 19

SYStem.Option.AXICACHEFLAGS Configure AXI-AP cache bits 19

SYStem.Option.AXIHPROT Select AXI-AP HPROT bits 20

SYStem.Option.BrkHandler Control writing of software break handler 20

SYStem.Option.BrkVector Configures an alternative breakvector 21

SYStem.Option.DAPDBGPWRUPREQ Force debug power in DAP 22
SYStem.Option.DAPNOIRCHECK No DAP instruction register check 22
SYStem.Option.DAPREMAP Rearrange DAP memory map 23
SYStem.Option.DAPSYSPWRUPREQ Force system power in DAP 23
SYStem.Option.DEBUGPORTOptions Options for debug port handling 24
SYStem.Option.IMASKASM Interrupt disable on ASM 25
SYStem.Option.IMASKHLL Interrupt disable on HLL 25
SYStem.Option.LittleEndian Select little endian mode 25
SYStem.Option.ResetMode Select the reset mode 25
SYStem.Option.DUALPORT Use real-time access by default 26
SYStem.Option.MDMSINGLELMB Use MDM LMB master 0 for all cores 26
TERM.METHOD.MDMUART Terminal configuration 26
Memory Classes 28
Register Names 28

CPU specific SYStem Commands 29
SYStem.CPU Select the used CPU 29
SYStem.JtagClock Selects the frequency for the debug interface 30
SYStem.LOCK Lock and tristate the debug port 30
SYStem.MemAccess Select run-time memory access method 31
SYStem.Mode Select operation mode 32
SYStem.CONFIG Configure debugger according to target topology 33
Daisy-Chain Example 37
TapStates 38
<parameters> configuring a CoreSight Debug Access Port “AP” 39
SYStem.CONFIG.CORE Assign core to TRACE32 instance 45
SYStem.CONFIG.state Display target configuration 46
SYStem.CONFIG.MDM.Base Select MDM base address 46
SYStem.CONFIG.MDM.DebugPort Set core to debug 46
SYStem.CONFIG.MDM.RESet Reset MDM configuration 47
SYStem.CONFIG.MDM.view Display MDM configuration 47
SYStem.CONFIG.MDM.UserInst Set default user BSCAN port 47

TrOnchip Commands 48
TrOnchip.state Display on-chip trigger window 48
TrOnchip.RESet Set on-chip trigger to default state 48
TrOnchip.CONVert Adjust range breakpoint in on-chip resource 48
TrOnchip.VarCONVert Adjust complex breakpoint in on-chip resource 49

CPU specific MMU Commands 50
MMU.DUMP Page wise display of MMU translation table 50
MMU.List Compact display of MMU translation table 51
MMU.SCAN Load MMU table from CPU 53

Real-Time Trace 55
SYStem.Option.DTM Control data trace messages 55

SYStem.Option.QUICKSTOP Control trace of software breakpoints 55

Configuring your FPGA 56

JTAG Connector 57

Mechanical Description 57

JTAG Connector for Xilinx Microblaze 57

MicroBlaze Debugger and Trace

Version 03-Apr-2026

General Note

Before starting, please ensure that your debugger software is up to date by downloading the latest update from the LAUTERBACH website. Note that the downloads on the website are stable releases but may not necessarily be the latest versions. If you encounter any issues, please submit a support ticket at **support.lauterbach.com/new-ticket** for getting the latest software update.

Introduction

Please note that only the **Processor Architecture Manual** (the document you are currently reading) is specific to the core architecture. All other parts of the online help are general and independent of any core architecture. Therefore, if you have questions related to the core architecture, the **Processor Architecture Manual** should be your primary reference.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Debugger Tutorial"** (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.
- **"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.

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**.

Please note that multicore configuration will be required in most cases, even when there is only a single Microblaze processor in the target. For information about setting up multicore-configuration see the application note **“Connecting to MicroBlaze Targets for Debug and Trace”** (app_microblaze.pdf).

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known MicroBlaze based hardware.

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/microblaze/` subfolder of the system directory of TRACE32.

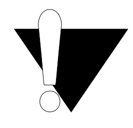
MicroBlaze Debug and Trace Features Supported by TRACE32

TRACE32 for MicroBlaze supports the following features:

- Basic debugging (stop, go, software breakpoints, ...).
- Debugging Linux kernel code and user applications.
- MMU translation.
- Onchip breakpoints (program breakpoints, data write and read breakpoints).
- Off-chip program and data trace are supported via the MicroBlaze Debug Module (MDM).
- Use the Xilinx Vivado Design Suite for hardware analysis and the Lauterbach TRACE32 infrastructure for software debugging - over a single (shared) connection to the target board via Lauterbach hardware. For more details, see **"Integration for Xilinx Vivado"** (int_vivado.pdf).

NOTE:	• As onchip breakpoints require additional FPGA resources and may slow down the maximum frequency of a MicroBlaze design, it is necessary to explicitly configure them in the FPGA design. • Trace via the MicroBlaze Debug Module requires at least Vivado 2018.3 or Vivado 2018.2 with a special patch, see Xilinx application note AR71422. • There is a deprecated MicroBlaze trace IP (Xilinx MicroBlaze Trace Core, XMTC). Support for this IP has ended with Xilinx EDK 12.4. This trace is no longer supported with TRACE32.

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.

	Multicore configuration will be required in most cases, even when there is only a single Microblaze processor in the target. For information about setting up multicore-configuration see the application note **“Connecting to MicroBlaze Targets for Debug and Trace”** (app_microblaze.pdf).

For getting started with debugging, the installation DVD contains sample bit streams and scripts for ML310, ML403, Spartan3EStarter, Spartan3ADSP1800Starter boards. You find them in the TRACE32 demo folder: `~~/demo/microblaze/hardware`

The following example uses ML403. Configure the target with the bit stream `~~/demo/microblaze/hardware/memecfx12lc/stopandgo/download.bit`

The FPGA configuration can be done using Xilinx Vivado (or its predecessor Xilinx iMPACT) or the TRACE32 command **JTAG.PROGRAM** (or the old version of the command, **JTAG.LOADBIT**).

After starting the TRACE32 software enter the following commands for connecting to the target and load a sample file:

1. Select the correct endianness of your core:

```
SYStem.Option.LittleEnd ON
```

2. Configure multicore settings for telling the debugger where the MicroBlaze core is located in the JTAG scan chain. If in doubt, you can use the command **SYStem.DETECT.SHOWChain** to get a list of all JTAG TAPs. The correct TAP to use is the one of the Xilinx FPGA. For example, for the Xilinx EVB ML403 use the following settings:.

```
SYStem.CONFIG.IRPOST 28.                ; Note the ´.´ indicating
SYStem.CONFIG.IRPRE 8.                  ; decimal numbers.
SYStem.CONFIG.DRPOST 2.
SYStem.CONFIG.DRPRE 1.
```

3. If your FPGA design contains multiple MicroBlaze processors, select which one you want to debug:

```
; Setting "Specifies the JTAG user-defined register used" in the
; Vivado MDM configuration dialog or configuration C_JTAG_CHAIN
SYStem.CONFIG.MDM UserInst USER2
; Which MDM debug port your core is connected to, valid range
; 0.--31.
SYStem.CONFIG.MDM DebugPort 0.
```

4. Attach to the target and enter debug mode, using the multicore settings from above:

```
SYStem.Up
```

This command resets the CPU and enters debug mode. After executing this command, memory and registers can be accessed.

5. Load a sample program..

```
CD ~~/demo/microblaze/hardware/ml403/mb.v710d.xmtc.100b.noddrram
Data.LOAD.Elf sieve_00000000.elf /CYGDRIVE
```

Note the option `/CYGDRIVE`. As the Xilinx MicroBlaze compiler is executed within a Cygwin environment it creates debug symbols with paths beginning with `\cygdrive\c\`. By using the option `/CYGDRIVE` TRACE32 internally converts this prefix to the correct syntax e.g. to `c:\` on windows hosts. Refer for more information to **Data.LOAD.Elf**.

6. Open the disassembly and register windows:

```
Data.List                                    ; Open disassembly window

Register                                     ; Open register window
```

7. You are now ready to debug your program.

Quick-Start of the Real-Time Trace

To use real-time trace, you first have to correctly configure the block design in Vivado. Also note that you need at least Vivado 2018.3 or Vivado 2018.2 with a special patch from Xilinx answer record AR71422.

1. In the configuration dialog of the core, select “Advanced” at the top, Navigate to the tab “Debug” and select “External Trace” and set “Trace Buffer Size” to “8kB”.

2. In the configuration dialog of the MicroBlaze Debug Module, set “Select External Trace Output Interface” to “EXTERNAL” and set “External Trace Data Width” to twice the desired trace port width (number of data pins connected from the FPGA to the off-chip trace connector).

3. In the “Block Properties” of the MDM, set “CONFIG.C_TRACE_PROTOCOL” to 1. Alternatively, use the following command in the TCL console, assuming your MDM has the default instance name “mdm_0”:

   ```
   set_property CONFIG.C_TRACE_PROTOCOL 1 [get_bd_cells /mdm_0]
   ```

4. Connect an appropriate clock signal to the TRACE.TRACE_CLOCK port and export the TRACE.TRACE_DATA signal. Also export the that feeds TRACE.TRACE_CLOCK. Leave TRACE.TRACE_CLK_OUT and TRACE.TRACE_CTL unconnected.

5. Use ODDR buffers to create DDR (double data rate) signalling on the external trace port. Refer to **~~/demo/microblaze/etc/hdl/mdm_parallel_trace_adapter.vhd** in your TRACE32 installation directory for an example module you can instantiate in your block design.

After you have programmed your modified design to the FPGA, connect for debugging as usual. To configure and enable trace, use the following commands:

```
Trace.METHOD Analyzer ; Tell TRACE32 that we wish to use off-chip trace.
Trace.PortSize 16.    ; Configure the number of connected data pins
Trace.AutoFocus       ; Execute a test program to detect the best
                      ; electrical parameters for sampling.
```

Compiling Software with Debug Information

For debugging, the target programs need to contain debug information. It is **recommended** to compile MicroBlaze software with the **GCC option `-g3`**. The option `-g` creates debug info that does not work well with TRACE32. Also keep in mind that using code optimization can cause problems with debugging.

NOTE:	It is recommended to compile MicroBlaze software with the GCC option -g3.

Troubleshooting

SYStem.Up Errors

The **SYStem.Up** command is used to establish a debug connection to the target. If you receive error messages while executing this command this may have one reasons listed below.

All	The target has no power.
All	The multicore settings are incorrect. For information how to calculate the multicore settings see **"Connecting to MicroBlaze Targets for Debug and Trace"** (app_microblaze.pdf)
All	The debugger software is out of date. The Microblaze architecture evolves rapidly and therefore regular updates of the debugger software are necessary. Note that the software downloads on the LAUTERBACH website represent stable releases but are not necessarily the latest versions. If the problems persist after updating from the website, please contact LAUTERBACH support.
All	The target FPGA is not configured correctly. The FPGA configuration (e.g. via ACE files) can be disturbed, if the debug cable is attached to the target but the debugger is powered down. Try to detach the debug cable and attach it after FPGA configuration.
All	The target is in reset: The debugger controls the processor reset and use the RESET line to reset the CPU on every SYStem.Up.
All	You used a wrong JTAG connector on the target. In particular on ML310 always use the 14pin JTAG connector J9 for debugging Microblaze.

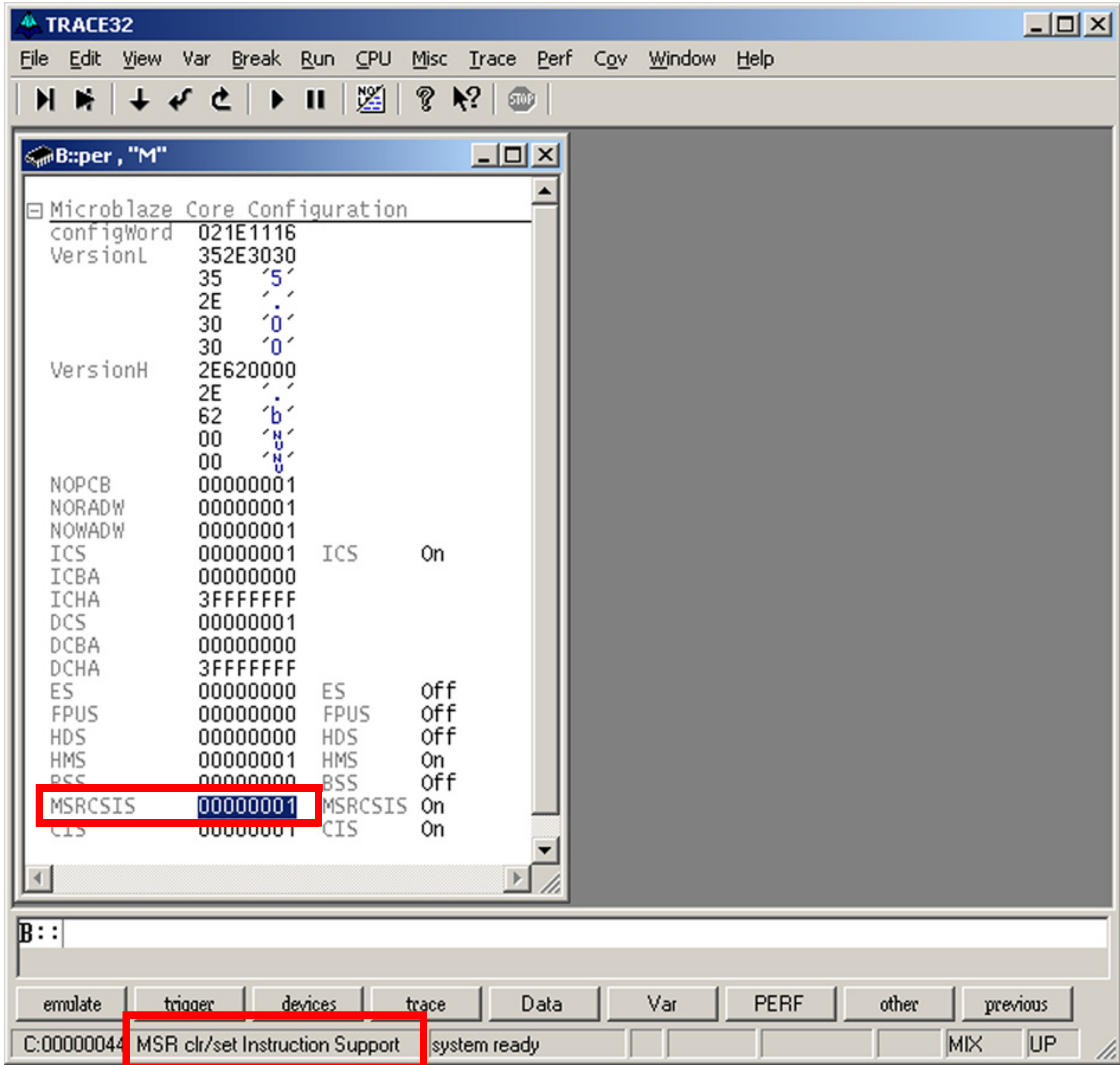
FAQ

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

Displaying MicroBlaze Core Configuration

As the Microblaze core is configurable the available debug features depend on the current core.

The configuration of the core can be displayed using the command `per`. When pointing the mouse at an entry, the debugger displays an explanation in the status line.



CPU specific Implementations

This section gives information about design decision regarding the implementation of some special features.

Memory Accesses Causing Bus Errors

Bus errors can be caused by pointers to invalid address regions or memory that is not mapped by the MMU (e.g. when using an operating system). Normally bus errors are detected by the debugger and displayed as "`?????????`" in memory dump windows.

However, due to a core limitation detecting bus errors while the core is inside an exception handler would alter the system state in a way preventing correct continuation of the program. Therefore inside an exception handler (`MSR.EIP=1`) , the debugger uses a different memory access method that preserves the correct system state but does not detect bus errors. In this case the contents of invalid memory regions will show random data.

Under Linux the most common case for this problem is when a system call branches to the hardware exception vector on 0x20. In this case the core switches to real mode (`MSR.VM=0`) but the stack pointer R1 still points to an address in (now unmapped) virtual memory, until it is adapted a few instructions later. If there is an open register window, the stack area will consequently show random data for a few cycles (instead of indicating a bus error). Once the stack pointer is set up correctly inside the exception handler, the stack area is displayed correctly.

Breakpoints

There are two types of breakpoints available:

- Software breakpoints (SW-BP) and
- Onchip breakpoints.

Software Breakpoints

Software breakpoints are implemented via a breakpoint instruction. These are the default breakpoints and are usually used in RAM areas. Utilizing advanced TRACE32 mechanisms, in software breakpoints can also be used in FLASH areas.There is no restriction in the number of software breakpoints.

For using SW breakpoints with ucLinux or other operating systems, setting the option **SYStem.Option.BrkVector** may be required.

On-chip Breakpoints

Onchip breakpoints (Lauterbach terminology) allow to stop the core in specific conditions. As this is implemented via hardware-resources, they are also referred to as “hardware breakpoints” in non-Lauterbach terminology.

The following list gives an overview of the usage of the on-chip breakpoints by TRACE32-ICD:

- **Instruction breakpoints** stop the core when it reaches a certain program location.
- **Read/Write address breakpoints** can stop the core upon read or write data accesses.
- **Data breakpoints** stop the program when a specific data value is written to an address or when a specific data value is read from an address.

NOTE:	The number of available onchip breakpoints depends on the configuration of the MicroBlaze core defined in the FPGA design.

Breakpoints in ROM

With the command **MAP.BOnchip <address_range>**, TRACE32 is configured to use onchip breakpoints in the specified address range. Therefore the command **Break.Set** will set an onchip breakpoint in this range and the parameter **/Onchip** can be omitted. Typically this feature is used with ROM or FLASH memories that prevent the use of software breakpoints.

Example for Breakpoints

Assume you have a target with FLASH from 0 to 0xFFFFF and RAM from 0x100000 to 0x11FFFF. The command to configure TRACE32 correctly for this configuration is:

```
Map.BOnchip 0x0--0x0FFFFF
```

The following breakpoint combinations are possible.

Software breakpoints:

```
Break.Set 0x100000 /Program                ; Software Breakpoint 1
Break.Set 0x101000 /Program                ; Software Breakpoint 2
Break.Set 0xx /Program                     ; Software Breakpoint 3
```

On-chip breakpoints:

```
Break.Set 0x100 /Program                   ; On-chip Breakpoint 1
Break.Set 0x0ff00 /Program                 ; On-chip Breakpoint 2
```

SYStem.Option.AHBHPROT

Select AHB-AP HPROT bits

Format:	**SYStem.Option.AHBHPROT** *<value>* (deprecated) **Use SYStem.CONFIG.AHBAPn.HPROT instead.**

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.

SYStem.Option.AXIACEEnable

ACE enable flag of the AXI-AP

Format:	**SYStem.Option.AXIACEEnable** [**ON** | **OFF**] (deprecated) **Use SYStem.CONFIG.AXIAPn.ACEEnable instead.**

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.

SYStem.Option.AXICACHEFLAGS

Configure AXI-AP cache bits

Format:	**SYStem.Option.AXICACHEFLAGS** *<value>* (deprecated) **Use SYStem.CONFIG.AXIAPn.CacheFlags instead.**

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.

SYStem.Option.AXIHPROT   Select AXI-AP HPROT bits

Format:	**SYStem.Option.AXIHPROT** *<value>* (deprecated) **Use SYStem.CONFIG.AXIAPn.HPROT instead.**

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.

SYStem.Option.BrkHandler   Control writing of software break handler

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

Default: AUTO

The option controls whether the debugger writes a handler for software breakpoints to the target memory. The address can be configured via **SYStem.Option.BrkVector**.

The option **AUTO** detects if the software breakpoint handler is required by the current core.

This can be overridden by the options **ON** or **OFF** for special cases. The breakpoint handler should be switched **OFF** when

- using Linux because it utilizes a breakpoint handler created by the kernel
- if the vector table resides in ROM or “fetch-only” memory areas. In this case the vector table pre-loaded with the memory image must contain a breakpoint handler.

If all program memory is read-only consider the use of OnChip breaks as alternative.

NOTE:	A software breakpoint handler is required for using software breakpoints on MicroBlaze cores with versions < 7.20.A.

Format:	**SYStem.Option.BrkVector** *<vector>*
<vector>:	**0 … 0xFFFC, 32-bit aligned**

Use this option to set an alternative address for the software breakpoint handler created by the debugger. Changing the default address is necessary when the vector 0x18 is occupied e.g. by interrupt handlers.

The option must be set before attaching to the target to have an effect.

The vector should be 32bit-aligned. Do not use 0x0 as break vector.

For ucLinux it is recommended to set the handler address to 0x70.

When changing the breakpoint vector, the debugger automatically uses a matching opcode for software breakpoints.

NOTE:	For additional information see **SYStem.Option.BrkHandler**.

SYStem.Option.DAPDBGPWRUPREQ   Force debug power in 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.

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.
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.

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**.

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.

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 memory mapping on the DAP is different from the processor memory view, then this re-mapping command can be used. The *<address_range>* parameter describes the processor memory view to re-map. The *<address>* parameter is the re-map target memory offset on the DAP bus.

NOTE:	Up to 16 *<address_range>*/*<address>* pairs are possible. Each pair has to contain an address range followed by a single address. Executing this command will replace **all** 16 previously specified pairs. That means all required pairs have to be specified at once. Any pairs left empty will be cleared.

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.
OFF	System power is **not** requested by the debugger on a debug session start, and the control bit is set to 0.
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.

This option is for target processors having a Debug Access Port (DAP) e.g., Cortex-A or Cortex-R.

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

SYStem.Option.IMASKASM — Interrupt disable on ASM

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

Mask interrupts during assembler single steps. Useful to prevent interrupt disturbance during assembler single stepping.

SYStem.Option.IMASKHLL — Interrupt disable on HLL

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

Mask interrupts during HLL single steps. Useful to prevent interrupt disturbance during HLL single stepping.

SYStem.Option.LittleEndian — Select little endian mode

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

Selects endianness.

SYStem.Option.ResetMode — Select the reset mode

Format:	**SYStem.Option.ResetMode** *<mode>*
<mode>:	**CORE** | **SYSTEM**

Use this option to select the reset mode. CORE will only reset the MicroBlaze core while SYSTEM will also reset the peripherals.

Note that a reset of the MicroBlaze core does not reset the register R1-R31, caches and UTLB.

SYStem.Option.DUALPORT — Use real-time access by default

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

If this option is enabled all memory access use real-time access if possible. This has the same effect as using the “E:” access class modifier.

SYStem.Option.MDMSINGLELMB — Use MDM LMB master 0 for all cores

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

If multiple cores are connected to a single MDM with master ports enabled, there are two options to implement run-time memory access:

OFF	Use a separate LMB master for each core. This is the default setting. In this mode, the debugger will always use the LMB master corresponding to the core. If the core at debug port x is debugged, the debugger uses LMB master x. Use this mode if different LMB slaves are mapped to the same address on different cores,
ON	Use a single LMB master for all cores. In this mode, the debugger always uses LMB master 0. Use this mode if the cores have a common address map.

TERM.METHOD.MDMUART — Terminal configuration

Format:	**TERM.METHOD.MDMUART**

Configures the TRACE32 terminal functionality to access the UART controller of the MDM core. Use this option when your design handles STDIO via MDM UART.

Sample script for opening term window attached to MDM UART core:

```
TERM.RESet                    ; be sure to reset term functionality
TERM.METHOD.MDMUART           ; configure MDM UART for stdio
TERM.SIZE 110. 1000.          ; cosmetics
TERM.GATE                     ; make T32 poll the target for data
```

To confirm if the MDM UART is enabled in your design, open the peripheral window via the **PER** command and look for the section “MDM UART Configuration”.

Memory Classes

The following memory classes are available:

Memory Class	Description
P	Program memory
D	Data memory

Register Names

In TRACE32, the general purpose registers (`R0-R31`) and special purpose registers (e.g. `MSR` - machine state register, `SLR` - Stack low register etc.) are named according to the convention in the *MicroBlaze Processor Reference Guide* and shown accordingly in the **Register.view** window.

These names are also used in the disassembly views and the **Data.Assemble** command. This is in deviation from the Xilinx suggestions to use `rmsr`, `rslr`, etc. in the context of assembly language.

```
Data.Assemble 0x1000 mfs r0, MSR
Data.Assemble 0x1004 mts SLR, r3
```

SYStem.CPU

Select the used CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**MicroBlaze** | **ZYNQ-ULTRASCALE+-PMU** **MicroBlaze0** | **MicroBlaze1** | **MicroBlaze2** | **MicroBlaze3** (deprecated)

This command selects the CPU that shall be debugged.

For softcores instaniated in an FPGA design, select “MicroBlaze”. For the special case of the PMU in a Zynq UltraScale+ MPSOC, select “ZYNQ-ULTRASCALE+-PMU”.

The deprecated options were used for selecting one of multiple cores in an FPGA design. Instead of using the deprecated options, the following sequence is recommended to attach to a specific core in an FPGA design:

```
SYStem.CONFIG MDM UserInst USER2
SYStem.CONFIG MDM DebugPort <core_to_use> ; port numbers start with 0

SYStem.Up
```

Note that **all the cores inside an FPGA share identical multicore settings** (PRE, POST values) because they are accessed via the same TAP controller implemented in the Xilinx MDM IP block.

SYStem.JtagClock — Selects the frequency for the debug interface

Format:	**SYStem.JtagClock** *<rate>* **SYStem.BdmClock** *<rate>* (deprecated)
<fixed>:	**1 000 000**…**25 000 000**

Selects the JTAG clock frequency for the debug interface.

For fast setup of the clock speed pre-configured buttons can be used to select commonly used frequencies. The default frequency is 1.0 MHz.

NOTE:	Buffers, additional loads or high capacities on the JTAG lines reduce the maximum operation frequency of the JTAG clock and should be avoided.

SYStem.LOCK — Lock and tristate the debug port

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

Default: OFF.

If the system is locked, no access to the debug port will be performed by the debugger. While locked, the debug connector of the debugger is tristated. The main intention of the **SYStem.LOCK** command is to give debug access to another tool.

Format:	**SYStem.MemAccess** <*mode*>
<*mode*>:	**Denied** **Enable** **StopAndGo**

Run-time memory access is possible for Microblaze cores via the optional master ports of the MDM. These have to first be enabled and connected in Vivado.

Denied	No memory access is possible while the CPU is executing the program.
Enable CPU (deprecated)	Accesses are performed via the MicroBlaze Debug Module.
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.

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** **NoDebug** **Go** **Attach** **Up**

Select target reset mode.

Down	Disables the Debugger. The state of the CPU remains unchanged.
NoDebug	Resets the target with debug mode disabled (for the PPC400 family the same as Go). In this mode no debugging is possible. The CPU state keeps in the state of NoDebug
Go	Resets the target with debug mode enabled and prepares the CPU for debug mode entry. After this command the CPU is in the system.up mode and running. Now, the processor can be stopped with the break command or until any break condition occurs.
Up	Resets the target and sets the CPU to debug mode. After execution of this command the CPU is stopped and prepared for debugging. All register are set to the default value.
Attach	This command works similar to Up command. The difference is that the target CPU is not reset. The BDM/JTAG/COP interface will be synchronized and the CPU state will be read out. After this command the CPU is in the SYStem.Up mode and can be stopped for debugging.
StandBy	Not supported.

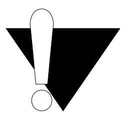
Format:	**SYStem.CONFIG** *<parameter> <number_or_address>* **SYStem.MultiCore** *<parameter> <number_or_address>* (deprecated)
<parameter>:	**CORE** *<core>* **SWDPIdleHigh** [**ON** | **OFF**] **SWDPTargetSel** *<value>*
<parameter>: (JTAG)	**DRPRE** *<bits>* **DRPOST** *<bits>* **IRPRE** *<bits>* **IRPOST** *<bits>* **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**] **Slave** [**ON** | **OFF**]

<parameter>: **(AccessPorts)**	**AHBAPn.Base** *<address>* **AHBAPn.HPROT** [*<value>* | *<name>*] **AHBAPn.Port** *<port>* **AHBAPn.RESet** **AHBAPn.view** **AHBAPn.XtorName** *<name>* **APBAPn.Base** *<address>* **APBAPn.HPROT** [*<value>* | *<name>*] **APBAPn.Port** *<port>* **APBAPn.RESet** **APBAPn.view** **APBAPn.XtorName** *<name>* **AXIAPn.ACEEnable** [**ON** | **OFF**] **AXIAPn.Base** *<address>* **AXIAPn.CacheFlags** *<value>* **AXIAPn.HPROT** [*<value>* | *<name>*] **AXIAPn.Port** *<port>* **AXIAPn.RESet** **AXIAPn.view** **AXIAPn.XtorName** *<name>* **DEBUGAPn.Port** *<port>* **DEBUGAPn.RESet** **DEBUGAPn.view** **DEBUGAPn.XtorName** *<name>* **JTAGAPn.Base** *<address>* **JTAGAPn.Port** *<port>* **JTAGAPn.CorePort** *<port>* **JTAGAPn.RESet**
<parameter>: (AccessPorts cont.)	**JTAGAPn.view** **JTAGAPn.XtorName** *<name>* **MEMORYAPn.HPROT** [*<value>* | *<name>*] **MEMORYAPn.Port** *<port>* **MEMORYAPn.RESet** **MEMORYAPn.view** **MEMORYAPn.XtorName** *<name>*

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are required to inform the debugger about the TAP controller position in the JTAG chain, if there is more than one core in the JTAG chain (e.g. Arm + DSP). The information is required before the debugger can be activated e.g. by a **SYStem.Up**. See **Daisy-chain Example**.

For some CPU selections (**SYStem.CPU**) the above setting might be automatically included, since the required system configuration of these CPUs is known.

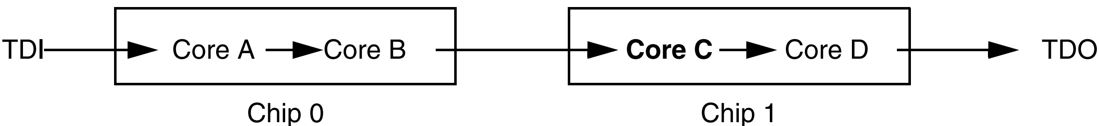
TriState has to be used if several debuggers (“via separate cables”) are connected to a common JTAG port at the same time in order to ensure that always only one debugger drives the signal lines. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to 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.

	Multicore debugging is not supported for the DEBUG INTERFACE (LA-7701).

CORE	For multicore debugging one TRACE32 PowerView GUI has to be started per core. To bundle several cores in one processor as required by the system this command has to be used to define core and processor coordinates within the system topology. Further information can be found in **SYStem.CONFIG.CORE**.
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).
DRPRE	(default: 0) *<number>* of TAPs in the JTAG chain between the core of interest and the TDO signal of the debugger. If each core in the system contributes only one TAP to the JTAG chain, DRPRE is the number of cores between the core of interest and the TDO signal of the debugger.
DRPOST	(default: 0) *<number>* of TAPs in the JTAG chain between the TDI signal of the debugger and the core of interest. If each core in the system contributes only one TAP to the JTAG chain, DRPOST is the number of cores between the TDI signal of the debugger and the core of interest.
IRPRE	(default: 0) *<number>* of instruction register bits in the JTAG chain between the core of interest and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the core of interest and the TDO signal of the debugger.

IRPOST	(default: 0) *<number>* of instruction register bits in the JTAG chain between the TDI signal and the core of interest. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the core of interest.
TAPState	(default: 7 = Select-DR-Scan) This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.
TCKLevel	(default: 0) Level of TCK signal when all debuggers are tristated.
TriState	(default: OFF) If several debuggers share the same debug port, this option is required. The debugger switches to tristate mode after each debug port access. Then other debuggers can access the port. JTAG: This option must be used, if the JTAG line of multiple debug boxes are connected by a JTAG joiner adapter to access a single JTAG chain.
Slave	(default: OFF) If more than one debugger 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).

Daisy-Chain Example



Below, configuration for core C.

Instruction register length of

- Core A: 3 bit
- Core B: 5 bit
- Core D: 6 bit

```
SYStem.CONFIG.IRPRE  6.                ; IR Core D
SYStem.CONFIG.IRPOST 8.                ; IR Core A + B
SYStem.CONFIG.DRPRE  1.                ; DR Core D
SYStem.CONFIG.DRPOST 2.                ; DR Core A + B
SYStem.CONFIG.CORE 0. 1.               ; Target Core C is Core 0 in Chip 1
```

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

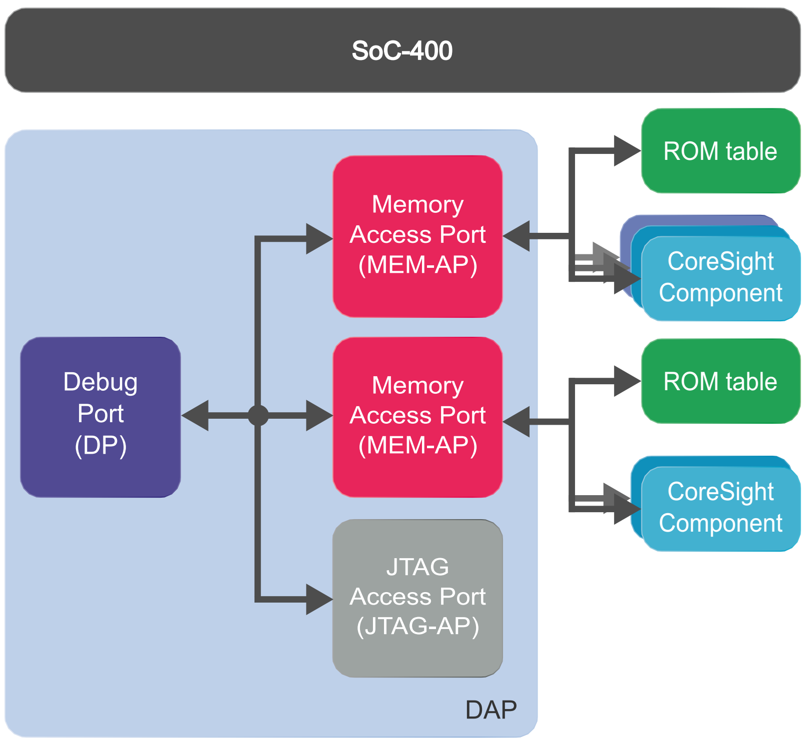
An Access Port (AP) is a CoreSight module from Arm which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

1. Different memory buses (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:”, “DP:”, “E:”. The interface to these buses is called Memory Access Port (MEM-AP).

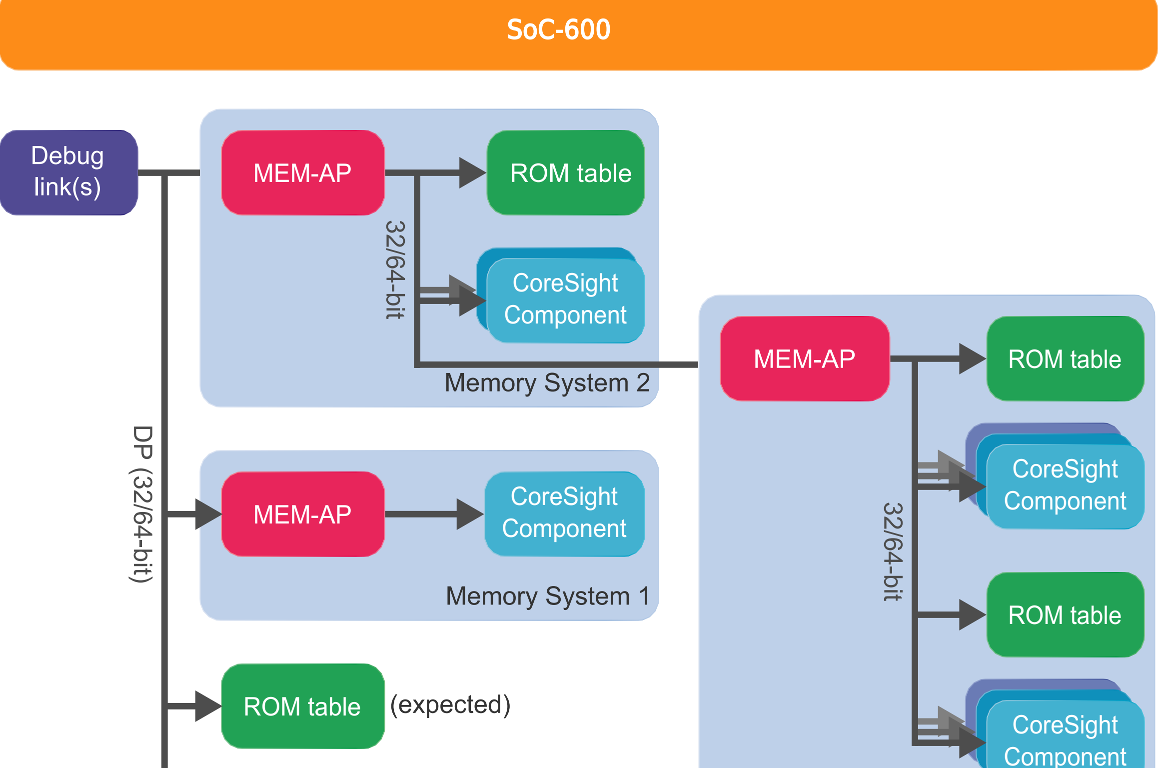
2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number between 0 and 7 denotes the JTAG interfaces to be addressed.

3. 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.AHBHPROT** [*<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.
APBAPn.HPROT [*<value>* | *<name>*]	Default: 0. This option selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight APB Access Port, when using the APB: memory class. The secure access bit HPROT[1] is not controlled by this option, but via the access class prefixes “Z” and “N” as well as “L” and “O” if the Access Port supports Realm Management Extension.

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. The secure access bit HPROT[1] is not controlled by this option, but via the access class prefixes “Z” and “N” as well as “L” and “O” if the Access Port supports Realm Management Extension.
MEMORYAPn.HPROT [*<value>* | *<name>*] **SYStem.Option.MEMORYHPROT** [*<value>* | *<name>*] (deprecated)	Default: 0. This option selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight Memory Access Port, when using the E: 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

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.
DEBUGAPn.XtorName *<name>*	APB bus transactor name identifying the bus where the debug register can be found. Used for “DAP:” access class.
MEMORYAPn.XtorName *<name>*	AHB bus transactor name identifying the bus where system memory can be accessed even during runtime. Used for “E:” access class while running.

... .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.Port *<port>*	Access Port Number (0-255) of a SoC-400 system which shall be used for “AHBn:” access class. Default: *<port>*=0.
APBAPn.Port *<port>*	Access Port Number (0-255) of a SoC-400 system which shall be used for “APBn:” access class. Default: *<port>*=1.
AXIAPn.Port *<port>*	Access Port Number (0-255) of a SoC-400 system which shall be used for “AXIn:” access class. Default: port not available.
DEBUGAPn.Port *<port>*	AP access port number (0-255) of a SoC-400 system where the debug register can be found (typically on APB). Used for “DAP:” access class. Default: *<port>*=1.
JTAGAPn.CorePort *<port>*	JTAG-AP port number (0-7) connected to the core which shall be debugged.
JTAGAPn.Port *<port>*	Access port number (0-255) of a SoC-400 system of the JTAG Access Port.
MEMORYAPn.Port *<port>*	AP access port number (0-255) of a SoC-400 system where system memory can be accessed even during runtime (typically an AHB). Used for “E:” access class while running. Default: *<port>*=0.

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.

Format:	**SYStem.CONFIG.CORE** *<core_index> <chip_index>* **SYStem.MultiCore.CORE** *<core_index> <chip_index>* (deprecated)
<chip_index>:	**1 … i**
<core_index>:	**1 … k**

Default *core_index*: depends on the CPU, usually 1. for generic chips

Default *chip_index*: derived from CORE= parameter of the configuration file (config.t32). The CORE parameter is defined according to the start order of the GUI in T32Start with ascending values.

To provide proper interaction between different parts of the debugger, the systems topology must be mapped to the debugger's topology model. The debugger model abstracts chips and sub cores of these chips. Every GUI must be connect to one unused core entry in the debugger topology model. Once the **SYStem.CPU** is selected, a generic chip or non-generic chip is created at the default *chip_index.*

Non-generic Chips

Non-generic chips have a fixed number of sub cores, each with a fixed CPU type.

Initially, all GUIs are configured with different *chip_index* values. Therefore, you have to assign the *core_index* and the *chip_index* for every core. Usually, the debugger does not need further information to access cores in non-generic chips, once the setup is correct.

Generic Chips

Generic chips can accommodate an arbitrary amount of sub-cores. The debugger still needs information how to connect to the individual cores e.g. by setting the JTAG chain coordinates.

Start-up Process

The debug system must not have an invalid state where a GUI is connected to a wrong core type of a non-generic chip, two GUIs are connected to the same coordinate or a GUI is not connected to a core. The initial state of the system is valid since every new GUI uses a new *chip_index* according to its CORE= parameter of the configuration file (config.t32). If the system contains fewer chips than initially assumed, the chips must be merged by calling **SYStem.CONFIG.CORE**.

For MicroBlaze specific information please refer to **"Connecting to MicroBlaze Targets for Debug and Trace"** (app_microblaze.pdf).

SYStem.CONFIG.state

Display target configuration

Format:	**SYStem.CONFIG.state**

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.

SYStem.CONFIG.MDM.Base

Select MDM base address

[build 142333 - DVD 02/2022]

Format:	**SYStem.CONFIG.MDM.Base** *<address>*

Defines the base address of the MDM.

SYStem.CONFIG.MDM.DebugPort

Set core to debug

Format:	**SYStem.CONFIG.MDM.DebugPort** *<core>*
<core>	**0. ... 31. |** **NONE** (deprecated)

This command selects which of the core(s) attached to a MicroBlaze Debug Module (MDM) is to be debugged by the current GUI.

The first core connected to the MDM is always numbered 0. Connecting to the core will fail if the selected number exceeds the number of debug ports of the MDM.

If the command is not used or if the special value NONE is used, the core index is determined by the **CORE.ASSIGN** and **SYStem.CONFIG CORE** commands. This method is deprecated, do not use it in new scripts.

NOTE:	If your design has multiple MDMs, use the command **SYStem.CONFIG.MDM DebugPort** to make sure the debugger connects to the correct MDM instance.

SYStem.CONFIG.MDM.RESet — Reset MDM configuration

Format:	**SYStem.CONFIG.MDM.RESet**

Resets MDM settings.

SYStem.CONFIG.MDM.view — Display MDM configuration

Format:	**SYStem.CONFIG.MDM.view**

Displays MDM settings.

SYStem.CONFIG.MDM.UserInst — Set default user BSCAN port

Format:	**SYStem.CONFIG.MDM.UserInst** *<inst>* **SYStem.Option.UserBSCAN** ***<...>*** (deprecated)
<inst>	**USER1** | **USER2** | **USER3** | **USER4**

Default: USER2 (same as the default when instantiating a MicroBlaze Debug Module (MDM) in the Vivado block design editor).

This command selects the JTAG instruction used for communicating with the FPGA. Xilinx FPGAs offer four different instructions, which can be used by different IP inside the FPGA.

If connection to the MDM via the specified instruction fails, the debugger will try the other instructions. Therefore, it is only required that you set the correct instruction if you either

- have multiple MDM instances in your FPGA design and wish to select a specific one, or
- have other IP in your FPGA design (e. g. an Integrated Logic Analyzer (ILA)) that uses a JTAG user instructions and want to avoid disturbing that IP with the debugger's attempt to connect to the MDM.

NOTE:	Usually, a design with multiple MicroBlaze cores will use a single MDM shared by all cores. In that case, you need to use the command **SYStem.CONFIG.MDM DebugPort** to select the desired core.

TrOnchip Commands

TrOnchip.state — Display on-chip trigger window

Format:	**TrOnchip.state**

Opens the **TrOnchip.state** window.

TrOnchip.RESet — Set on-chip trigger to default state

Format:	**TrOnchip.RESet**

Sets the TrOnchip settings and trigger module to the default settings.

TrOnchip.CONVert — Adjust range breakpoint in on-chip resource

Format:	**TrOnchip.CONVert** [**ON**	**OFF**] (deprecated) **Use Break.CONFIG.InexactAddress instead**

The on-chip breakpoints can only cover specific ranges. If a range cannot be programmed into the breakpoint, it will automatically be converted into a single address breakpoint when this option is active. This is the default. Otherwise an error message is generated.

```
TrOnchip.CONVert ON
Break.Set 0x1000..0x17ff /Write        ; sets breakpoint at range
Break.Set 0x1001..0x17ff /Write        ; 1000..17ff sets single breakpoint
…                                      ; at address 1001

TrOnchip.CONVert OFF                   ; sets breakpoint at range
Break.Set 0x1000..0x17ff /Write        ; 1000..17ff
Break.Set 0x1001..0x17ff /Write        ; gives an error message
```

Format:	**TrOnchip.VarCONVert** [**ON** | **OFF**] (deprecated) **Use Break.CONFIG.VarConvert instead**

The on-chip breakpoints can only cover specific ranges. If you want to set a marker or breakpoint to a complex variable, the on-chip break resources of the CPU may be not powerful enough to cover the whole structure. If the option **TrOnchip.VarCONVert** is set to **ON**, the breakpoint will automatically be converted into a single address breakpoint. This is the default setting. Otherwise an error message is generated.

©1989-2026 Lauterbach

MMU.DUMP — Page wise display of MMU translation table

Format:	**MMU.DUMP** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*]
<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**.

MMU.List — Compact display of MMU translation table

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>*]
<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.

NOTE:	MMU translation tables (page tables) are dynamic structures and are frequently modified by the OS. Instead of **MMU.SCAN**, use **TRANSlation.TableWalk ON** to enable the debugger table walk. This method dynamically parses the page tables on demand for every debugger address translation. The debugger table walk is faster than repetitive **MMU.SCAN** calls and ensures that the debugger address translations correspond to the current OS address translations.

Real-Time Trace

This sections list CPU specific options for the real-time trace.

SYStem.Option.DTM — Control data trace messages

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

Default: OFF.

Enable this system option in order to record data trace messages of the target program. Note that MicroBlaze XMTC only supports tracing of data load messages. Data write messages can not be triggered.

The option needs to be enabled before connecting the debugger to the target.

NOTE:	To see data trace messages in the **<trace>.List** window, it is necessary to increase the level of displayed details by clicking the **more** button.

SYStem.Option.QUICKSTOP — Control trace of software breakpoints

Format:	**SYStem.Option.QUICKSTOP** [**ON** | **OFF**] (deprecated)

Default: OFF.

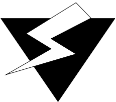
Enable this system option in order to optimize tracing of software breakpoints.

When hitting a software break, earlier versions of MicroBlaze jump to a software break handler and loop there until the debugger detects the break. As this can last some milliseconds, the trace buffer will contain irrelevant trace data.

By enabling the option is enabled, the debugger sets an on-chip breakpoint onto the software break handler and thus stops the core immediately.

Configuring your FPGA

Before debugging, the FPGA needs to be configured with a design containing a MicroBlaze core enabled for JTAG debugging. The configuration is done via the command **JTAG.PROGRAM** or its predecessor **Java.JTAG.LOADBIT**. .

	Be sure to have **correct multicore settings** before configuring the FPGA, otherwise the configuration will fail. These settings are identical with those used for debugging a MicroBlaze core. Also ensure that the debugger is in **SYStem.down mode**, before configuring your FPGA. Configuring the FPGA will break the communication link between the debugger and the MicroBlaze core, if your debugger is in SYStem.up mode. Configuration **using compressed bitstreams is supported.** It is recommended to configure the target with the configuration option "**JTAG dedicated**" i.e. not using a mode where JTAG overrides other configurations like MSI, SPI etc. In the latter case configuration via TRACE32 may fail silently (no error message), though configuration via Xilinx Impact works.

JTAG Connector

Mechanical Description

JTAG Connector for Xilinx Microblaze

It is recommended to connect all N/C Pins to GND (if you work with LAUTERBACH tools only).

The following chart details the pinout of the16 pin **PPC400 debug cable**, that is also used for debugging Microblaze cores.

Signal	Pin	Pin	Signal
TDO	1	2	N/C
TDI	3	4	TRST- (*)
N/C	5	6	VCCS
TCK	7	8	N/C
TMS	9	10	N/C
HALT-	11	12	N/C
N/C	13	14	KEY
N/C	15	16	GND

Pinout of PPC400 debug cable

The debugger includes the adapter (LA-3731) that converts the PPC400 pinout to that of the 14 pin Xilinx JTAG connector which is listed below:

Signal	Pin	Pin	Signal
GND	1	2	VREF
GND	3	4	TMS
GND	5	6	TCK
GND	7	8	TDO
GND	9	10	TDI
GND	11	12	NC
GND	13	14	NC

Pinout of Xilinx JTAG connector

NOTE:	The HALT- and TRST- signals are irrelevant for debugging MicroBlaze designs.They are only used for debugging the boot process of PPC cores. See the PowerPC debugger user guide for details.

•