LAUTERBACH
DEVELOPMENT TOOLS

Cortex-M Debugger

Release 09.2024

MANUAL

Cortex-M Debugger

Version 05-Oct-2024

History

31-Jul-23	Chapter 'Products for Debugging and Tracing Cortex-M Cores' updated for latest configurations.
20-Jul-23	Chapter 'Breakpoints' fully revised.
28-Nov-22	New command SYStem.Option.CLEARHARDFAULT.
14-Nov-22	New command SYStem.Option.CFLUSHAFTERBREAK.
05-Oct-22	New commands BMC.ON and BMC.OFF.
19-Aug-22	Link to manual **XCP Debug Back-End** added in chapter' Brief Overview of Documents for New Users'.
06-Jul-22	New command SYStem.Option.FORCESECure.
15-Jun-22	New subchapter 'XCP Specific Commands', describes the XCP subcommands of SYStem.CONFIG.
22-Apr-22	New command SYStem.Option.MDMAP.

Warning

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.

Introduction

This document describes the processor-specific settings and features for the Cortex-M debugger.

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

Brief Overview of Documents for New Users

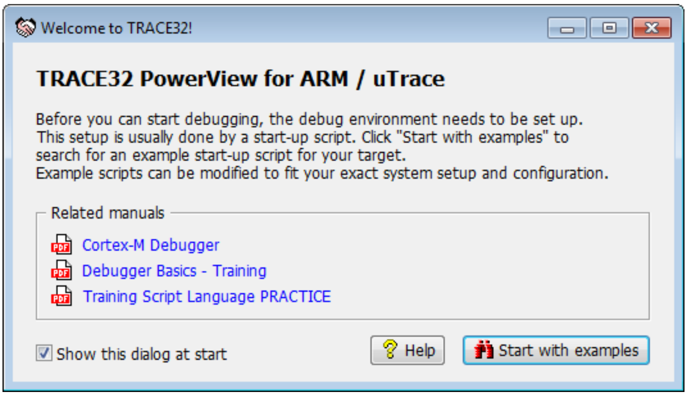
Architecture-independent information:

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

Architecture-specific information:

- **“Processor Architecture Manuals”**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **“OS Awareness Manuals”** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.
- This manual does not cover the Cortex-A/R (ARMv7, 32-bit) cores. If you are using this processor architecture, please refer to **“Arm Debugger”** (debugger_arm.pdf).
- This manual does not cover the Cortex-A/R (Armv8 and Armv9, 32/64-bit) cores. If you are using these processor architectures, please refer to **“Armv8 and Armv9 Debugger”** (debugger_armv8v9.pdf).
- **“XCP Debug Back-End”** (backend_xcp.pdf): This manual describes how to debug a target over a 3rd-party tool using the XCP protocol.

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



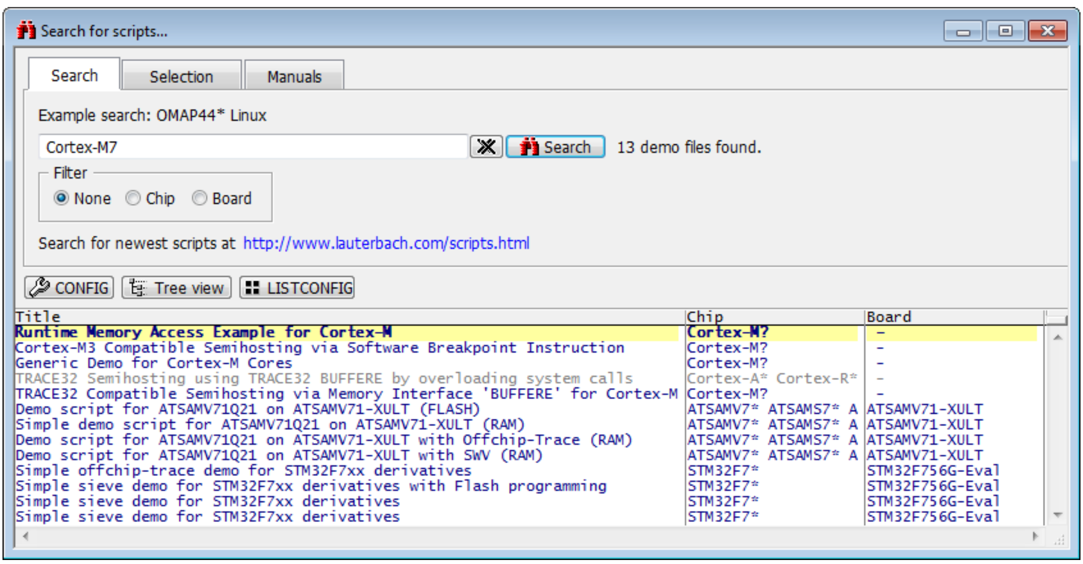
Demo and Start-up Scripts

Lauterbach provides ready-to-run PRACTICE start-up scripts for known Cortex-M-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/arm/` subfolder of the system directory of TRACE32.

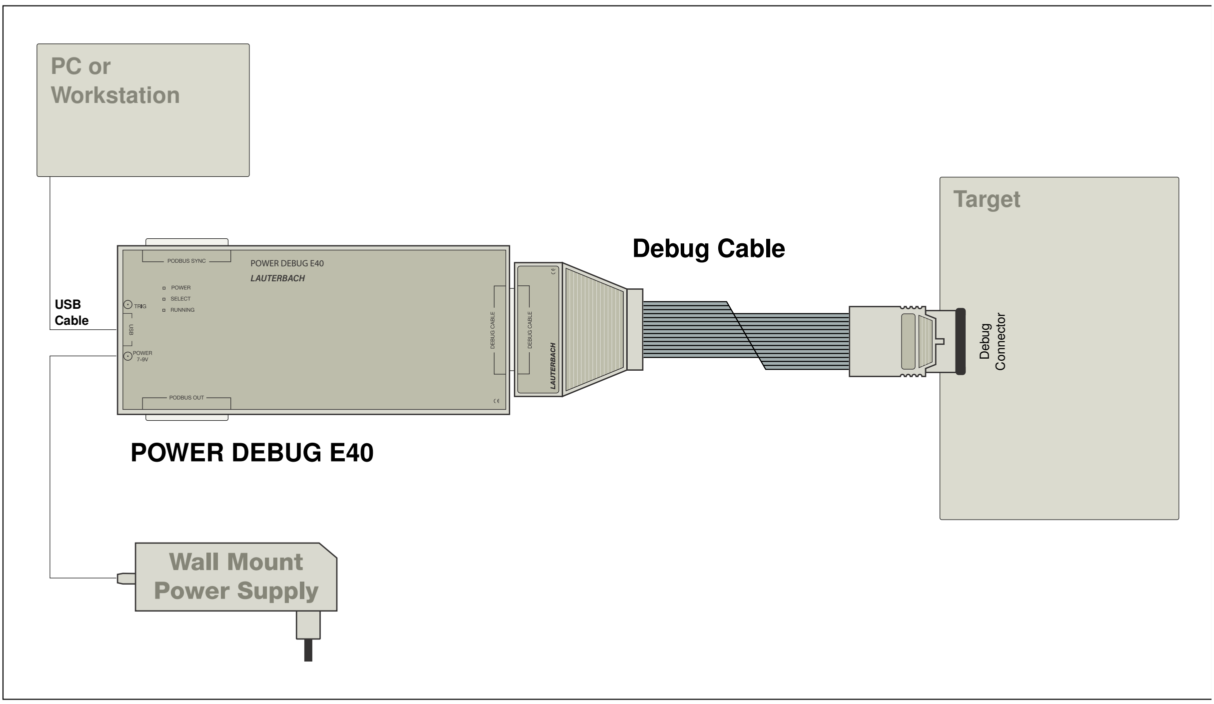
Products for Debugging and Tracing Cortex-M Cores

Lauterbach offers different tool configurations for debugging and tracing of Cortex-M cores. This chapter presents the individual configurations and their main applications briefly.

The following configurations are provided:

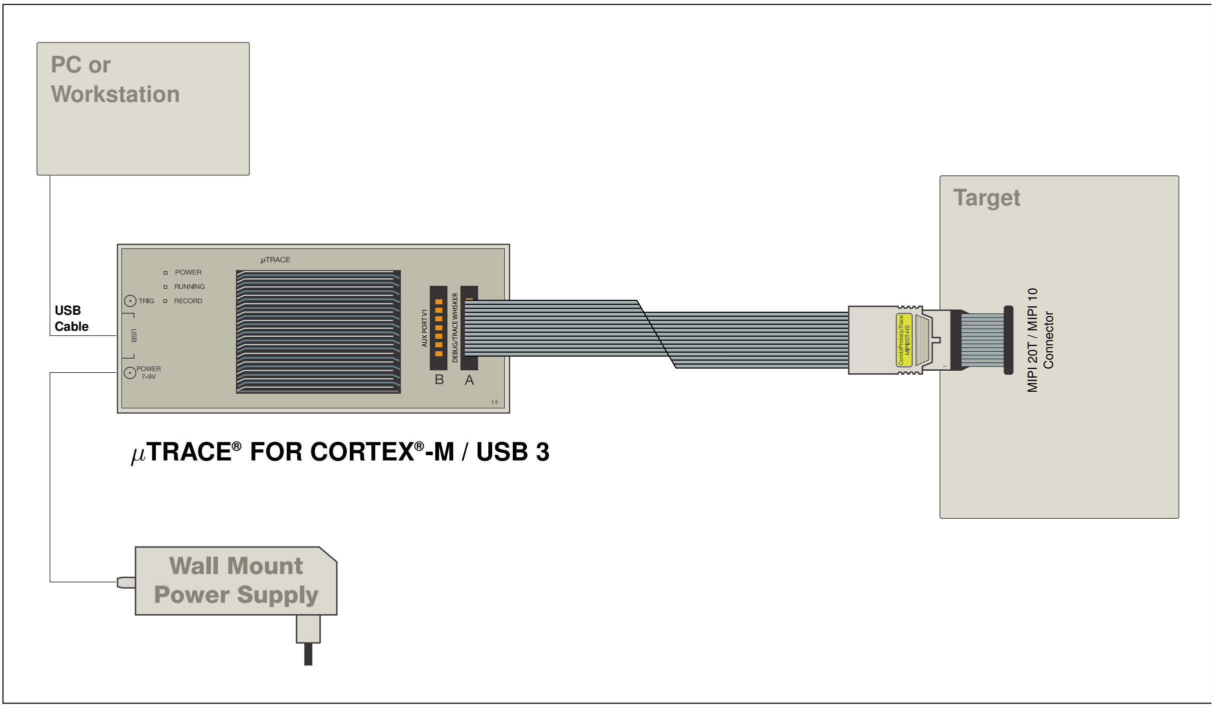
- PowerDebug and Debug Cable
- µTrace (with MIPI20T-HS Whisker)
- PowerDebug and CombiProbe (with MIPI20T-HS Whisker)
- PowerDebug and CombiProbe (with CombiProbe MIPI34 Whisker)
- Power Debug and PowerTrace

PowerDebug and Debug Cable



You have chosen a pure debug solution because your processor has no off-chip trace option or you have no interest in off-chip tracing.

For all Cortex-M specific debug features, please refer to **"Cortex-M Debugger"** (debugger_cortexm.pdf).



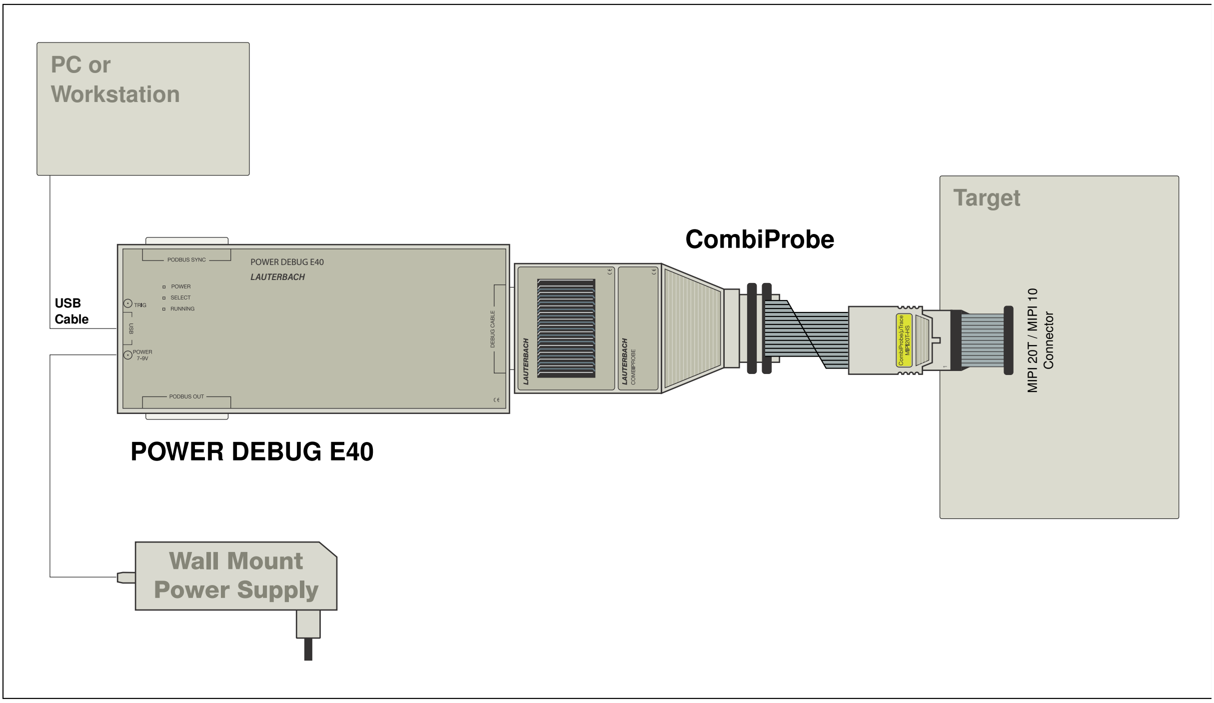
You have chosen the all-in-one debug and off-chip trace solution developed by Lauterbach especially for Cortex-M processors.

For all Cortex-M specific debug features, please refer to **"Cortex-M Debugger"** (debugger_cortexm.pdf).

For all Cortex-M specific trace features, please refer to **"MicroTrace for Cortex-M User's Guide"** (microtrace_cortexm.pdf).

Optionally, the TRACE32 Mixed-Signal Probe can be used with the µTrace for Cortex-M.

PowerDebug and CombiProbe (with MIPI20T-HS Whisker)



You have chosen a debug and off-chip trace solution for your processor which is tailor-made for the Cortex-M, but provides you with greater flexibility than the all-in-one debug and trace solution µTrace (MicroTrace). The combination of CombiProbe and MIPI20T-HS whisker supports:

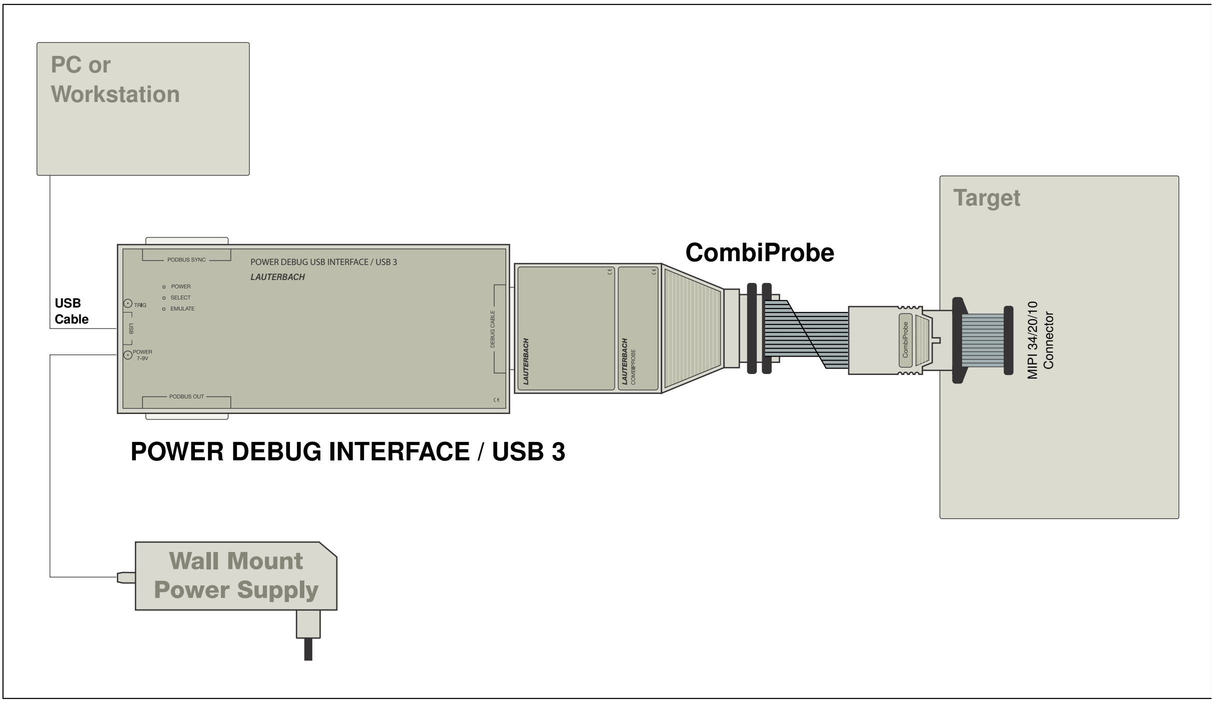
- Debugging via JTAG (IEEE 1149.1), SWD (Serial Wire Debug) or cJTAG (IEEE 1149.7) at clock rates up to 100 MHz
- Debug connectors MIPI20T and MIPI10 (without adapter), Arm-20 (with included adapter)
- Parallel trace using ETM/ITM in TPIU continuous mode with up to 4 data pins and bit rates of up to 400 Mbit/s per pin
- SWV (Serial Wire Viewer) / SWO (Serial Wire Output) trace at port rates up to 200 Mbit/s
- Automatic configuration and advanced diagnostics of electrical parameters of the used trace port
- Optional logic analyzer extension TRACE32 Mixed-Signal Probe.
- Debugging of CPU types other than Cortex-M (e. g. Cortex-A/R)
- Debugging two chips with two separate debug connectors (using a second whisker cable)

This combination requires TRACE32 R.2018.09 or newer.

For all Cortex-M specific debug features, please refer to **"Cortex-M Debugger"** (debugger_cortexm.pdf).

For all Cortex-M specific trace features, please refer to **"CombiProbe for Cortex-M User's Guide"** (combiprobe_cortexm.pdf).

PowerDebug and CombiProbe (with CombiProbe MIPI34 Whisker)



This solution is outdated.

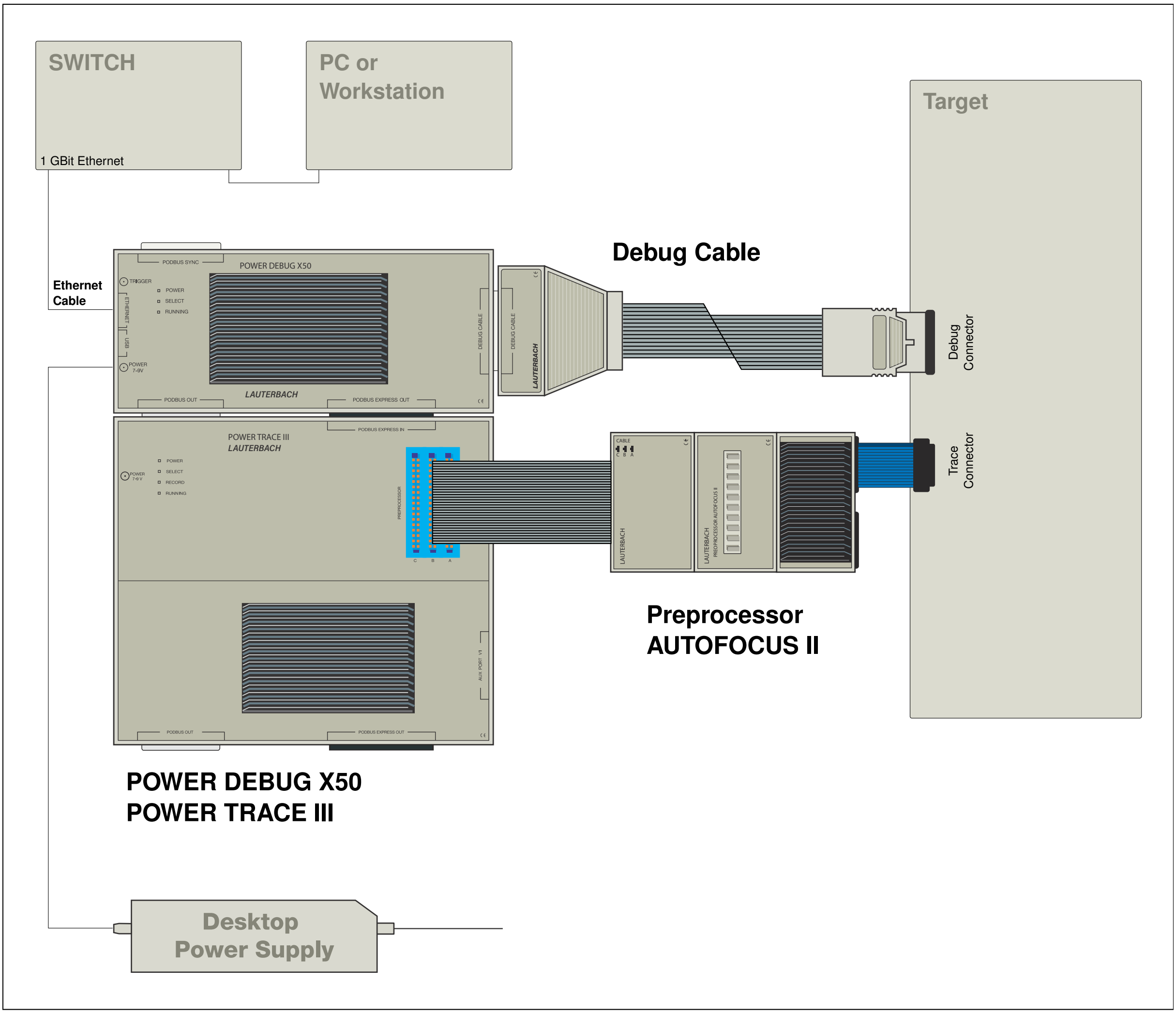
You have chosen a debug and off-chip trace solution for your processor which is tailor-made for the Cortex-M, but provides you with greater flexibility than the all-in-one debug and trace solution µTrace (MicroTrace). The combination of CombiProbe and MIPI34 whisker supports:

- Debugging via JTAG (IEEE 1149.1), SWD (Serial Wire Debug) or cJTAG (IEEE 1149.7) at clock rates up to 100 MHz
- Debug connectors MIPI34, MIPI20D, MIPI20T and MIPI10 (without adapter), Arm-20 (with included adapter)
- Parallel trace using ETM/ITM/STM either in TPIU continuous mode or without a TPIU with up to 4 data pins and bit rates of up to 200 Mbit/s per pin
- SWV (Serial Wire Viewer) / SWO (Serial Wire Output) trace at port rates up to 100 Mbit/s
- Optional logic analyzer extension (PowerProbe or PowerIntegrator)
- Debugging of CPU types other than Cortex-M (e. g. Cortex-A/R)
- Debugging two chips with two separate debug connectors (using a second whisker cable)

For all Cortex-M specific debug features, please refer to **“Cortex-M Debugger”** (debugger_cortexm.pdf).

For all Cortex-M specific trace features, please refer to **“CombiProbe for Cortex-M User’s Guide”** (combiprobe_cortexm.pdf).

PowerDebug and PowerTrace (X-License)



You have the TRACE32 high-end debug and off-chip trace solution for your processor and it is likely that your Cortex-M is part of a complex SoC.

For all Cortex-M specific debug features, please refer to **"Cortex-M Debugger"** (debugger_cortexm.pdf).

For all Cortex-M specific trace features, please refer to **"Training Cortex-M Tracing"** (training_cortexm_etm.pdf).

Quick Start of the JTAG Debugger

Starting up the debugger is done as follows:

1. Select the device prompt for the ICD Debugger and reset the system.

```
B::

RESet
```

   The device prompt `B::` is normally already selected in the TRACE32 command line. If this is not the case, enter `B::` to set the correct device prompt. The **RESet** command is only necessary if you do not start directly after booting the TRACE32 development tool.

2. Specify the core specific settings.

```
SYStem.CPU <cpu_type>

SYStem.Option.EnReset [ON|OFF]
```

   The default values of all other options are set in such a way that it should be possible to work without modification. Please consider that this is probably not the best configuration for your target.

3. Inform the debugger about read-only address ranges (ROM, FLASH).

```
MAP.BOnchip 0x100000++0x0fffff
```

   The B(reak)Onchip information is necessary to decide where on-chip breakpoints must be used. On-chip breakpoints are necessary to set program breakpoints to FLASH/ROM.

4. Enter debug mode.

```
SYStem.Up
```

   This command resets the core and enters debug mode. After this command is executed, it is possible to access memory and registers.

5. Load stack pointer and program counter from the vector table.

```
Register.Init
```

6. Load the program.

```
Data.LOAD.AIF armf                    (aif specifies the format,
                                      armf is the file name)
```

The format of the **Data.LOAD** command depends on the file format generated by the compiler.

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

A typical start sequence 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 CORTEXM3                 ;Select the core type

MAP.BOnchip 0x100000++0xfffff       ;Specify where FLASH/ROM is

SYStem.Up                           ;Reset the target and enter debug
                                    ;mode

Register.Init                       ;Load stack pointer and program
                                    ;counter

Data.LOAD.AXF armf                  ;Load the application

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

Register.Set R13 0x8000             ;Set the stack pointer to address
                                    ;8000

List.Mix                            ;Open source code window    *)

Register.view /SpotLight            ;Open register window       *)

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

Break.Set 0x1000 /p                 ;Set software breakpoint to address
                                    ;1000 (address 1000 outside of
                                    ;BOnchip range)

Break.Set 0x101000 /p               ;Set on-chip breakpoint to address
                                    ;101000 (address 101000 is within
                                    ;BOnchip range)
```

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

Communication between Debugger and Processor Cannot Be Established

Typically the **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages like “debug port fail” or “debug port timeout” while executing this command, this may have the reasons below. “target processor in reset” is just a follow-up error message. Open the **AREA.view** window to see all error messages.

- The target has no power or the debug cable is not connected to the target. This results in the error message “target power fail”.
- You did not select the correct core type **SYStem.CPU <type>**.
- There is an issue with the JTAG interface. See **“Arm Debug and Trace Interface Specification”** (app_arm_target_interface.pdf) and the manuals or schematic of your target to check the physical and electrical interface. Maybe there is the need to set jumpers on the target to connect the correct signals to the JTAG connector.
- There is the need to enable (jumper) the debug features on the target. It will e.g. not work if nTRST signal is directly connected to ground on target side.
- The debug access port is in an unrecoverable state. Re-power your target and try again.
- The target can not communicate with the debugger while in reset. Try **SYStem.Mode Attach** followed by Break instead of **SYStem.Up** or use **SYStem.Option.EnReset OFF**.
- The default frequency of the JTAG/SWD/cJTAG debug port is too high, especially if you emulate your core or if you use an FPGA-based target. In this case try **SYStem.JtagClock 50kHz** and optimize the speed when you got it working.
- The core is used in a multicore system and the appropriate multicore settings for the debugger are missing. See for example **SYStem.CONFIG.DAPIRPRE**. This is the case if you get a value IR_Width > 4 when you enter “DIAG 3400” and “AREA”. If you get IR_Width = 4, then you have just the Cortex-M3 and you do not need to set these options. If the value can not be detected, then you might have a JTAG interface issue.
- The core has no clock.
- The core is kept in reset.
- There is a watchdog which needs to be deactivated.

Your target needs special debugger settings. Check the directory \demo\arm if there is an suitable PRACTICE script file (*.cmm) for your target.

FAQ

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

Trace Extensions

A Embedded Trace Macrocell (ETM) might be integrated into the core. The Embedded Trace Macrocell provides program and data flow information plus trigger and filter features.

Please refer to the online help books **"Arm ETM Trace"** (trace_arm_etm.pdf) for detailed information about the usage of ETM.

Please note that you need to inform the debugger in the start-up script about the location of the trace control register and funnel configuration in case a trace bus is used. See **SYStem.CONFIG.ETMBASE**, **SYStem.CONFIG.FUNNELBASE**, **SYStem.CONFIG.TPIUBASE**, **SYStem.CONFIG.ETMFUNNELPORT**. In case a HTM or ITM module is available and shall be used you need also settings for that.

Breakpoints

Software Breakpoints

Software breakpoints are set by replacing the opcode of an instruction in RAM with a breakpoint instruction. Software breakpoints are used by TRACE32 by default to set breakpoints of the type Program. If TRACE32 detects that the opcode is not in RAM, it automatically sets a breakpoint of the method ONCHIP.

```
Break.Set my_function /Program
```

The number of software breakpoints is theoretically unlimited. However, a large number of software breakpoints reduces the overall performance of TRACE32.

For special use cases, TRACE32 can also set software breakpoints into flash memory. For details refer to **"Software Breakpoints in FLASH"** in Onchip/NOR FLASH Programming User's Guide, page 32 (norflash.pdf).

On-chip Breakpoints Cortex-M Armv6/v7

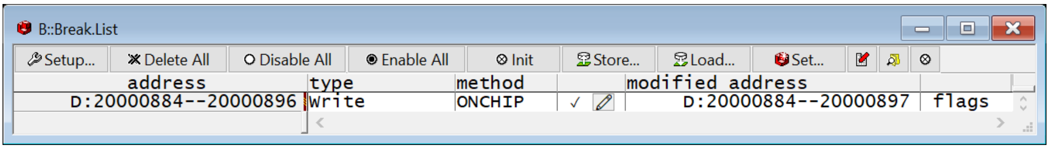
Onchip breakpoints are multifunctional and, most importantly, non-intrusive. However, TRACE32 depends on the breakpoint resources in the onchip debug logic.

The following basic usage scenarios are possible:

- **Program breakpoints:** Onchip breakpoints can be used to set program breakpoints into ROM/FLASH/EPROM. It is recommended to configure TRACE32 with the help of the command **MAP.BOnchip** *<flash_address_range>* so that onchip breakpoints are automatically used for the flash address range.

- **Read/Write breakpoints:** Onchip breakpoints can be used to stop the program execution when a read or write access to an address occurs. In the Arm notation these breakpoints are called watchpoints.

```
Var.Break.Set flags /Write
```



Since address ranges are realized via bit masks for the Cortex-M Armv6/v7, the address range of the breakpoint (modified address column in the above screenshot) extends beyond the address range of the variable (address column in the above screenshot). The program execution can thus be stopped even at a write access to the address directly after the variable.

- **Data value breakpoint:** Onchip breakpoints can be used to stop the program execution when a specific data value is written to an address or when a specific data value is read from an address.

```
Var.Break.Set flags[3] /Read /DATA.auto 0.
```

The following onchip resources can be used for the implementation of onchip breakpoints, depending on the Cortex-M core in use.

BPU	**BreakPoint Unit** The BPU allows to set single address breakpoints into the program address range 0x00000000 - 0x1fffffff.
DWU	**Data Watchpoint Unit** The DWU allows setting single address breakpoints and breakpoints to address ranges using bitmaks. Breakpoints implemented via the DWU are asynchronous, which means that program execution is stopped with a delay of several assembly instructions.

DWT **Data Watchpoint and Trace unit**

The DWT allows setting single address breakpoints and breakpoints to address ranges using bitmaks.

The DWT needs two breakpoint comparators to implement a data value breakpoint, one for the data address (range) and one for the data value.

Breakpoints implemented via the DWT are asynchronous, which means that program execution is stopped with a delay of several assembly instructions.

FPB **Flash Patch and Breakpoint unit**

The FPB allows to set single address breakpoints into the program address range 0x00000000 - 0x1fffffff.

Here is the list of the onchip breakpoints resources available for the individual Cortex-M cores.

	Onchip Break-points in Total	Program Breakpoints	Read/Write Breakpoints	Data Value Breakpoint
Cortex-M0/M0+	1-4 by BU 1-2 by DW	1-4 by BU 1-2 by DW	1-2 by DW	-
Cortex-M1	2/4 by BU 1/2 by DW	2/4 by BU 1/2 by DW	1/2 by DW	-
Cortex-M3	6 by FPB 4 by DWT	6 by FPB 4 by DWT	4 by DWT	1 by DWT
Cortex-M4	2/6 by FPB 1/4 by DWT	2/6 by FPB 1/4 by DWT	1/4 by DWT	0/1 by DWT
Cortex-M7	4/8 by FPB 2/4 by DWT	4/8 by FPB 2/4 by DWT	2/4 by DWT	1 by DWT
SC000	1-4 by BU 1-2 by DW	1-4 by BU 1-2 by DW	1-2 by DW	-
SC300	6 by FPB 4 by DWT	6 by FPB 4 by DWT	4 by DWT	1 by DWT

The command **Break.List /OnchipDetail** provides details about the usage of the onchip breakpoint resources.

Onchip breakpoints are multifunctional and, most importantly, non-intrusive. However, TRACE32 depends on the breakpoint resources in the onchip debug logic.

The following basic usage scenarios are possible:

- **Program breakpoints:** Onchip breakpoints can be used to set program breakpoints into ROM/FLASH/EPROM. It is recommended to configure TRACE32 with the help of the command **MAP.BOnchip** *<flash_address_range>* so that onchip breakpoints are automatically used for the flash address range.
- **Read/Write breakpoints:** Onchip breakpoints can be used to stop the program execution when a read or write access to an address occurs. In the Arm notation these breakpoints are called watchpoints.

```
Var.Break.Set flags /Write
```

- **Data value breakpoint:** Onchip breakpoints can be used to stop the program execution when a specific data value is written to an address or when a specific data value is read from an address.

```
Var.Break.Set flags[3] /Read /DATA.auto 0.
```

The following onchip resources can be used for the implementation of onchip breakpoints, depending on the Cortex-M core in use.

BPU	**BreakPoint Unit** The BPU provides single address breakpoints for the complete program address range.
DWT	**Data Watchpoint and Trace unit** The DWT allows setting single address breakpoints and breakpoints to address ranges. An address range needs two breakpoint comparators. The DWT needs 2 breakpoint comparators to implement a data value breakpoint, one for the data address and one for the data value. Breakpoints implemented via the DWT are asynchronous, which means that program execution is stopped with a delay of several assembly instructions.

FPB	**Flash Patch and Breakpoint Unit** The FBB provides single address breakpoints for the complete program address range.

Here is the list of the onchip breakpoints resources available for the individual Cortex-M cores.

	On-chip Breakpoints	**Instruction Breakpoints**	**Read/Write Breakpoints**	**Data Breakpoint**
Cortex-M23	4 by FPB 4 by DWT	4 by FPB 4 by DWT	4 by DWT	1 by DWT
Cortex-M33	4/8 by BPU 4 by DWT	4 by FPB 4 by DWT	4 by DWT	1 by DWT
Cortex-M35P	4/8 by BPU 4 by DWT	4/8 by BPU 4 by DWT	4 by DWT	1 by DWT
Cortex-M55	4/8 by BPU 2/4/8 by DWT	4/8 by BPU 2/4/8 by DWT	2/4/8 by DWT	1 by DWT 2 for DWT with 8
Cortex-M85	4/8 by BPU 4/8 by DWT	4/8 by BPU 4/8 by DWT	4/8 by DWT	1 by DWT 2 for DWT with 8
STAR	4/8 by BPU 4 by DWT	4/8 by BPU 4 by DWT	4 by DWT	1 by DWT

The command **Break.List /OnchipDetail** provides details about the usage of the onchip breakpoint resources.

Access Classes

This section describes the available ARM access classes and provides background information on how to create valid access class combinations in order to avoid syntax errors.

For background information about the term access class, see **"TRACE32 Concepts"** (trace32_concepts.pdf).

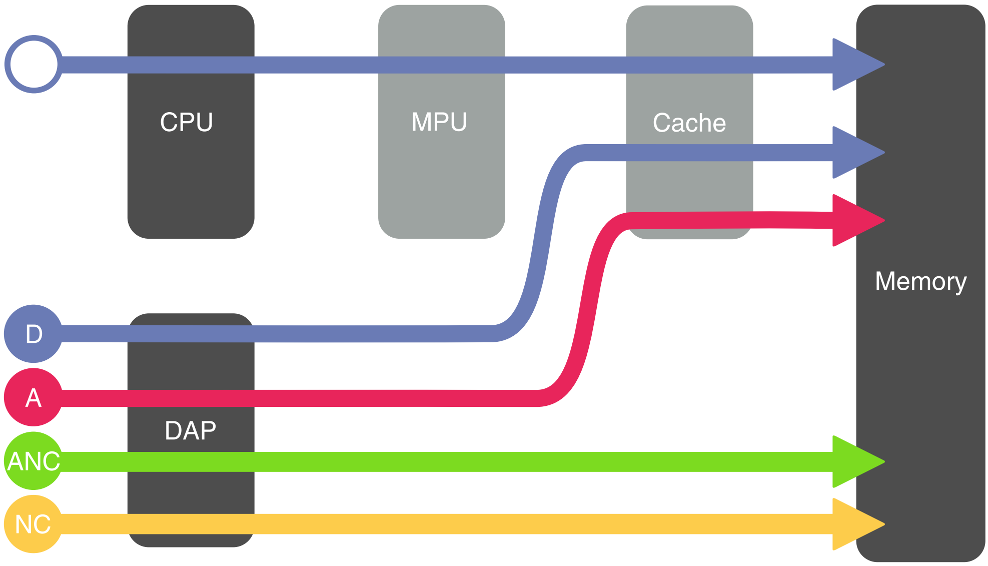
In this section:

Description of the Individual Access Classes

Access Class	Description
A	Absolute addressing (physical address)
AHB, AHB2	See DAP.
APB, APB2	See DAP.
AXI, AXI2	See DAP.
D	Data Memory

Access Class	Description
DAP, DAP2, AHB, AHB2, APB, APB2, AXI, AXI2	Memory access via bus masters, so named Memory Access Ports (MEM-AP), provided by a Debug Access Port (DAP). The DAP is a CoreSight component mandatory on Cortex based devices. Which bus master (MEM-AP) is used by which access class (e.g. AHB) is defined by assigning a MEM-AP number to the access class: SYStem.CONFIG DEBUGACCESSPORT *<mem_ap#>* -> “DAP” SYStem.CONFIG AHBACCESSPORT *<mem_ap#>* -> “AHB” SYStem.CONFIG APBACCESSPORT *<mem_ap#>* -> “APB” SYStem.CONFIG AXIACCESSPORT *<mem_ap#>* -> “AXI” You should assign the memory access port connected to an AHB (AHB MEM-AP) to “AHB” access class, APB MEM-AP to “APB” access class and AXI MEM-AP to “AXI” access class. “DAP” should get the memory access port where the debug register can be found which typically is an APB MEM-AP (AHB MEM-AP in case of a Cortex-M). There is a second set of access classes (DAP2, AHB2, APB2, AXI2) and configuration commands (e.g. SYStem.CONFIG DAP2AHBACCESSPORT *<mem_ap#>*) available in case there are two DAPs which needs to be controlled by the debugger.
E	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)
N	EL0/1 Non-Secure Mode (TrustZone devices)
P	Program Memory
R	AArch32 ARM Code (A32, 32-bit instr. length)
S	Supervisor Memory (privileged access)
T	AArch32 Thumb Code (T32, 16-bit instr. length)
U	User Memory (non-privileged access) not yet implemented; privileged access will be performed.
USR	Access to Special Memory via User-Defined Access Routines
JSEQ:	Access data via JTAG sequences registered with **JTAG.SEQuence.MemAccess.ADD**
VM	Virtual Memory (memory on the debug system)
X ARMv8-A only	AArch64 ARM64 Code (A64, 32-bit instr. length)
Z	Secure Mode (TrustZone devices)

Combinations of access classes are possible as shown in the example illustration below:



The access class “A” in the red path means “physical access”, i.e. it will only bypass the MMU but consider the cache content. As generic Cortex-M based designs do not feature an MMU, it is equal to the standard access class in such cases. The access class “NC” in the yellow path means “no cache”, so it will bypass the cache.

If both access classes “A” and “NC” are combined to “ANC”, this means that the properties of both access classes are summed up, i.e. both the MMU and the cache will be bypassed on a memory access. As generic Cortex-M based designs do not feature an MMU, it is equal to access class “NC” in such cases.

The blue path is an example of an access which is done when no access class is specified. It is similar to the memory view of the CPU core. The MPU is always bypassed when the debugger performs a memory access.

The access classes “A” and “NC” are not the only two access classes that can be combined. An access class combination can consist of up to five access class specifiers. But any of the five specifiers can also be omitted.

Three specifiers: Let's assume you want to view a secure memory region that contains Thumb code. To ensure a secure access, use the access class specifier "Z". And to make the debugger disassemble the memory content as Thumb code use "T". By combining both access class specifiers, we obtain the access class combination "ZT".

NOTE:	This example refers to secure/non-secure memory available on TrustZone enabled devices like Cortex-M23/M33.

```
List.Mix ZT:0x10000000 // View THUMB code in secure memory
```

One specifier: Let's imagine a physical access should be done. To accomplish that, start with the "A" access class specifier right away and omit all other possible specifiers.

```
Data.dump A:0x80000000 // Physical memory dump at address 0x80000000
```

No specifiers: Let's now consider what happens when you omit all five access class specifiers. In this case the memory access by the debugger will be a virtual access using the current CPU context, i.e. the debugger has the same view on memory as the CPU core.

```
Data.dump 0xFB080000   // Virtual memory dump at address 0xFB080000
```

Using no or just a single access class specifier is easy. Combining at least two access class specifiers is slightly more challenging because access class specifiers cannot be combined in an arbitrary order. Instead you have to take the syntax of the access class specifiers into account.

If we refer to the above example "ZT" again, it would not be possible to specify the access class combination as "TZ". You have to follow certain rules to make sure the syntax of the access class specifiers is correct. This will be illustrated in the next section.

How to Create Valid Access Class Combinations

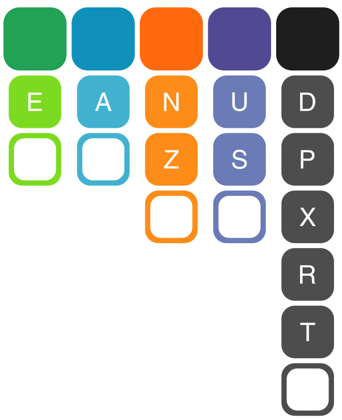
The illustrations below will show you how to combine access class specifiers for frequently-used access class combinations.

Rules to create a valid access class combination:

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

Memory Access through CPU (CPU View)

The debugger uses the CPU to access memory and peripherals like UART or DMA controllers. This means the CPU will carry out the accesses requested by debugger. Examples would be virtual, physical, secure, or non-secure memory accesses.

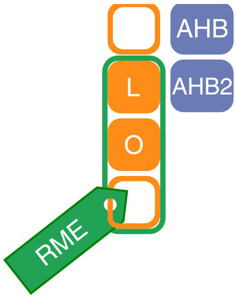


Example combinations:

AD	View physical data (current CPU mode).
ED	Access data at run-time.
ZSD	View data in secure supervisor mode at virtual address location.
AX	Disassemble code of ARMv8-A code section, e.g. of main application processor.

CoreSight Access

These accesses are typically used to access the CoreSight busses APB, AHB and AXI directly through the DAP bypassing the CPU. For example, this could be used to view physical memory at run-time.

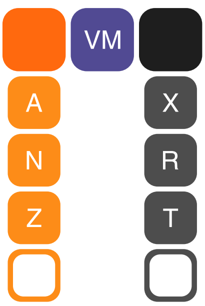


Example combinations:

EAXI	Access memory location via AXI during run-time.
EZAXI	Access secure memory location via AXI during run-time.
DAP	Access debug access port (e.g. core debug registers).

Cache and Virtual Memory Access

These accesses are used to either access the TRACE32 virtual memory (VM:) or to access data and instruction caches directly or to bypass them.



Example combinations:

VM	Access virtual memory using current CPU context.
AVM	Access virtual memory ignoring current CPU context.
NC	Bypass all cache levels during memory access.
ANC	Bypass MMU and all cache levels during memory access.

If you omit access class specifiers in an access class combination, then TRACE32 will make an educated guess to fill in the blanks. The access class is expanded based on:

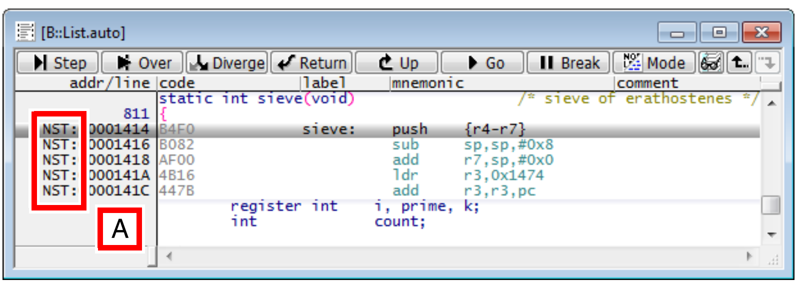
- The current CPU context (architecture specific)
- The used window type (e.g. **Data.dump** window for data or **List.Mix** window for code)
- Symbol information of the loaded application (e.g. combination of code and data)
- Segments that use different instruction sets
- Debugger specific settings (e.g. **SYStem.Option**.*)

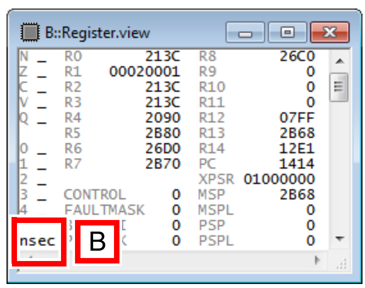
Examples: Memory Access through CPU

Let's assume the CPU is in non-secure supervisor mode, executing 32-bit code.

User input at the command line	Expansion by TRACE32	These access classes are added because...
`List.Mix` (see also illustration below)	**NST**:	**N**: … the CPU is in non-secure mode. **S**: … the CPU is in supervisor mode. **R**: … code is viewed (not data) and the CPU uses Thumb instructions.
`Data.dump A:0x0`	A**NSD**:0x0	**N**: … the CPU is in non-secure mode. **S**: … the CPU is in supervisor mode. **D**: … data is viewed (not code).
`Data.dump Z:0x0`	Z**SD**:0x0	**S**: … the CPU is in supervisor mode. **D**: … data is viewed (not code).
NOTE: 'E' and 'A' are not automatically added because the debugger cannot know if you intended a run-time or physical access.		

Your input, here `List.Mix` at the TRACE32 command line, remains unmodified. TRACE32 performs an access class expansion and visualizes the result in the window you open, here in the **List.Mix** window.





A TRACE32 makes an educated guess to expand your *omitted* access class to "NST".

B Indicates that the CPU is in non-secure mode.

Benchmark counters are on-chip counters that count specific hardware events, e.g., the number of executed instructions. This allows to calculate typical performance metrics like clocks per instruction (CPI).The benchmark counters can be read at run-time.

The following counters are available on Cortex-M:

- **CYC**, Cycle Counter: This counter counts the total number of CPU clock cycles. The counter will be used in conjunction with the CPU clock frequency to calculate the running times.
- **CPI**, This counter counts the total number of cycles per instruction *after* the first cycle. For example: If an instruction takes 5 cycles, the CPI counter will be incremented by 4. The slower this counter increases, the more instructions per cycle are executed.
- **EXC**, Exception Counter: This counter counts the number of cycles spent in interrupt processing specific operations. The counter counts the overhead incurred because of interrupts (like entry sequences which put registers onto the stack, exit sequences which restore registers from the stack, etc.).
- **SLP**, Sleep Counter: This counter counts the number of FCLK cycles the CPU spent sleeping.
- **LSU**, Load and Store Unit Counter: This counter counts the number of cycles spent in load and store instructions *after* the first cycle. For example: If a load instruction takes 4 cycles, the LSU counter will be incremented by 4. The slower this counter increases, the more instructions per cycle are executed.
- **FLD**, Fold Counter: In certain situations, the Cortex-M core is able to spent zero clock cycles for an instruction. Such instructions are called *folded* instructions. The FLD counter counts the number of folded instructions.

TRACE32 PowerView supports enabling and analyzing the BenchMark Counters graphically.

Example:

The following example computes the LSU counter rate (in Events/Second) for different sorting algorithms. The results are displayed graphically for the different counter rates over the time and correlated to the different defined sections.

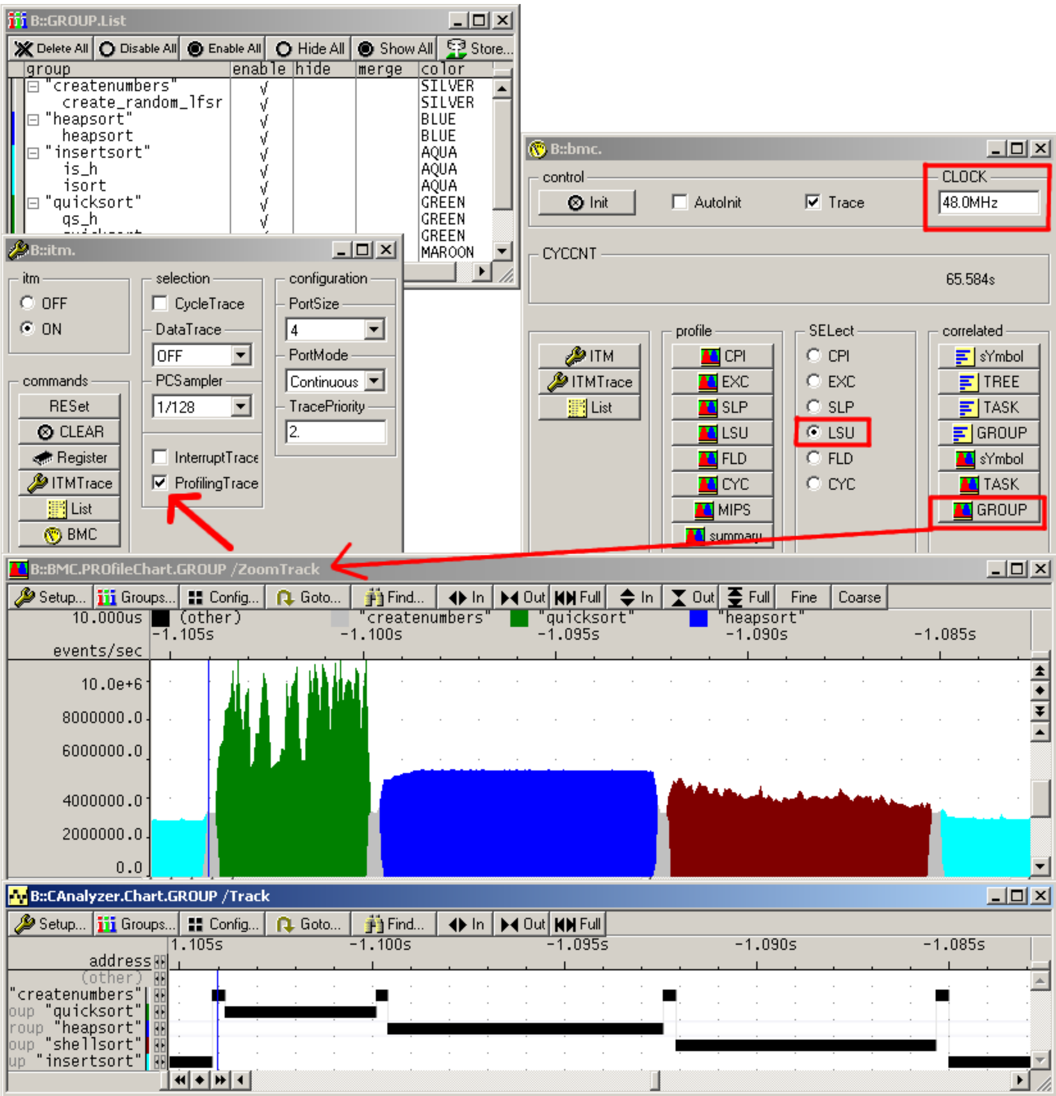
```
; Enable the BMC counters with command
ITM.ProfilingTrace ON

; Provide code clock for cycle counter, e.g. here 48MHz
BMC.CLOCK 48MHz

; Group the program into interesting sections using the GROUP command
GROUP.Create ...

; display the results graphically and track with the trace results
BMC.PROfileChhart.GROUP /ZoomTrack
CAnalyzer.Chart.GROUP /Track
```

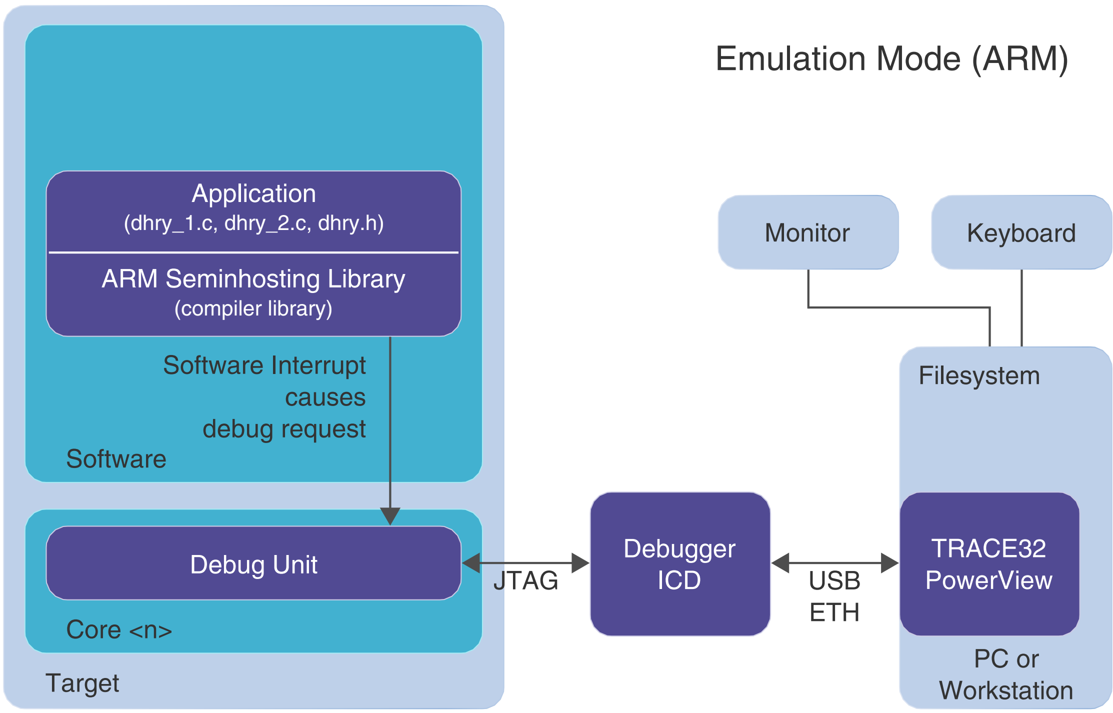
In this example, the *quicksort* algorithm produces the highest rate for the **LSU** counter. This means that the bottleneck for this algorithm is the access to memory where the data is stored; the CPU spends more cycles waiting for memory than in all other algorithms.
This is a *good* sign; it means that the code is very optimized, so that the CPU itself does not have to execute many non-load/store instructions.



Semihosting

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

Normally semihosting is invoked by code within the C library functions of the ARM RealView compiler like printf() and scanf(). The application can also invoke the operations used for keyboard input, screen output, and file I/O directly. The operations are described in the RealView Compilation Tools Developer Guide from ARM in the chapter “Semihosting Operations”.



A semihosting call from the application causes a BKPT exception in the semihosting library function. The immediate BKPT parameter 0xAB is indicating a semihosting request. The type of operation is passed in R0. R1 points to the other parameters. The debugger handles the request while the application is stopped, provides the required communication with the host, and restarts the application.

This mode is enabled by **TERM.METHOD ARMSWI** and by opening a **TERM.GATE** window for the semihosting screen output. The handling of the semihosting requests is only active when the **TERM.GATE** window is existing.

TERM.HEAPINFO defines the system stack and heap location. The C library reads these memory parameters by a SYS_HEAPINFO semihosting call and uses them for initialization.

An code example can be found in ~~/demo/arm/etc/semihosting_arm_emulation.

The Cortex-M does not have a Debug Communication Channel (DCC) like other Cortex cores. Therefore this mode can not be used. Alternatively, to avoid stopping the application, the BufferE method can be used. Then the semihosting requests are processed via a buffer located in the system memory which can be accessed by the debugger without stopping the core. There is an example in ~~/demo/arm/etc/semihosting_trace32_dcc which uses the TRACE32 proprietary semihosting functions. And there is an example in ~~/demo/arm/etc/semihosting_arm_syscalls which allow to use the ARM semihosting library functions with BufferE method.

Virtual Terminal

The command **TERM** opens a terminal window in the debugger which allows to communicate with the program running on the Cortex-M. All data received are displayed in this window and all data inputs to this window are sent to the program running on the Cortex-M.

The **TERM.METHOD** command selects which method is used for the communication.

The Cortex-M does not have a Debug Communication Channel (DCC) as other Cortex cores but even better it's system memory can be accessed by the debugger during run time. Therefore you can e.g. reserve a single memory byte for input data and one for output data somewhere in the system memory. The following command tells the debugger the addresses of the reserved bytes which shall be used for the communication. You can use address values or symbol names as command parameter:

TERM.METHOD SingleE E:*<byte_address_to_debugger>* **E:***<byte_address_from_debugger>*

A data value of 0 in the byte buffer indicates an empty byte buffer. This is the way the handshake works. After data is read a 0 is placed in the buffer to indicate the data is taken and a new byte can be sent.

The TRACE32 ~~/demo/arm/etc/virtual_terminal/memory_based directory contains an example of this method.

Alternatively BufferE method could be used which works quite similar but with a bigger buffer to transfer more than one byte at once.

Runtime Measurement

The command **RunTime** allows run time measurement based on polling the core run status by software. Therefore the result will be about few milliseconds higher than the real value.

If the signal DBGACK on the JTAG connector is available, the measurement will automatically be based on this hardware signal which delivers very exact results.

Trigger

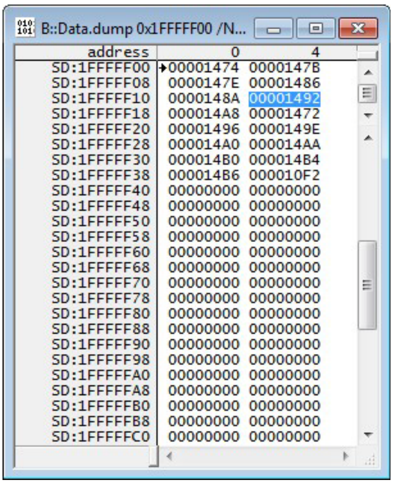
A bidirectional trigger system allows the following two events:

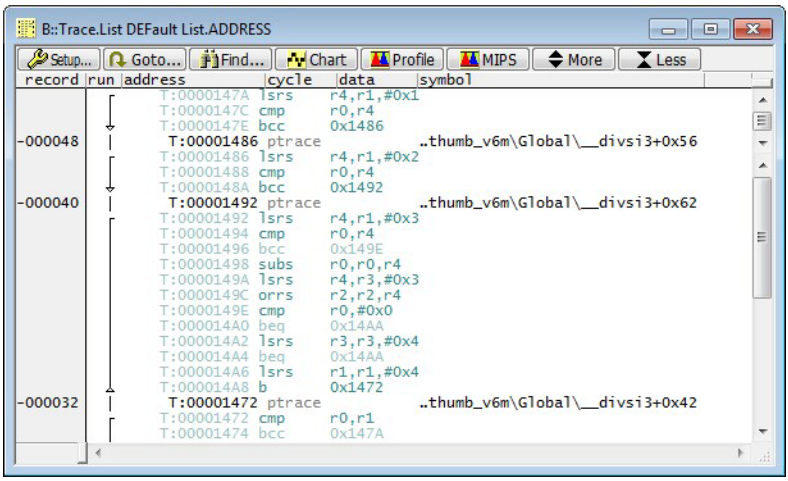
- trigger an external system (e.g. logic analyzer) if the program execution is stopped.
- stop the program execution if an external trigger is asserted.

For more information refer to the **TrBus** command.

Micro Trace Buffer (MTB) for Cortex-M0+

Take-off and landing addresses of all branches are recorded to the MTB. The **Data.dump** screenshot shows the trace row data, the **Trace.List** screenshot shows the instruction execution sequence decoded by TRACE32.





Cortex-M specific Onchip Commands

Onchip.Mode.RAMPRIV SRAM privilege access

Format:	**Onchip.Mode.RAMPRIV** [**ON** | **OFF**]

Enables SRAM privilege access.

Onchip.Mode.SFRWPRIV Special function register write access

Format:	**Onchip.Mode.SFRWPRIV** [**ON** | **OFF**]

Enables privilege write access to the Special Function Register.

Onchip.Mode.TSTARTEN Enable TSTART signal

Format:	**Onchip.Mode.TSTARTEN** [**ON** | **OFF**]

Enables the external control of the trace by the TSTART signal. If set to ON and the TSTART signal is set HIGH, then the trace gets starts recording.

Onchip.Mode.TSTOPEN Enable TSTOP signal

Format:	**Onchip.Mode.TSTOPEN** [**ON** | **OFF**]

Enables the external control of the trace by the TSTOP signal. If set to ON and the TSTOP signal is set HIGH, then the trace stops recording.

Format:	**Onchip.TBADDRESS** *<address>*

Sets up the base address of the trace buffer inside the internal SRAM. The part of the SRAM must not be used by the target application as long as the trace is used.

SYStem.CLOCK — Inform debugger about core clock

Format:	**SYStem.CLOCK** *<frequency>*

Informs the debugger about the core clock frequency. This information is used for analysis functions where the core frequency needs to be known. This command is only available if the debugger is used as front-end for virtual prototyping.

SYStem.CONFIG.state — Display target configuration

Format:	**SYStem.CONFIG.state** [**/**<tab>]
<tab>:	**DebugPort** | **Jtag** | **MultiTap** | **AP** | **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**.
MultiTap	Informs the debugger about the existence and type of a System/Chip Level Test Access Port. The debugger might need to control it in order to reconfigure the JTAG chain or to control power, clock, reset, and security of different chip components. For descriptions of the commands on the **MultiTap** tab, see **MultiTap**.
AP	This tab informs the debugger about an Arm CoreSight Access Port (AP) and about how to control the AP to access chip-internal memory busses (AHB, APB, AXI) or chip-internal JTAG interfaces. For a descriptions of a corresponding commands, refer to **AP**.
COmponents	The **COmponents** tab informs the debugger (a) about the existence and interconnection of on-chip CoreSight debug and trace modules and (b) informs the debugger on which memory bus and at which base address the debugger can find the control registers of the modules. For descriptions of the commands on the **COmponents** tab, see **COmponents**.

SYStem.CONFIG — Configure debugger according to target topology

Format:	**SYStem.CONFIG** *<parameter>* **SYStem.MultiCore** *<parameter>* (deprecated)
<parameter>: **(DebugPort)**	**CJTAGFLAGS** *<flags>* **CJTAGTCA** *<value>* **CONNECTOR** [**MIPI34** | **MIPI20T**] **CORE** *<core> <chip>* **CoreNumber** *<number>* **DEBUGPORT** [**DebugCable0** | **DebugCableA** | **DebugCableB**] **DEBUGPORTTYPE** [**JTAG** | **SWD** | **CJTAG**] **NIDNTTRSTTORST** [**ON** | **OFF**] **NIDNTPSRISINGEDGE** [**ON** | **OFF**] **NIDNTRSTPOLARITY** [**High** | **Low**] **PortSHaRing** [**ON** | **OFF** | **Auto**]

<parameter>: (DebugPort cont.)	**Slave** [**ON** | **OFF**] **SWDP** [**ON** | **OFF**] **SWDPIDLEHIGH** [**ON** | **OFF**] **SWDPTargetSel** *<value>* **DAP2SWDPTargetSel** *<value>* **TriState** [**ON** | **OFF**]
<parameter>: **(JTAG)**	**CHIPDRLENGTH** *<bits>* **CHIPDRPATTERN** [**Standard** | **Alternate** *<pattern>*] **CHIPDRPOST** *<bits>* **CHIPDRPRE** *<bits>* **CHIPIRLENGTH** *<bits>* **CHIPIRPATTERN** [**Standard** | **Alternate** *<pattern>*] **CHIPIRPOST***<bits>* **CHIPIRPRE** *<bits>* **DAP2DRPOST** *<bits>* **DAP2DRPRE** *<bits>* **DAP2IRPOST** *<bits>* **DAP2IRPRE** *<bits>* **DAPDRPOST** *<bits>* **DAPDRPRE** *<bits>* **DAPIRPOST** *<bits>* **DAPIRPRE** *<bits>* **DRPOST** *<bits>* **DRPRE** *<bits>* **ETBDRPOST** *<bits>* **ETBDRPRE** *<bits>* **ETBIRPOST** *<bits>* **ETBIRPRE** *<bits>* **IRPOST***<bits>* **IRPRE** *<bits>* **NEXTDRPOST** *<bits>* **NEXTDRPRE** *<bits>* **NEXTIRPOST***<bits>* **NEXTIRPRE** *<bits>* **RTPDRPOST** *<bits>* **RTPDRPRE** *<bits>* **RTPIRPOST** *<bits>* **RTPIRPRE** *<bits>* **Slave** [**ON** | **OFF**] **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**]

<parameter>: **(MultiTap)**	**CFGCONNECT** *<code>* **DAP2TAP** *<tap>* **DAPTAP** *<tap>* **DEBUGTAP** *<tap>* **ETBTAP** *<tap>* **MULTITAP** [**NONE** | **IcepickA** | **IcepickB** | **IcepickC** | **IcepickD** | **IcepickBB** | **IcepickBC** | **IcepickCC** | **IcepickDD** | **STCLTAP1** | **STCLTAP2** | **STCLTAP3** | **MSMTAP** *<irlength> <irvalue> <drlength> <drvalue>* **JtagSEQuence** *<sub_cmd>*] **NJCR** *<tap>* **RTPTAP** *<tap>* **SLAVETAP** *<tap>*
<parameter>: **(AccessPorts)**	**AHBAPn.Base** *<address>* **AHBAPn.HPROT** [*<value>* | *<name>*] **AHBAPn.Port** *<port>* **AHBAPn.RESet** **AHBAPn.view** **AHBAPn.XCPTRI** *<tri>* **AHBAPn.XtorName** *<name>* **APBAPn.Base** *<address>* **APBAPn.HPROT** [*<value>* | *<name>*] **APBAPn.Port** *<port>* **APBAPn.RESet** **APBAPn.view** **APBAPn.XCPTRI** *<tri>* **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.XCPTRI** *<tri>* **AXIAPn.XtorName** *<name>* **DAP2JTAGPORT** *<port>* **DAPNAME** *<name>* **DAP2NAME** *<name>* **DEBUGAPn.Port** *<port>* **DEBUGAPn.RESet** **DEBUGAPn.view** **DEBUGAPn.XtorName** *<name>*

<parameter>: (AccessPorts cont.)	**JTAGAPn.Base** *<address>* **JTAGAPn.Port** *<port>* **JTAGAPn.CorePort** *<port>* **JTAGAPn.RESet** **JTAGAPn.view** **JTAGAPn.XtorName** *<name>* **MEMORYAPn.HPROT** [*<value>* | *<name>*] **MEMORYAPn.Port** *<port>* **MEMORYAPn.RESet** **MEMORYAPn.view** **MEMORYAPn.XtorName** *<name>*

<parameter>: **(COmponents)**	**AMU.Base** *<address>* **AMU.RESet** **AMU.view** **AHBBRIDGE.Base** *<address>* **AHBBRIDGE.RESet** **AHBBRIDGE.view** **BMC.Base** *<address>* **BMC.RESet** **BMC.view** **BPU.Base** *<address>* **BPU.RESet** **BPU.view** **CMI.Base** *<address>* **CMI.RESet** **CMI.TraceID** *<id>* **COREDEBUG.Base** *<address>* **COREDEBUG.RESet** **COREDEBUG.view** **CTI.Base** *<address>* **CTI.Config** *<interconnection>* **CTI.RESet** **CTI.view** **DRM.Base** *<address>* **DRM.RESet** **DRM.view** **DTM.RESet** **DTM.Type** [**None** | **Generic**] **DTM.view**

<parameter>:
(COmponents cont.)

DWT.Base *<address>*
DWT.RESet
DWT.view

EPM.Base *<address>*
EPM.RESet
EPM.view

ETB2AXI.Base *<address>*
ETB2AXI.RESet
ETB2AXI.view

ETB.ATBSource *<source>*
ETB.Base *<address>*
ETB.Name *<string>*
ETB.NoFlush [**ON** | **OFF**]
ETB.RESet
ETB.Size *<size>*
ETB.STackMode [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULL STOP** | **FULLCTI**]
ETB.view

ETF.ATBSource *<source>*
ETF.Base *<address>*
ETF.Name *<string>*
ETF.NoFlush [**ON** | **OFF**]
ETF.RESet
ETF.Size *<size>*
ETF.STackMode [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULL STOP** | **FULLCTI**]
ETF.view

ETM.Base *<address>*
ETM.RESet
ETM.view

ETR.ATBSource *<source>*
ETR.CATUBase *<address>*
ETR.Base *<address>*
ETR.Name *<string>*
ETR.NoFlush [**ON** | **OFF**]
ETR.RESet
ETR.Size *<size>*
ETR.STackMode [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULL STOP** | **FULLCTI**]
ETR.view

<parameter>:
(COmponents cont.)

ETS.ATBSource *<source>*
ETS.Base *<address>*
ETS.Name *<string>*
ETS.NoFlush [**ON** | **OFF**]
ETS.RESet
ETS.Size *<size>*
ETS.STackMode [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULLSTOP** | **FULLCTI**]
ETS.view

FPB.Base *<address>*
FPB.RESet
FPB.view

FUNNEL.ATBSource *<sourcelist>*
FUNNEL.Base *<address>*
FUNNEL.Name *<string>*
FUNNEL.PROGrammable [**ON** | **OFF**]
FUNNEL.RESet
FUNNEL.view

HSM.Base *<address>*
HSM.RESet
HSM.view

HTM.Base *<address>*
HTM.RESet
HTM.Type [**CoreSight** | **WPT**]
HTM.view

ICE.Base *<address>*
ICE.RESet
ICE.view

ITM.Base *<address>*
ITM.Name *<string>*
ITM.RESet
ITM.view

L2CACHE.Base *<address>*
L2CACHE.RESet
L2CACHE.Type [**NONE** | **Generic** | **L210** | **L220** | **L2C-310** | **AURORA** | **AURORA2**]
L2CACHE.view

MPAM.Base *<address>*
MPAM.RESet
MPAM.view

<parameter>:
(COmponents cont.)

MTB.Base *<address>*
MTB.RESet
MTB.Size *<size>*
MTB.SRAMBase *<address>*
MTB.view

MTBDWT.Base *<address>*
MTBDWT.RESet
MTBDWT.view

OCP.Base *<address>*
OCP.RESet
OCP.TraceID *<id>*
OCP.Type *<type>*

PMI.Base *<address>*
PMI.RESet
PMI.TraceID *<id>*

RAS.Base *<address>*
RAS.RESet
RAS.view

REP.ATBSource *<source>*
REP.Base *<address>*
REB.Name *<string>*
REP.RESet
REP.view

RTP.Base *<address>*
RTP.PerBase *<address>*
RTP.RamBase *<address>*
RTP.RESet
RTP.view

SC.Base *<address>*
SC.RESet
SC.TraceID *<id>*

SDC.Base *<address>*
SDC.RESet

STM.Base *<address>*
STM.Mode [**NONE** | **XTIv2** | **SDTI** | **STP** | **STP64** | **STPv2**]
STM.RESet
STM.Type [**None** | **GenericARM** | **SDTI** | **TI**]

TBR.ATBSource *<source>*
TBR.Base *<address>*
TBR.Name *<string>*
TBR.NoFlush [**ON** | **OFF**]
TBR.RESet

<parameter>: (Components cont.)	**TBR.Size** *<size>* **TBR.STackMode** [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULL STOP** | **FULLCTI**] **TBR.view** **TISCTM.Base** *<address>* **TISCTM.RESet** **TISCTM.view** **TPIU.ATBSource** *<source>* **TPIU.Base** *<address>* **TPIU.Name** *<string>* **TPIU.RESet** **TPIU.Type** [**CoreSight** | **Generic**] **TPIU.view**

<parameter>: **(Deprecated)**	**AHBACCESSPORT** *<port>* **APBACCESSPORT** *<port>* **AXIACCESSPORT** *<port>* **BMCBASE** *<address>* **BYPASS** *<seq>* **COREBASE** *<address>* **COREJTAGPORT** *<port>* **CTIBASE** *<address>* **CTICONFIG** [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**] **DAP2AHBACCESSPORT** *<port>* **DAP2APBACCESSPORT** *<port>* **DAP2AXIACCESSPORT** *<port>* **DAP2COREJTAGPORT** *<port>* **DAP2DEBUGACCESSPORT** *<port>* **DEBUGACCESSPORT** *<port>* **DEBUGBASE** *<address>* **DTMCONFIG** [**ON** | **OFF**] **DTMETBFUNNELPORT** *<port>* **DTMFUNNEL2PORT** *<port>* **DTMFUNNELPORT** *<port>* **DTMTPIUFUNNELPORT** *<port>* **DWTBASE** *<address>* **ETB2AXIBASE** *<address>* **ETBBASE** *<address>* **ETBFUNNELBASE** *<address>* **ETFBASE** *<address>* **ETMBASE** *<address>* **ETMETBFUNNELPORT** *<port>* **ETMFUNNEL2PORT** *<port>* **ETMFUNNELPORT** *<port>* **ETMTPIUFUNNELPORT** *<port>* **FILLDRZERO** [**ON** | **OFF**] **FUNNEL2BASE** *<address>* **FUNNELBASE** *<address>* **HSMBASE** *<address>* **HTMBASE** *<address>* **HTMETBFUNNELPORT** *<port>* **HTMFUNNEL2PORT** *<port>* **HTMFUNNELPORT** *<port>* **HTMTPIUFUNNELPORT** *<port>* **ITMBASE** *<address>* **ITMETBFUNNELPORT** *<port>*

<parameter>: (Deprecated cont.)	**ITMFUNNEL2PORT** *<port>* **ITMFUNNELPORT** *<port>* **ITMTPIUFUNNELPORT** *<port>* **JTAGACCESSPORT** *<port>* **MEMORYACCESSPORT** *<port>* **PERBASE** *<address>* **RAMBASE** *<address>* **RTPBASE** *<address>* **SDTIBASE** *<address>* **STMBASE** *<address>* **STMETBFUNNELPORT***<port>* **STMFUNNEL2PORT***<port>* **STMFUNNELPORT***<port>* **STMTPIUFUNNELPORT***<port>* **TIDRMBASE** *<address>* **TIEPMBASE** *<address>* **TIICEBASE** *<address>* **TIOCPBASE** *<address>* **TIOCPTYPE** *<type>* **TIPMIBASE** *<address>* **TISCBASE** *<address>* **TISTMBASE** *<address>* **TPIUBASE** *<address>* **TPIUFUNNELBASE** *<address>* **view**

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

In many cases, selecting the chip under debug with the **SYStem.CPU** command is sufficient. TRACE32 recognizes the available on-chip debug and trace components and can configure them accordingly.

If the components require configuration using the **SYStem.CONFIG** commands, you must first set the chip under debug using the **SYStem.CPU** command. Then, configure the components with the **SYStem.CONFIG** commands. Finally, start the debug session e.g. with the **SYStem.Up** command.

Syntax Remarks

The commands are not case-sensitive. Capital letters indicate how the command can be abbreviated.
Example: “SYStem.CONFIG.DWT.Base 0x1000” -> “SYS.CONFIG.DWT.B 0x1000”

The dots after “SYStem.CONFIG” can alternatively be replaced with a space.
Example: “SYStem.CONFIG.DWT.Base 0x1000” or “SYStem.CONFIG DWT Base 0x1000”.

More Information on the Deprecated Commands

General information on deprecated commands and command parameters can be found here.

The table **Mapping Deprecated to New Commands** provides a mapping of the deprecated command parameters to the new command parameters.

A detailed description of the deprecated command parameters can be found in **“<parameters> which are “Deprecated””**.

CJTAGFLAGS *<flags>*	Activates bug fixes for "cJTAG" implementations. Bit 0: Disable scanning of cJTAG ID. Bit 1: Target has no "keeper". Bit 2: Inverted meaning of SREDGE register. Bit 3: Old command opcodes. Bit 4: Unlock cJTAG via APFC register. Default: 0.
CJTAGTCA *<value>*	Selects the TCA (TAP Controller Address) to address a device in a cJTAG Star-2 configuration. The Star-2 configuration requires a unique TCA for each device on the debug port.
CONNECTOR [**MIPI34** | **MIPI20T**]	Specifies the connector "MIPI34" or "MIPI20T" on the target. This is mainly needed in order to notify the trace pin location. Default: MIPI34 if CombiProbe is used, MIPI20T if µTrace (MicroTrace) is used.
CORE *<core>* *<chip>*	The command helps to identify debug and trace resources which are commonly used by different cores. The command might be required in a multicore environment if you use multiple debugger instances (multiple TRACE32 PowerView GUIs) to simultaneously debug different cores on the same target system. Because of the default setting of this command debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=1 *<chip>*=2 ... each debugger instance assumes that all notified debug and trace resources can exclusively be used. But some target systems have shared resources for different cores, for example a common trace port. The default setting causes that each debugger instance controls the same trace port. Sometimes it does not hurt if such a module is controlled twice. But sometimes it is a must to tell the debugger that these cores share resources on the same *<chip>*. Whereby the "chip" does not need to be identical with the device on your target board: debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=2 *<chip>*=1

CORE *<core> <chip>*

(cont.)

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

Default:
<core> depends on CPU selection, usually 1.
<chip> If you start multiple debugger instances with **TargetSystem.NewInstance**, you get ascending values (1, 2, 3,...).

CoreNumber *<number>*

Number of cores to be considered in an SMP (symmetric multiprocessing) debug session. There are core types like ARM11MPCore, CortexA5MPCore, CortexA9MPCore and Scorpion 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** | **DebugCableA** | **DebugCableB**]

It specifies which probe cable shall be used e.g. "DebugCableA" or "DebugCableB". At the moment only the CombiProbe allows to connect more than one probe cable.

Default: depends on detection.

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

It specifies the used debug port type "JTAG", "SWD", "CJTAG", "CJTAG-SWD". It assumes the selected type is supported by the target.

Default: JTAG.

What is NIDnT?

NIDnT is an acronym for "Narrow Interface for Debug and Test". NIDnT is a standard from the MIPI Alliance, which defines how to reuse the pins of an existing interface (like for example a microSD card interface) as a debug and test interface.

To support the NIDnT standard in different implementations, TRACE32 has several special options:

NIDNTPSRISINGEDGE [**ON** | **OFF**]	Send data on rising edge for NIDnT PS switching. NIDnT specifies how to switch, for example, the microSD card interface to a debug interface by sending in a special bit sequence via two pins of the microSD card. TRACE32 will send the bits of the sequence incident to the falling edge of the clock, because TRACE32 expects that the target samples the bits on the rising edge of the clock. Some targets will sample the bits on the falling edge of the clock instead. To support such targets, you can configure TRACE32 to send bits on the rising edge of the clock by using SYStem.CONFIG NIDNTPSRISINGEDGE ON **NOTE**: Only enable this option right before you send the NIDnT switching bit sequence. Make sure to DISABLE this option, before you try to connect to the target system with for example **SYStem.Up**.
NIDNTRSTPOLARITY [**High** | **Low**]	Usually TRACE32 requires that the system reset line of a target system is low active and has a pull-up on the target system. When connecting via NIDnT to a target system, the reset line might be a high-active signal. To configure TRACE32 to use a high-active reset signal, use SYStem.CONFIG NIDNTRSTPOLARITY High This option must be used together with SYStem.CONFIG NIDNTTRSTTORST ON because you also have to use the TRST signal of an Arm debug cable as reset signal for NIDnT in this case.
NIDNTTRSTTORST [**ON** | **OFF**]	Usually TRACE32 requires that the system reset line of a target system is low active and has a pull-up on the target system. This is how the system reset line is usually implemented on regular Arm-based targets. When connecting via NIDnT (e.g. a microSD card slot) to the target system, the reset line might not include a pull-up on the target system. To circumvent problems, TRACE32 allows to drive the target reset line via the TRST signal of an Arm debug cable. Enable this option if you want to use the TRST signal of an Arm debug cable as reset signal for a NIDnT.

PortSHaRing [**ON** | **OFF** | **Auto**]	Configure if the debug port is shared with another tool, e.g. an ETAS ETK. **OFF**: Default. Communicate with the target without sending requests. **ON**: Request for access to the debug port and wait until the access is granted before communicating with the target. **Auto**: Automatically detect a connected tool on next **SYStem.Mode Up**, **SYStem.Mode Attach** or **SYStem.Mode Go**. If a tool is detected switch to mode **ON** else switch to mode **OFF**. The current setting can be obtained by the **PORTSHARING()** function, immediate detection can be performed using **SYStem.DETECT PortSHaRing**.
Slave [**ON** | **OFF**]	If several debugger instances 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 ON**. Default: OFF for the first debugger instance. Default: ON for all further debugger instances you open with TargetSystem.NewInstance.
SWDP [**ON** | **OFF**]	With this command you can change from the normal JTAG interface to the serial wire debug mode. SWDP (Serial Wire Debug Port) uses just two signals instead of five. It is required that the target and the debugger hard- and software supports this interface. Default: OFF.
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).

DAP2SWDPTargetSel *<value>*	Device address of the second CoreSight DAP (DAP2) in case of a multidrop serial wire debug port (SWD). Default: none set (any address accepted).
TriState [**ON** | **OFF**]	TriState has to be used if several debug cables are connected to a common JTAG port. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: • nTRST must have a pull-up resistor on the target. • TCK can have a pull-up or pull-down resistor. • Other trigger inputs need to be kept in inactive state. Default: OFF.

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

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

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

NOTE:	There are rarely implemented DAP (Debug Access Port) TAPs, having an 8-bit wide instruction register (IR) instead of 4-bit. They can be identified with the **SYStem.DETECT.DaisyChain** command. Their IDCODE is 0x?ba03477 or 0x?ba07477. They require you to set (or add) SYStem.CONFIG DAPIRPOST 4.

CHIPDRLENGTH *<bits>*	Number of Data Register (DR) bits which needs to get a certain BYPASS pattern.
CHIPDRPATTERN [**Standard** | **Alternate** *<pattern>*]	Data Register (DR) pattern which shall be used for BYPASS instead of the standard (1...1) pattern.
CHIPIRLENGTH *<bits>*	Number of Instruction Register (IR) bits which needs to get a certain BYPASS pattern.
CHIPIRPATTERN [**Standard** | **Alternate** *<pattern>*]	Instruction Register (IR) pattern which shall be used for BYPASS instead of the standard pattern.
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 for the first debugger instance. Default: ON for all further debugger instances you open with TargetSystem.NewInstance.

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.

TAP types:

Core TAP providing access to the debug register of the core you intend to debug.
-> DRPOST, DRPRE, IRPOST, IRPRE.

DAP (Debug Access Port) TAP providing access to the debug register of the core you intend to debug. It might be needed additionally to a Core TAP if the DAP is only used to access memory and not to access the core debug register.
-> DAPDRPOST, DAPDRPRE, DAPIRPOST, DAPIRPRE.

DAP2 (Debug Access Port) TAP in case you need to access a second DAP to reach other memory locations.
-> DAP2DRPOST, DAP2DRPRE, DAP2IRPOST, DAP2IRPRE.

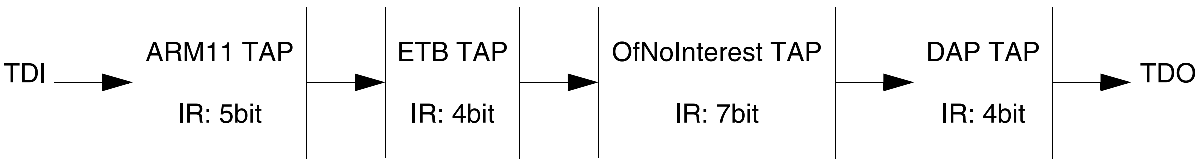
ETB (Embedded Trace Buffer) TAP if the ETB has its own TAP to access its control register (typical with Arm11 cores).
-> ETBDRPOST, ETBDRPRE, ETBIRPOST, ETBIRPRE.

NEXT: If a memory access changes the JTAG chain and the core TAP position then you can specify the new values with the NEXT... parameter. After the access for example the parameter NEXTIRPRE will replace the IRPRE value and NEXTIRPRE becomes 0. Available only on ARM11 debugger.
-> NEXTDRPOST, NEXTDRPRE, NEXTIRPOST, NEXTIRPRE.

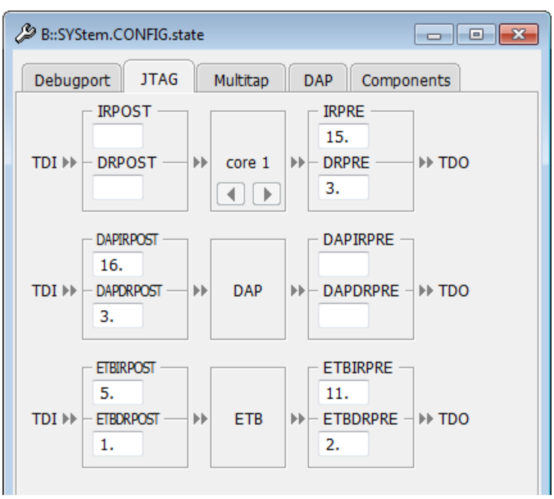
RTP (RAM Trace Port) TAP if the RTP has its own TAP to access its control register.
-> RTPDRPOST, RTPDRPRE, RTPIRPOST, RTPIRPRE.

CHIP: Definition of a TAP or TAP sequence in a scan chain that needs a different Instruction Register (IR) and Data Register (DR) pattern than the default BYPASS (1...1) pattern.
-> CHIPDRPOST, CHIPDRPRE, CHIPIRPOST, CHIPIRPRE.

Example:




```
SYStem.CONFIG IRPRE 15.
SYStem.CONFIG DRPRE  3.
SYStem.CONFIG DAPIRPOST 16.
SYStem.CONFIG DAPDRPOST  3.
SYStem.CONFIG ETBIRPOST  5.
SYStem.CONFIG ETBDRPOST  1.
SYStem.CONFIG ETBIRPRE  11.
SYStem.CONFIG ETBDRPRE   2.
```

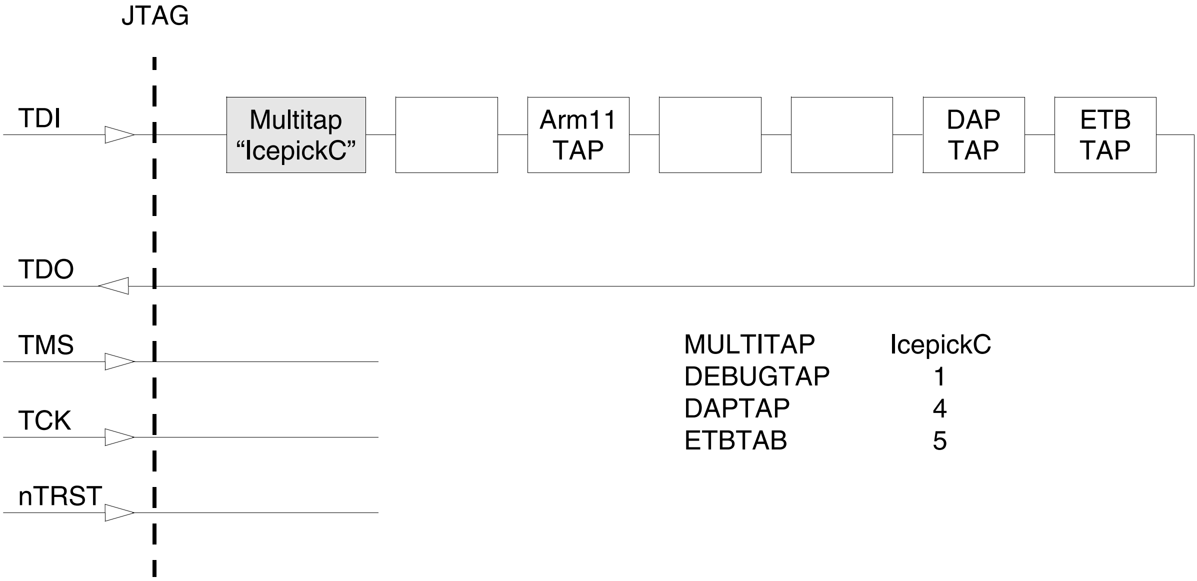


<parameters> describing a system level TAP “MultiTap”

A “Multitap” is a system level or chip level test access port (TAP) in a JTAG scan chain. It can for example provide functions to re-configure the JTAG chain or view and control power, clock, reset and security of different chip components.

At the moment the debugger supports three types and its different versions:
Icepickx, STCLTAPx, MSMTAP:

Example:



CFGCONNECT *<code>*	The *<code>* is a hexadecimal number which defines the JTAG scan chain configuration. You need the chip documentation to figure out the suitable code. In most cases the chip specific default value can be used for the debug session. Used if MULTITAP=STCLTAPx.
DAPTAP *<tap>*	Specifies the TAP number which needs to be activated to get the DAP TAP in the JTAG chain. Used if MULTITAP=Icepickx.
DAP2TAP *<tap>*	Specifies the TAP number which needs to be activated to get a 2nd DAP TAP in the JTAG chain. Used if MULTITAP=Icepickx.

DEBUGTAP *<tap>*	Specifies the TAP number which needs to be activated to get the core TAP in the JTAG chain. E.g. ARM11 TAP if you intend to debug an ARM11. Used if MULTITAP=Icepickx.
ETBTAP *<tap>*	Specifies the TAP number which needs to be activated to get the ETB TAP in the JTAG chain. Used if MULTITAP=Icepickx. ETB = Embedded Trace Buffer.
MULTITAP [**NONE** | **IcepickA** | **IcepickB** | **IcepickC** | **IcepickD** | **IcepickM** | **IcepickBB** | **IcepickBC** | **IcepickCC** | **IcepickDD** | **STCLTAP1** | **STCLTAP2** | **STCLTAP3** | **MSMTAP** *<irlength> <irvalue> <drlength> <drvalue>* **JtagSEQuence** *<sub_cmd>*]	Selects the type and version of the MULTITAP. In case of MSMTAP you need to add parameters which specify which IR pattern and DR pattern needed to be shifted by the debugger to initialize the MSMTAP. Please note some of these parameters need a decimal input (dot at the end). IcepickXY means that there is an Icepick version “X” which includes a subsystem with an Icepick of version “Y”. For a description of the **JtagSEQuence** subcommands, see **SYStem.CONFIG.MULTITAP JtagSEQuence**.
NJCR *<tap>*	Number of a Non-JTAG Control Register (NJCR) which shall be used by the debugger. Used if MULTITAP=Icepickx.
RTPTAP *<tap>*	Specifies the TAP number which needs to be activated to get the RTP TAP in the JTAG chain. Used if MULTITAP=Icepickx. RTP = RAM Trace Port.
SLAVETAP *<tap>*	Specifies the TAP number to get the Icepick of the sub-system in the JTAG scan chain. Used if MULTITAP=IcepickXY (two Icepicks).

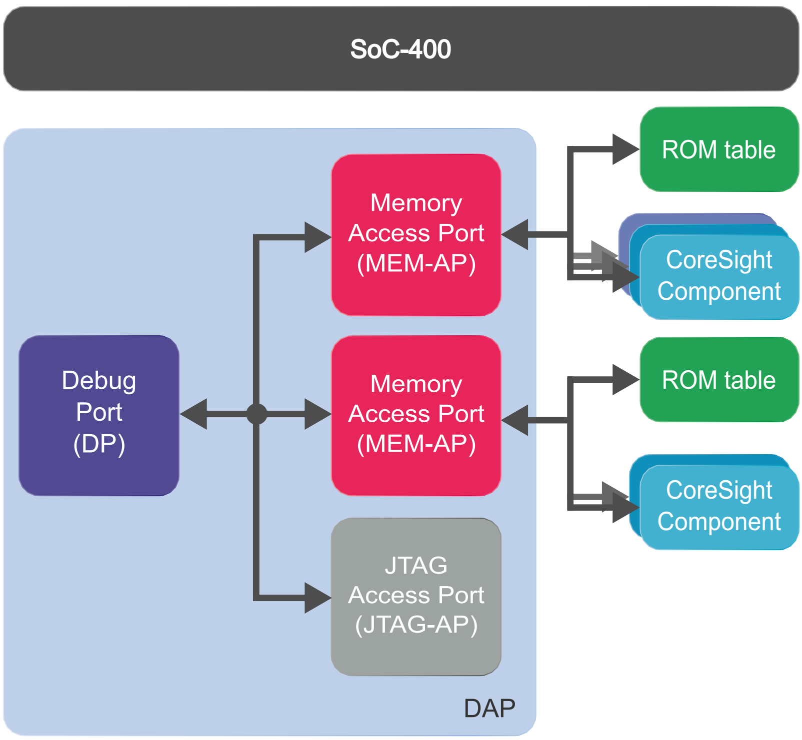
An Access Port (AP) is a CoreSight module from Arm which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

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).

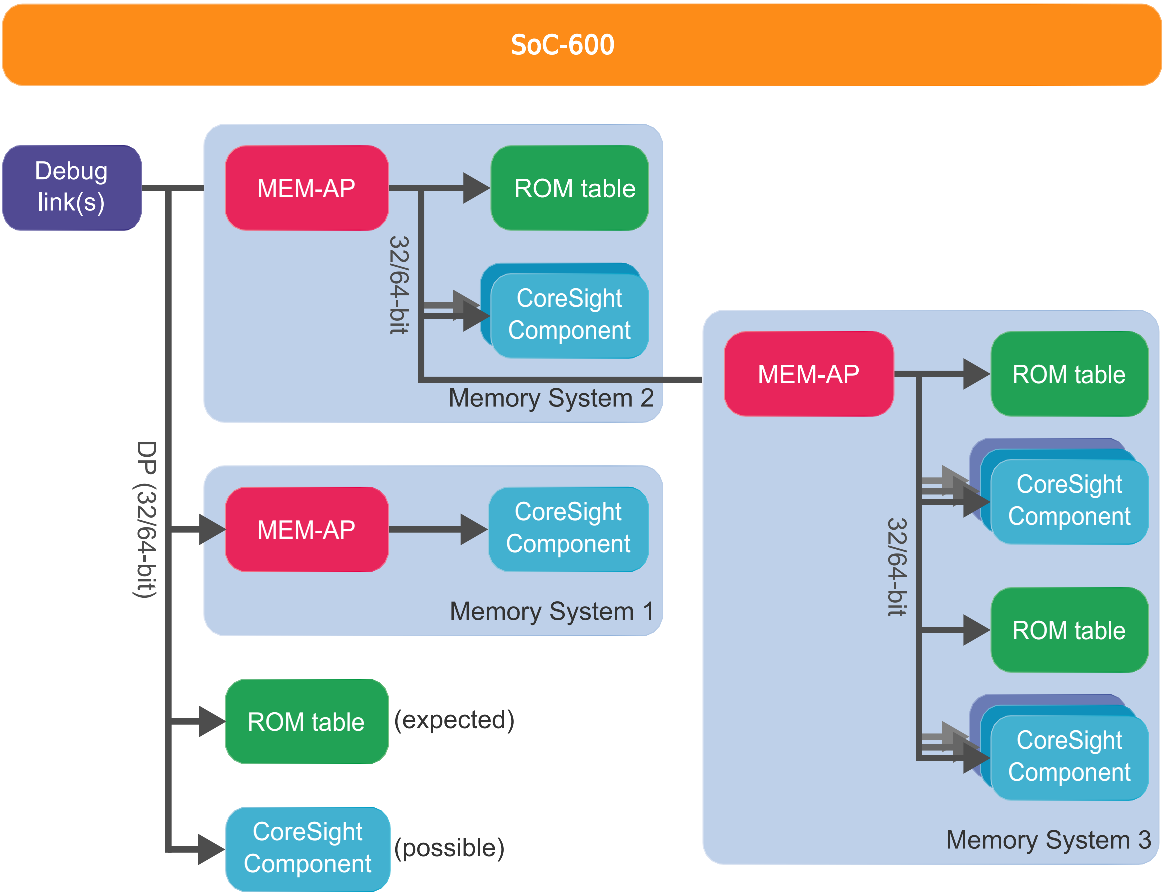
2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number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

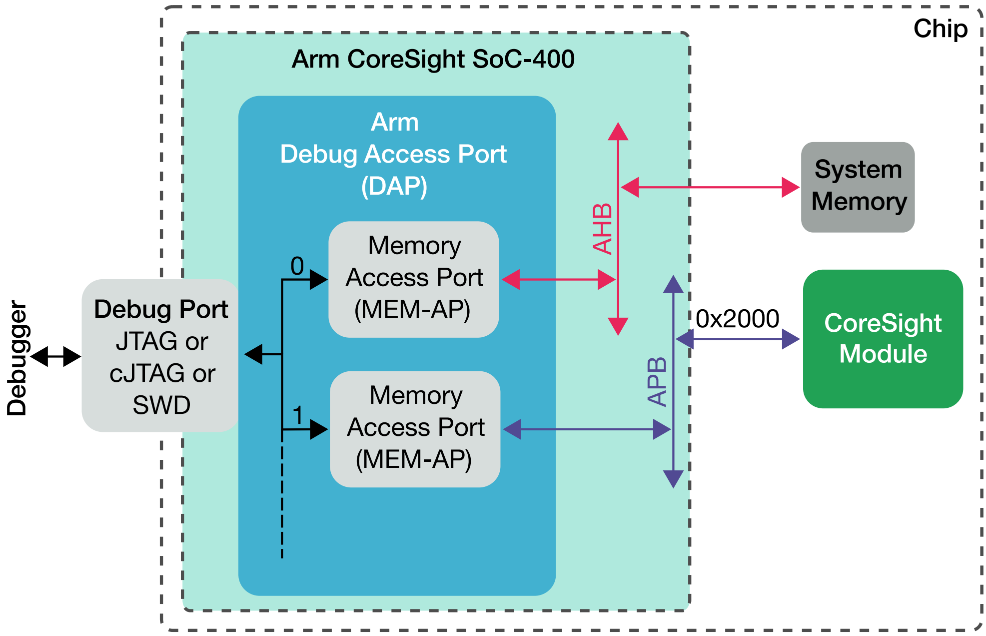
Example 1: SoC-400



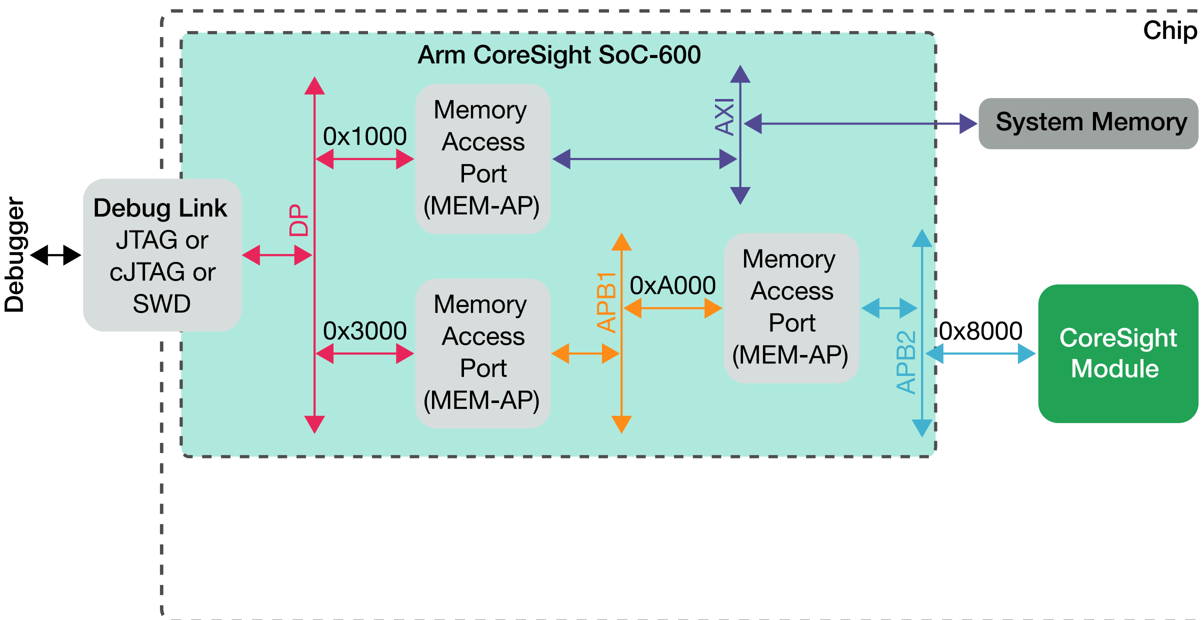
Example 2: SoC-600



Configuration examples for memory access ports and a CoreSight component



TRACE32 configuration:
SYStem.CONFIG AHBAP1.Port 0.
SYStem.CONFIG APBAP1.Port 1.
SYStem.CONFIG <*module*>.Base APB:0x2000



TRACE32 configuration:
SYStem.CONFIG AXIAP1.Base DP:0x1000
SYStem.CONFIG APBAP1.Base DP:0x3000
SYStem.CONFIG APBAP2.Base APB1:0xA000
SYStem.CONFIG <*module*>.Base APB2:0x8000

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>* **AHBNAME** *<name>* (deprecated) **DAP2AHBNAME** *<name>* (deprecated)	AHB bus transactor name that shall be used for “AHBn:” access class.
APBAPn.XtorName *<name>* **APBNAME** *<name>* (deprecated) **DAP2APBNAME** *<name>* (deprecated)	APB bus transactor name that shall be used for “APBn:” access class.
AXIAPn.XtorName *<name>* **AXINAME** *<name>* (deprecated) **DAP2AXINAME** *<name>* (deprecated)	AXI bus transactor name that shall be used for “AXIn:” access class.
DEBUGAPn.XtorName *<name>* **DEBUGBUSNAME** *<name>* (deprecated) **DAP2DEBUGBUSNAME** *<name>* (deprecated)	APB bus transactor name identifying the bus where the debug register can be found. Used for “DAP:” access class.
MEMORYAPn.XtorName *<name>* **MEMORYBUSNAME** *<name>* (deprecated) **DAP2MEMORYBUSNAME** *<name>* (deprecated)	AHB bus transactor name identifying the bus where system memory can be accessed even during runtime. Used for “E:” access class while running, assuming “**SYStem.MemAccess DAP**”.

DAPNAME *<name>*	DAP transactor name that shall be used for DAP access ports.
DAP2NAME *<name>*	DAP transactor name that shall be used for DAP access ports of 2nd order.

... .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>* **AHBACCESSPORT** *<port>* (deprecated) **DAP2AHBACCESSPORT** *<port>* (deprecated)	Access Port Number (0-255) of a SoC-400 system which shall be used for “AHBn:” access class. Default: *<port>*=0.
APBAPn.Port *<port>* **APBACCESSPORT** *<port>* (deprecated) **DAP2APBACCESSPORT** *<port>* (deprecated)	Access Port Number (0-255) of a SoC-400 system which shall be used for “APBn:” access class. Default: *<port>*=1.
AXIAPn.Port *<port>* **AXIACCESSPORT** *<port>* (deprecated) **DAP2AXIACCESSPORT** *<port>* (deprecated)	Access Port Number (0-255) of a SoC-400 system which shall be used for “AXIn:” access class. Default: port not available.
DAP2JTAGPORT *<port>*	JTAG-AP port number (0-7) for an (other) DAP which is connected to a JTAG-AP.
DEBUGAPn.Port *<port>* **DEBUGACCESSPORT** *<port>* (deprecated) **DAP2DEBUGACCESS-PORT** *<port>* (deprecated)	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>* **COREJTAGPORT** *<port>* (deprecated) **DAP2COREJTAGPORT** *<port>* (deprecated)	JTAG-AP port number (0-7) connected to the core which shall be debugged.
JTAGAPn.Port *<port>* **JTAGACCESSPORT** *<port>* (deprecated)	Access port number (0-255) of a SoC-400 system of the JTAG Access Port.
MEMORYAPn.Port *<port>* **MEMORYACCESSPORT** *<port>* (deprecated) **DAP2MEMORYACCESS-PORT** *<port>* (deprecated)	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, assuming “**SYStem.MemAccess DAP**”. 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.

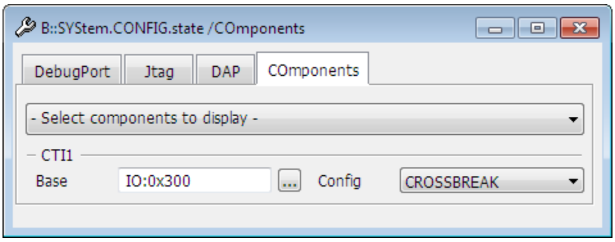
The following commands are used with the XCP backend to configure access to target resources via the XCP slave. If the value is not set, the debugger will fall back to a method that might have less performance.

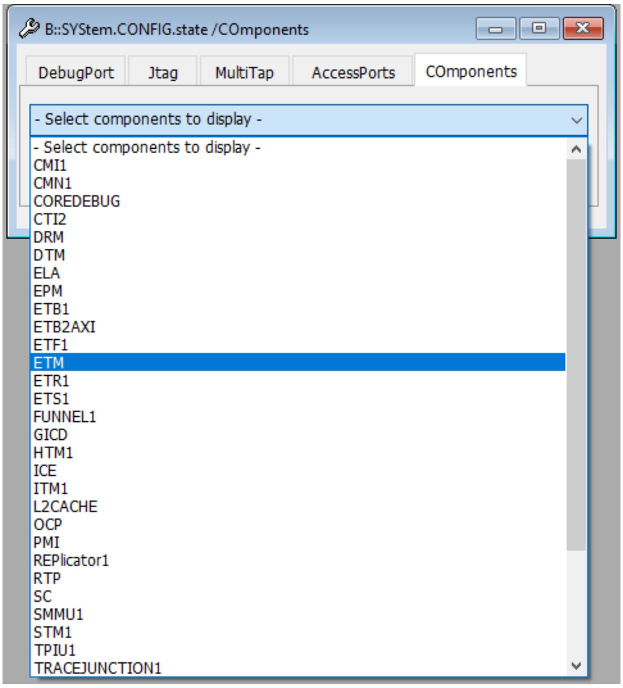
Normally, these values can be set automatically using **SYStem.DETECT.XCPTRI**. For Details, see **“Target Resources”** in XCP Debug Back-End, page 7 (backend_xcp.pdf)

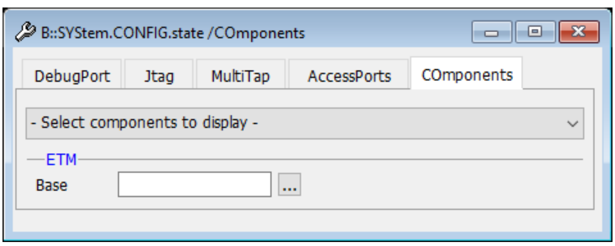
AHBAPn.XCPTRI *<tri>*	Configures the debugger to use the target resource identifier <tri> (0 to 255) for AHB accesses. Default: not set.
APBAPn.XCPTRI *<tri>*	Configures the debugger to use the target resource identifier <tri> (0 to 255) for APB accesses. Default: not set.
AXIAPn.XCPTRI *<tri>*	Configures the debugger to use the target resource identifier <tri> (0 to 255) for AXI accesses. Default: not set.

<parameters> describing debug and trace “Components”

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.





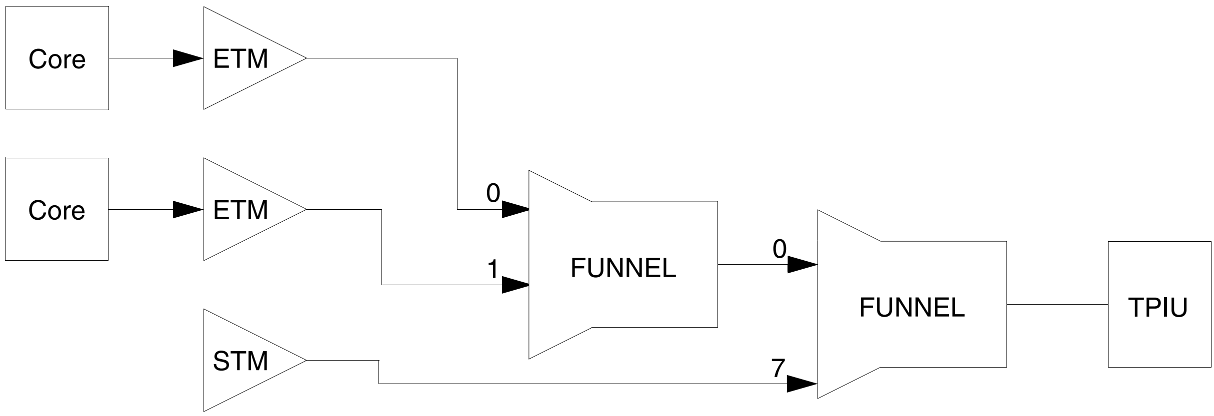


Each configuration can be done by a command in a script file as well. Then you do not need to enter everything again on the next debug session. If you press the button with the three dots you get the corresponding command in the command line where you can view and maybe copy it into a script file.

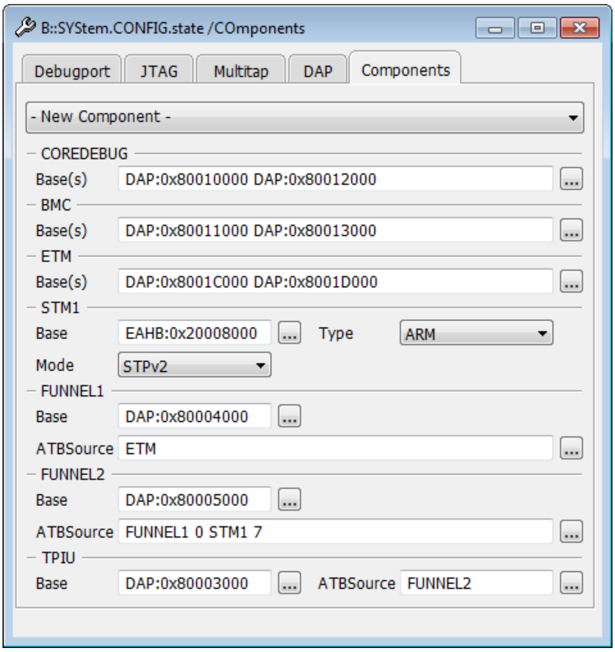
You can have several of the following components: CMI, ETB, ETF, ETR, FUNNEL, STM.
Example: FUNNEL1, FUNNEL2, FUNNEL3,...

The *<address>* parameter can be just an address (e.g. 0x80001000) or you can prepend the access class (e.g. AHB:0x80001000). Without an access class, it gets the command-specific default access class, which in most cases is “EDAP:”. For a configuration example using access classes, see **Configuration examples for memory access ports and a CoreSight component**.

Example:




```
SYStem.CONFIG.COREDEBUG.Base 0x80010000 0x80012000
SYStem.CONFIG.BMC.Base 0x80011000 0x80013000
SYStem.CONFIG.ETM.Base 0x8001c000 0x8001d000
SYStem.CONFIG.STM1.Base EAHB:0x20008000
SYStem.CONFIG.STM1.Type ARM
SYStem.CONFIG.STM1.Mode STPv2
SYStem.CONFIG.FUNNEL1.Base 0x80004000
SYStem.CONFIG.FUNNEL2.Base 0x80005000
SYStem.CONFIG.TPIU.Base 0x80003000
SYStem.CONFIG.FUNNEL1.ATBSource ETM.0 0 ETM.1 1
SYStem.CONFIG.FUNNEL2.ATBSource FUNNEL1 0 STM1 7
SYStem.CONFIG.TPIU.ATBSource FUNNEL2
```



… **.ATBSource** *<source>*

Specify for components collecting trace information from where the trace data are coming from. This way you inform the debugger about the interconnection of different trace components on a common trace bus.

You need to specify the "... .Base *<address>*" or other attributes that define the amount of existing peripheral modules before you can describe the interconnection by "... .ATBSource *<source>*".

A CoreSight trace FUNNEL has eight input ports (port 0-7) to combine the data of various trace sources to a common trace stream. Therefore you can enter instead of a single source a list of sources and input port numbers.

Example:
SYStem.CONFIG FUNNEL.ATBSource ETM 0 HTM 1 STM 7

Meaning: The funnel gets trace data from ETM on port 0, from HTM on port 1 and from STM on port 7.

In an SMP (Symmetric MultiProcessing) debug session where you used a list of base addresses to specify one component per core you need to indicate which component in the list is meant:

Example: Four cores with ETM modules.
SYStem.CONFIG ETM.Base 0x1000 0x2000 0x3000 0x4000
SYStem.CONFIG FUNNEL1.ATBSource ETM.0 0 ETM.1 1 ETM.2 2 ETM.3 3
"...2" of "ETM.2" indicates it is the third ETM module which has the base address 0x3000. The indices of a list are 0, 1, 2, 3,...
If the numbering is accelerating, starting from 0, without gaps, like the example above then you can shorten it to
SYStem.CONFIG FUNNEL1.ATBSource ETM

Example: Four cores, each having an ETM module and an ETB module.
SYStem.CONFIG ETM.Base 0x1000 0x2000 0x3000 0x4000
SYStem.CONFIG ETB.Base 0x5000 0x6000 0x7000 0x8000
SYStem.CONFIG ETB.ATBSource ETM.2 2
The third "ETM.2" module is connected to the third ETB. The last "2" in the command above is the index for the ETB. It is not a port number which exists only for FUNNELs.

For a list of possible components including a short description see **Components and Available Commands**.

… **.BASE** *<address>*

This command informs the debugger of the start address for the component's register block, thereby notifying it of the component's existence. An on-chip debug and trace component typically includes a control register block that the debugger must access to control the component.

Example: SYStem.CONFIG ETM.Base APB:0x8011c000

Meaning: The control register block of the Embedded Trace Macrocell (ETM) starts at address 0x8011c000 and is accessible via APB bus.

In an SMP (Symmetric MultiProcessing) debug session you can enter for the components BMC, COREBEBUG, CTI, ETB, ETF, ETM, ETR a list of base addresses to specify one component per core.

Example assuming four cores: SYStem.CONFIG COREDEBUG.Base 0x80001000 0x80003000 0x80005000 0x80007000

For a list of possible components including a short description see **Components and Available Commands**.

... **.Name**	The name is a freely configurable identifier to describe how many instances exists in a target systems chip. TRACE32 PowerView GUI shares with other opened PowerView GUIs settings and the state of components identified by the same name and component type. Components using different names are not shared. Other attributes as the address or the type are used when no name is configured.
	Example 1: **Shared None-Programmable Funnel:** PowerView1: SYStem.CONFIG.FUNNEL.PROGramable OFF SYStem.CONFIG.FUNNEL.Name "shared-funnel-1" PowerView2: SYStem.CONFIG.FUNNEL.PROGramable OFF SYStem.CONFIG.FUNNEL.Name "shared-funnel-1" SYStem.CONFIG.Core 2. 1. ; merge configuration to describe a target system with one chip containing a single none-programmable FUNNEL. **Example 2: Cluster ETFs**: 1. Configures the ETF base address and access for each core SYStem.CONFIG.ETF.Base DAP:0x80001000 \\ APB:0x80001000 DAP:0x80001000 APB:0x80001000 2. Tells the system the core 1 and 3 share cluster-etf-1 and core 2 and 4 share cluster-etf-2 despite using the same address for all ETFs SYStem.CONFIG.ETF.Name "cluster-etf-1" "cluster-etf-2" \\ "cluster-etf-1" "cluster-etf-2"
... **.NoFlush** [**ON** | **OFF**]	Deactivates a component flush request at the end of the trace recording. This is a workaround for a bug on a certain chip. You will loose trace data at the end of the recording. Don't use it if not needed. Default: OFF.
… **.RESet**	Undo the configuration for this component. This does not cause a physical reset for the component on the chip. For a list of possible components including a short description see **Components and Available Commands**.
… **.Size** *<size>*	Specifies the size of the component. The component size can normally be read out by the debugger. Therefore this command is only needed if this can not be done for any reason.

… **.STackMode** [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULLSTOP** | **FULLCTI**]	Specifies the which method is used to implement the Stack mode of the on-chip trace. **NotAvailable**: stack mode is not available for this on-chip trace. **TRGETM**: the trigger delay counter of the onchip-trace is used. It starts by a trigger signal that must be provided by a trace source. Usually those events are routed through one or more CTIs to the on-chip trace. **FULLTIDRM**: trigger mechanism for TI devices. **NOTSET**: the method is derived by other GUIs or hardware. detection. **FULLSTOP**: on-chip trace stack mode by implementation. **FULLCTI**: on-chip trace provides a trigger signal that is routed back to on-chip trace over a CTI.
… **.view**	Opens a window showing the current configuration of the component. For a list of possible components including a short description see **Components and Available Commands**.
… **.TraceID** *<id>*	Identifies from which component the trace packet is coming from. Components which produce trace information (trace sources) for a common trace stream have a selectable ".TraceID *<id>*". If you miss this SYStem.CONFIG command for a certain trace source (e.g. ETM) then there is a dedicated command group for this component where you can select the ID (ETM.TraceID *<id>*). The default setting is typically fine because the debugger uses different default trace IDs for different components. For a list of possible components including a short description see **Components and Available Commands**.

CTI.Config *<type>*	Informs about the interconnection of the core Cross Trigger Interfaces (CTI). Certain ways of interconnection are common and these are supported by the debugger e.g. to cause a synchronous halt of multiple cores. NONE: The CTI is not used by the debugger. ARMV1: This mode is used for Arm7/9/11 cores which support synchronous halt, only. ARMPostInit: Like ARMV1 but the CTI connection differs from the Arm recommendation. OMAP3: This mode is not yet used. TMS570: Used for a certain CTI connection used on a TMS570 derivative. CortexV1: The CTI will be configured for synchronous start and stop via CTI. It assumes the connection of DBGRQ, DBGACK, DBGRESTART signals to CTI are done as recommended by Arm. The CTIBASE must be notified. “CortexV1” is the default value if a Cortex-A/R core is selected and the CTIBASE is notified. QV1: This mode is not yet used. CTICH01: Channel 0 and 1 of the CTM are used to distribute start/stop events from and to the CTIs. Armv8/Armv9 only. CTICH23: Channel 2 and 3 of the CTM are used to distribute start/stop events from and to the CTIs. Armv8/Armv9 only. ARMV8V3: Channel 0, 1 and 2 of the CTM are used to distribute start/stop events. Implemented on request. Armv8/Armv9 only.
DTM.Type [**None** | **Generic**]	Informs the debugger that a customer proprietary Data Trace Message (DTM) module is available. This causes the debugger to consider this source when capturing common trace data. Trace data from this module will be recorded and can be accessed later but the unknown DTM module itself will not be controlled by the debugger.
ETR.CATUBase *<address>*	Base address of the CoreSight Address Translation Unit (CATU).
FUNNEL.Name *<string>*	It is possible that different funnels have the same address for their control register block. This assumes they are on different buses and for different cores. In this case it is needed to give the funnel different names to differentiate them.

FUNNEL.PROGrammable [**ON** | **OFF**]	Default is ON. If set to ON the peripheral is controlled by TRACE32 in order to route ATB trace data through the ATB bus network. If PROGrammable is configured to value OFF then TRACE32 will not access the FUNNEL registers and the base address doesn't need to be configured. This can be useful for FUNNELs that don't have registers or when those registers are read-only. TRACE32 need still be aware of the connected ATB trace sources and sink in order to know the ATB topology. To build a complete topology across multiple instances of PowerView the property Name should be set at all instances to a chip wide unique identifier.
HTM.Type [**CoreSight** | **WPT**]	Selects the type of the AMBA AHB Trace Macrocell (HTM). CoreSight is the type as described in the Arm CoreSight manuals. WPT is a NXP proprietary trace module.
L2CACHE.Type [**NONE** | **Generic** | **L210** | **L220** | **L2C-310** | **AURORA** | **AURORA2**]	Selects the type of the level2 cache controller. L210, L220, L2C-310 are controller types provided by Arm. AURORAx are Marvell types. The ‘Generic’ type does not need certain treatment by the debugger.
MTB.SRAMBase	Sets an alternate address for SRAMBASE. This is needed for some chips where the value of the SRAMBASE register is wrong.
OCP.Type *<type>*	Specifies the type of the OCP module. The *<type>* is just a number which you need to figure out in the chip documentation.
RTP.PerBase *<address>*	PERBASE specifies the base address of the core peripheral registers which accesses shall be traced. PERBASE is needed for the RAM Trace Port (RTP) which is available on some derivatives from Texas Instruments. The trace packages include only relative addresses to PERBASE and RAMBASE.
RTP.RamBase *<address>*	RAMBASE is the start address of RAM which accesses shall be traced. RAMBASE is needed for the RAM Trace Port (RTP) which is available on some derivatives from Texas Instruments. The trace packages include only relative addresses to PERBASE and RAMBASE.
STM.Mode [**NONE** | **XTIv2** | **SDTI** | **STP** | **STP64** | **STPv2**]	Selects the protocol type used by the System Trace Module (STM).

STM.Type [**None** | **Generic** | **ARM** | **SDTI** | **TI**]	Selects the type of the System Trace Module (STM). Some types allow to work with different protocols (see STM.Mode).
TPIU.Type [**CoreSight** | **Generic**]	Selects the type of the Trace Port Interface Unit (TPIU). CoreSight: Default. CoreSight TPIU. TPIU control register located at TPIU.Base *<address>* will be handled by the debugger. Generic: Proprietary TPIU. TPIU control register will not be handled by the debugger.

Components and Available Commands

See the description of the commands above. Please note that there is a common description for ... .ATBSource, ...Base, , ...RESet, ...TraceID.

AHBBRIDGE.Base *<address>*
AHBBRIDGE.RESet
Used in a SOC-400 debug infrastructure, if the AHB-AP of the Cortex-M is accessible via an additional bridge on a AHB, AXI or APB bus and not directly as an Access Port. In a SOC-600 debug infrastructure it is irrelevant.

BMC.Base *<address>*
BMC.RESet
Performance Monitor Unit (PMU) - Arm debug module, e.g. on Cortex-A/R
Bench-Mark-Counter (BMC) is the TRACE32 term for the same thing.
The module contains counter which can be programmed to count certain events (e.g. cache hits).

BPU.Base *<address>*
BPU.RESet
Specifies on some Cortex-M core types the Break Point Unit or the Flash Patch and Breakpoint unit. Each kind of Cortex-M core has the one or the other, but not both. This component is already set by default and usually doesn't need to be changed.

CMI.Base *<address>*
CMI.RESet
CMI.TraceID *<id>*
Clock Management Instrumentation (CMI) - Texas Instruments
Trace source delivering information about clock status and events to a system trace module.

COREDEBUG.Base *<address>*
COREDEBUG.RESet
Core Debug Register - Arm debug register, e.g. on Cortex-A/R
Some cores do not have a fix location for their debug register used to control the core. In this case it is essential to specify its location before you can connect by e.g. SYStem.Up.

CTI.Base *<address>*
CTI.Config *<interconnection>*
Cross Trigger Interface (CTI) - Arm CoreSight module
If notified the debugger uses it to synchronously halt (and sometimes also to start) multiple cores.

DRM.Base *<address>*
DRM.RESet
Debug Resource Manager (DRM) - Texas Instruments
It will be used to prepare chip pins for trace output.

DTM.RESet
DTM.Type [**None** | **Generic**]
Data Trace Module (DTM) - generic, CoreSight compliant trace source module
If specified it will be considered in trace recording and trace data can be accessed afterwards.
DTM module itself will not be controlled by the debugger.

DWT.Base *<address>*
DWT.RESet
Data Watchpoint and Trace unit (DWT) - Arm debug module on Cortex-M cores
Normally fix address at 0xE0001000 (default).

EPM.Base *<address>*
EPM.RESet
Emulation Pin Manager (EPM) - Texas Instruments
It will be used to prepare chip pins for trace output.

ETB2AXI.Base *<address>*
ETB2AXI.RESet
ETB to AXI module
Similar to an ETR.

ETB.ATBSource *<source>*
ETB.Base *<address>*
ETB.RESet
ETB.Size *<size>*
Embedded Trace Buffer (ETB) - Arm CoreSight module
Enables trace to be stored in a dedicated SRAM. The trace data will be read out through the debug port after the capturing has finished.

ETF.ATBSource *<source>*
ETF.Base *<address>*
ETF.RESet
Embedded Trace FIFO (ETF) - Arm CoreSight module
On-chip trace buffer used to lower the trace bandwidth peaks.

ETM.Base *<address>*
ETM.RESet
Embedded Trace Macrocell (ETM) - Arm CoreSight module
Program Trace Macrocell (PTM) - Arm CoreSight module
Trace source providing information about program flow and data accesses of a core.
The ETM commands will be used even for PTM.

ETR.ATBSource *<source>*
ETR.CATUBase *<address>*
ETR.Base *<address>*
ETR.RESet
Embedded Trace Router (ETR) - Arm CoreSight module
Enables trace to be routed over an AXI bus to system memory or to any other AXI slave.

ETS.ATBSource *<source>*
ETS.Base *<address>*
ETS.RESet
Embedded Trace Streamer (ETS) - Arm CoreSight module

FPB.Base *<address>*
FPB.RESet
Specifies on some Cortex-M core types the Break Point Unit or the Flash Patch and Breakpoint unit. Each kind of Cortex-M core has the one or the other, but not both. This component is already set by default and usually doesn't need to be changed.

FUNNEL.ATBSource *<sourcelist>*
FUNNEL.Base *<address>*
FUNNEL.Name *<string>*
FUNNEL.PROGrammable [**ON** | **OFF**]
FUNNEL.RESet
CoreSight Trace Funnel (CSTF) - Arm CoreSight module
Combines multiple trace sources onto a single trace bus (ATB = AMBA Trace Bus)

REP.ATBSource *<sourcelist>*
REP.Base *<address>*
REP.Name *<string>*
REP.RESet
CoreSight Replicator - Arm CoreSight module
This command group is used to configure Arm Coresight Replicators with programming interface. After the Replicator(s) have been defined by the base address and optional names the ATB sources REPlicatorA and REPlicatorB can be used from other ATB sinks to connect to output A or B to the Replicator.

HSM.Base *<address>*
HSM.RESet
Hardware Security Module (HSM) - Infineon

HTM.Base *<address>*
HTM.RESet
HTM.Type [**CoreSight** | **WPT**]
AMBA AHB Trace Macrocell (HTM) - Arm CoreSight module
Trace source delivering trace data of access to an AHB bus.

ICE.Base *<address>*
ICE.RESet
ICE-Crusher (ICE) - Texas Instruments

ITM.Base *<address>*
ITM.RESet
Instrumentation Trace Macrocell (ITM) - Arm CoreSight module
Trace source delivering system trace information e.g. sent by software in printf() style.

L2CACHE.Base *<address>*
L2CACHE.RESet
L2CACHE.Type [**NONE** | **Generic** | **L210** | **L220** | **L2C-310** | **AURORA** | **AURORA2**]
Level 2 Cache Controller
The debugger might need to handle the controller to ensure cache coherency for debugger operation.

MTBDWT.Base *<address>*
MTBDWT.RESet
Is a custom extension for the MTB (Micro Trace Buffer) to offer additional address comparators to trigger events on the MTB. It has been implemented by some NXP Cortex-M controllers.

OCP.Base *<address>*
OCP.RESet
OCP.TraceID *<id>*
OCP.Type *<type>*
Open Core Protocol watchpoint unit (OCP) - Texas Instruments
Trace source module delivering bus trace information to a system trace module.

PMI.Base *<address>*
PMI.RESet
PMI.TraceID *<id>*
Power Management Instrumentation (PMI) - Texas Instruments
Trace source reporting power management events to a system trace module.

RTP.Base *<address>*
RTP.PerBase *<address>*
RTP.RamBase *<address>*
RTP.RESet
RAM Trace Port (RTP) - Texas Instruments
Trace source delivering trace data about memory interface usage.

SC.Base *<address>*
SC.RESet
SC.TraceID *<id>*
Statistic Collector (SC) - Texas Instruments
Trace source delivering statistic data about bus traffic to a system trace module.

SDC.Base *<address>*
SDC.RESet
Secure Debug Channel (SDC) - Arm CoreSight module
Communication module sdc600_apbcom_ext for debug authentication.

STM.Base *<address>*
STM.Mode [**NONE** | **XTIv2** | **SDTI** | **STP** | **STP64** | **STPv2**]
STM.RESet
STM.Type [**None** | **Generic** | **ARM** | **SDTI** | **TI**]
System Trace Macrocell (STM) - MIPI, Arm CoreSight, others
Trace source delivering system trace information e.g. sent by software in printf() style.

TPIU.ATBSource *<source>*
TPIU.Base *<address>*
TPIU.RESet
TPIU.Type [**CoreSight** | **Generic**]
Trace Port Interface Unit (TPIU) - Arm CoreSight module
Trace sink sending the trace off-chip on a parallel trace port (chip pins).

In recent years, chips and their debug and trace architectures have become much more complex. The CoreSight trace components and their interconnection on a common trace bus, in particular, necessitated a revision of our commands. The new commands can handle even the most complex structures.

… **BASE** *<address>*

This command informs the debugger about the start address of the register block of the component. And this way it notifies the existence of the component. An on-chip debug and trace component typically provides a control register block which needs to be accessed by the debugger to control this component.

Example: SYStem.CONFIG ETMBASE APB:0x8011c000

Meaning: The control register block of the Embedded Trace Macrocell (ETM) starts at address 0x8011c000 and is accessible via APB bus.

In an SMP (Symmetric MultiProcessing) debug session you can enter for the components BMC, CORE, CTI, ETB, ETF, ETM, ETR a list of base addresses to specify one component per core.

Example assuming four cores: “SYStem.CONFIG COREBASE 0x80001000 0x80003000 0x80005000 0x80007000”.

COREBASE (old syntax: DEBUGBASE): Some cores e.g. Cortex-A or Cortex-R do not have a fix location for their debug register which are used for example to halt and start the core. In this case it is essential to specify its location before you can connect by e.g. **SYStem.Up**.

PERBASE and RAMBASE are needed for the RAM Trace Port (RTP) which is available on some derivatives from Texas Instruments. PERBASE specifies the base address of the core peripheral registers which accesses shall be traced, RAMBASE is the start address of RAM which accesses shall be traced. The trace packages include only relative addresses to PERBASE and RAMBASE.

For a list of possible components including a short description see **Components and Available Commands**.

… **PORT** *<port>*	Informs the debugger about which trace source is connected to which input port of which funnel. A CoreSight trace funnel provides 8 input ports (port 0-7) to combine the data of various trace sources to a common trace stream. **Example**: SYStem.CONFIG STMFUNNEL2PORT 3 Meaning: The System Trace Module (STM) is connected to input port #3 on FUNNEL2. On an SMP debug session some of these commands can have a list of *<port>* parameter. In case there are dedicated funnels for the ETB and the TPIU their base addresses are specified by ETBFUNNELBASE, TPIUFUNNELBASE respectively. And the funnel port number for the ETM are declared by ETMETBFUNNELPORT, ETMTPIUFUNNELPORT respectively. For a list of possible components including a short description see **Components and Available Commands**.
BYPASS *<seq>*	With this option it is possible to change the JTAG bypass instruction pattern for other TAPs. It works in a multi-TAP JTAG chain for the IRPOST pattern, only, and is limited to 64 bit. The specified pattern (hexadecimal) will be shifted least significant bit first. If no BYPASS option is used, the default value is “1” for all bits.
CTICONFIG *<type>*	Informs about the interconnection of the core Cross Trigger Interfaces (CTI). Certain ways of interconnection are common and these are supported by the debugger e.g. to cause a synchronous halt of multiple cores. NONE: The CTI is not used by the debugger. ARMV1: This mode is used for Arm7/9/11 cores which support synchronous halt, only. ARMPostInit: Like ARMV1 but the CTI connection differs from the Arm recommendation. OMAP3: This mode is not yet used. TMS570: Used for a certain CTI connection used on a TMS570 derivative. CortexV1: The CTI will be configured for synchronous start and stop via CTI. It assumes the connection of DBGRQ, DBGACK, DBGRESTART signals to CTI are done as recommended by Arm. The CTIBASE must be notified. “CortexV1” is the default value if a Cortex-A/R core is selected and the CTIBASE is notified. QV1: This mode is not yet used.

DTMCONFIG [**ON** | **OFF**]	Informs the debugger that a customer proprietary Data Trace Message (DTM) module is available. This causes the debugger to consider this source when capturing common trace data. Trace data from this module will be recorded and can be accessed later but the unknown DTM module itself will not be controlled by the debugger.
FILLDRZERO [**ON** | **OFF**]	This changes the bypass data pattern for other TAPs in a multi-TAP JTAG chain. It changes the pattern from all "1" to all "0". This is a workaround for a certain chip problem. It is available on the Arm9 debugger, only.
TIOCPTYPE *<type>*	Specifies the type of the OCP module from Texas Instruments (TI).
view	Opens a window showing most of the SYStem.CONFIG settings and allows to modify them.

Mapping Deprecated to New Commands

In the following you find the list of deprecated commands which can still be used for compatibility reasons and the corresponding new command.

SYStem.CONFIG *<parameter>*

<parameter>: (Deprecated)	*<parameter>*: (New)
BMCBASE *<address>*	**BMC.Base** *<address>*
BYPASS *<seq>*	**CHIPIRPRE** *<bits>* **CHIPIRLENGTH** *<bits>* **CHIPIRPATTERN.Alternate** *<pattern>*
COREBASE *<address>*	**COREDEBUG.Base** *<address>*
CTIBASE *<address>*	**CTI.Base** *<address>*
CTICONFIG *<type>*	**CTI.Config** *<type>*
DEBUGBASE *<address>*	**COREDEBUG.Base** *<address>*
DTMCONFIG [**ON** | **OFF**]	**DTM.Type.Generic**
DTMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource DTM** *<port>* (1)
DTMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource DTM** *<port>* (1)
DTMFUNNELPORT *<port>*	**FUNNEL1.ATBSource DTM** *<port>* (1)
DTMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource DTM** *<port>* (1)
DWTBASE *<address>*	**DWT.Base** *<address>*
ETB2AXIBASE *<address>*	**ETB2AXI.Base** *<address>*

ETBBASE *<address>*	**ETB1.Base** *<address>*
ETBFUNNELBASE *<address>*	**FUNNEL4.Base** *<address>*
ETFBASE *<address>*	**ETF1.Base** *<address>*
ETMBASE *<address>*	**ETM.Base** *<address>*
ETMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource ETM** *<port>* (1)
ETMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource ETM** *<port>* (1)
ETMFUNNELPORT *<port>*	**FUNNEL1.ATBSource ETM** *<port>* (1)
ETMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource ETM** *<port>* (1)
FILLDRZERO [**ON** | **OFF**]	**CHIPDRPRE** 0 **CHIPDRPOST** 0 **CHIPDRLENGTH** *<bits_of_complete_dr_path>* **CHIPDRPATTERN.Alternate** 0
FUNNEL2BASE *<address>*	**FUNNEL2.Base** *<address>*
FUNNELBASE *<address>*	**FUNNEL1.Base** *<address>*
HSMBASE *<address>*	**HSM.Base** *<address>*
HTMBASE *<address>*	**HTM.Base** *<address>*
HTMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource HTM** *<port>* (1)
HTMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource HTM** *<port>* (1)
HTMFUNNELPORT *<port>*	**FUNNEL1.ATBSource HTM** *<port>* (1)
HTMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource HTM** *<port>* (1)
ITMBASE *<address>*	**ITM.Base** *<address>*
ITMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource ITM** *<port>* (1)
ITMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource ITM** *<port>* (1)
ITMFUNNELPORT *<port>*	**FUNNEL1.ATBSource ITM** *<port>* (1)
ITMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource ITM** *<port>* (1)
PERBASE *<address>*	**RTP.PerBase** *<address>*
RAMBASE *<address>*	**RTP.RamBase** *<address>*
RTPBASE *<address>*	**RTP.Base** *<address>*
SDTIBASE *<address>*	**STM1.Base** *<address>* **STM1.Mode SDTI** **STM1.Type SDTI**
STMBASE *<address>*	**STM1.Base** *<address>* **STM1.Mode STPV2** **STM1.Type ARM**
STMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource STM1** *<port>* (1)
STMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource STM1** *<port>* (1)
STMFUNNELPORT *<port>*	**FUNNEL1.ATBSource STM1** *<port>* (1)
STMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource STM1** *<port>* (1)

TIDRMBASE *<address>*	**DRM.Base** *<address>*
TIEPMBASE *<address>*	**EPM.Base** *<address>*
TIICEBASE *<address>*	**ICE.Base** *<address>*
TIOCPBASE *<address>*	**OCP.Base** *<address>*
TIOCPTYPE *<type>*	**OCP.Type** *<type>*
TIPMIBASE *<address>*	**PMI.Base** *<address>*
TISCBASE *<address>*	**SC.Base** *<address>*
TISTMBASE *<address>*	**STM1.Base** *<address>* **STM1.Mode STP** **STM1.Type TI**
TPIUBASE *<address>*	**TPIU.Base** *<address>*
TPIUFUNNELBASE *<address>*	**FUNNEL3.Base** *<address>*
view	**state**

(1) Further “*<component>*.ATBSource *<source>*” commands might be needed to describe the full trace data path from trace source to trace sink.

SYStem.CONFIG.EXTWDTDIS — Disable external watchdog

Format:	**SYStem.CONFIG.EXTWDTDIS** *<option>*
<option>:	**OFF** **High** **Low** **HighwhenStopped** **LowwhenStopped**

Default for Automotive/Automotive PRO Debug Cable: High.
Default for XCP: OFF.

Controls the WDTDIS pin of the debug port. This configuration is only available for tools with an Automotive Connector (e.g., Automotive Debug Cable, Automotive PRO Debug Cable) and XCP.

OFF	The WDTDIS pin is not driven. (XCP only)
High	The WDTDIS pin is permanently driven high.
Low	The WDTDIS pin is permanently driven low.
HighwhenStopped	The WDTDIS pin is driven high when program is stopped (not XCP).
LowwhenStopped	The WDTDIS pin is driven low when program is stopped (not XCP).

SYStem.CPU — Select the used CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**CORTEXM3**

Default selection: CORTEXM3

Selects the processor type. If your ASIC is not listed, select the type of the integrated ARM core.

Format:	**SYStem.JtagClock** [*<frequency>* | **RTCK** | **ARTCK** *<frequency>* | **CTCK** *<frequency>* | **CRTCK** *<frequency>*]
	SYStem.BdmClock (deprecated)
<frequency>:	**6 kHz** … **80 MHz**

Default frequency: 10 MHz.

Selects the frequency (TCK/SWCLK) used by the debugger to communicate with the processor in JTAG, SWD or cJTAG mode. The frequency can affect e.g. the download speed. It could be required to reduce the frequency if there are buffers, additional loads or high capacities on the debug port signals or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer. Therefore we recommend to use the default setting if possible.

<frequency>	• The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window. • Besides a decimal number like “100000.” short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.
RTCK	The debug clock is controlled by the RTCK signal (Returned TCK). On some multicore systems there is the need to synchronize the processor clock and the debug port clock. In this case RTCK shall be selected. Synchronization is maintained, because the debugger does not progress to the next TCK/SWCLK edge until after an RTCK edge is received. In case you have a processor derivative requiring a synchronization of the processor clock and the debug clock, but your target does not provide an RTCK signal, you need to select a fixed frequency below which is low enough to be adequate to the speed you would reach if RTCK is available. When RTCK is selected, the frequency depends on the processor clock and on the propagation delays. The maximum reachable frequency is about 16 MHz.

```
SYStem.JtagClock RTCK
```

ARTCK	Accelerated method to control the debug clock by the RTCK signal (Accelerated Returned TCK). This option is only relevant for JTAG debug ports. On some multicore systems, which need a synchronization of the processor clock RTCK mode only allows theoretical frequencies up to 1/6 or 1/8 of the processor clock. For such designs using a very low processor clock we offer a different mode (ARTCK) which does not work as recommended by ARM and might not work on all target systems. In ARTCK mode the debugger uses a fixed frequency for TCK, independent of the RTCK signal. This frequency must be specified by the user and has to be below 1/3 of the processor clock speed. TDI and TMS will be delayed by 1/2 TCK clock cycle. TDO will be sampled with RTCK.
CTCK	With this option higher debug port speeds can be reached. The TDO/SWDIO signal will be sampled by a signal which derives from TCK/SWCLK, but which is timely compensated regarding the debugger-internal driver propagation delays (**C**ompensation by **TCK**). This feature can be used with a debug cable version 3 or newer. If it is selected, although the debug cable is not suitable, a fixed frequency will be selected instead (minimum of 10 MHz and selected clock).
CRTCK	With this option higher debug port speeds can be reached. The TDO/SWDIO signal will be sampled by the RTCK signal. This compensates the debugger-internal driver propagation delays, the delays on the cable and on the target (**C**ompensation by **RTCK**). This feature requires that the target provides an RTCK signal. In contrast to the **RTCK** option, the TCK/SWCLK is always output with the selected, fixed frequency.

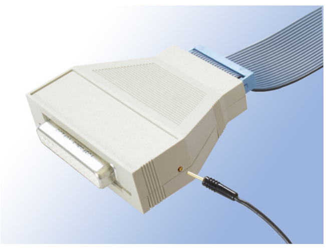
Format:	**SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

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

It must be ensured that the state of the Arm core 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. Please note: nTRST must have a pull-up resistor on the target, EDBGRQ must have a pull-down resistor.

There is a single cable contact on the casing of the debug cable. This contact can be used to detect if the JTAG connector is tristated. If tristated also this signal is tristated, it is pulled low otherwise.



Format:	**SYStem.MemAccess** *<mode>*
<mode>:	**AHB** | **AXI** | **APB** | … (SoC-600) **DAP** (SoC-400) **Enable** **RealMON** **TrkMON** **GdbMON** **Denied** **StopAndGo**

Default: Denied.

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

AHB, **AXI**, **APB**, …	Depending on which memory access ports are available on the chip, the memory access is done through the specified bus.
DAP	A run-time memory access is done through the MEM-AP AHB via DAP (Debug Access Port). Since this is nearly non-intrusive and does not require any monitor running on the target, it normally will be the best selection for Cortex-M.
Denied	No memory access is possible while the CPU is executing the program.
Enable **CPU** (deprecated)	Used to activate the memory access while the CPU is running on the TRACE32 Instruction Set Simulator and on debuggers which do not have a fixed name for the memory access method.
GdbMON	A run-time memory access is done via the GDB Server from Linux.
RealMON	A run-time memory access is done via the Real Monitor from ARM.
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.
TrkMON	A run-time memory access is done via the TRKMON from Symbian.

A run-time access can be done by using the access class prefix “E”. At first sight it is not clear, whether this causes a read access through the CPU, the AHB/AXI bypassing the CPU, or no read access at all. The following tables will summarize this effect. “E” can be combined with various access classes. The following example uses the access class “A” (physical access) to illustrate the effect of “E”.

CPU stopped

SYStem.CpuSpot Enabled				
SYS.MA. Access class	**Denied**	**DAP (SoC-400 only)**	**[AHB | AXI] (SoC-600 only)**	**StopAndGo**
EA	CPU*	AHB/AXI	AHB/AXI	CPU*
A	CPU*	CPU*	CPU*	CPU*
AHB or AXI	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI
EAHB or EAXI	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI

SYStem.CpuSpot [Denied | Target | SINGLE]				
SYS.MA. Access class	**Denied**	**DAP (SoC-400 only)**	**[AHB | AXI] (SoC-600 only)**	**StopAndGo**
EA	CPU*	AHB/AXI	AHB/AXI	not allowed
A	CPU*	CPU*	CPU*	not allowed
AHB or AXI	AHB/AXI	AHB/AXI	AHB/AXI	not allowed
EAHB or EAXI	AHB/AXI	AHB/AXI	AHB/AXI	not allowed

CPU running

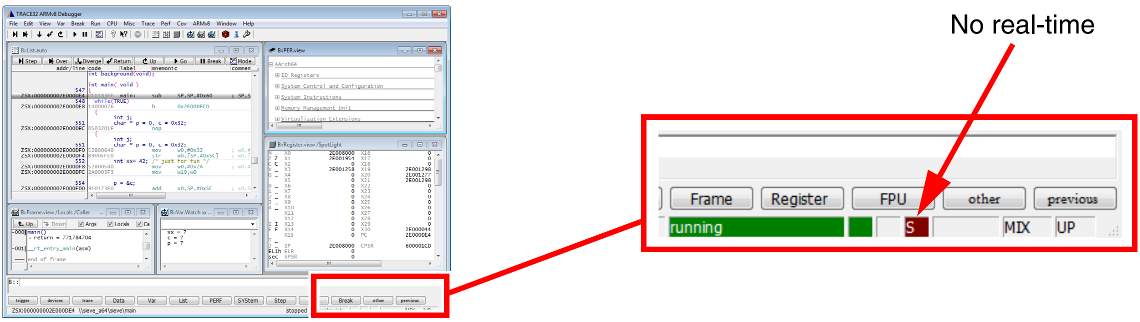
SYStem.CpuSpot Enabled				
SYS.MA. Access class	**Denied**	**DAP (SoC-400 only)**	**[AHB | AXI] (SoC-600 only)**	**StopAndGo**
EA	no access	AHB/AXI	AHB/AXI	CPU* (spotted)
A	no access	no access	no access	no access
AHB or AXI	no access	no access	no access	no access
EAHB or EAXI	AHB/AXI	AHB/AXI	AHB/AXI	AHB/AXI

SYStem.CpuSpot [Denied | Target | SINGLE]				
SYS.MA. Access class	**Denied**	**DAP (SoC-400 only)**	**[AHB | AXI] (SoC-600 only)**	**StopAndGo**
EA	no access	AHB/AXI	AHB/AXI	not allowed

<table>
<tr><th colspan="5">SYStem.CpuSpot [Denied | Target | SINGLE]</th></tr>
<tr><th>SYS.MA.
Access class</th><th>Denied</th><th>DAP
(SoC-400 only)</th><th>[AHB | AXI]
(SoC-600 only)</th><th>StopAndGo</th></tr>
<tr><td>A</td><td>no access</td><td>no access</td><td>no access</td><td>not allowed</td></tr>
<tr><td>AHB or AXI</td><td>no access</td><td>no access</td><td>no access</td><td>not allowed</td></tr>
<tr><td>EAHB or EAXI</td><td>AHB/AXI</td><td>AHB/AXI</td><td>AHB/AXI</td><td>not allowed</td></tr>
</table>

*) Cortex-M: The "CPU" access uses the AHB/AXI access path instead, due to the debug interface design.

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
```

If you want this for all your TRACE32 windows, then select **SYStem.Option.DUALPORT ON**.

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** **StandBy** **Up** **Prepare**

Down	Disables the debugger (default). The state of the CPU remains unchanged. The JTAG port is tristated.
NoDebug	Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tristated.
Go	Resets the target and enables the debugger and start the program execution. The nSRST signal will be pulsed and a software reset will be performed. Program execution can be stopped by the **Break** command or an external trigger.
Attach	The mode of the core (running or halted) does not change, but debugging will be initialized. After this command, the user program can be stopped with the **Break** command or if any break condition occurs.
StandBy	Resets the target, waits until power is detected, restores the debug registers (e.g. breakpoints, trace control), releases reset to start the program execution. When power goes down again, it switches automatically back to this state. This allows debugging of a power cycle, because debug register will be restored on power up. Please note that the debug register require a halt/go sequence to become active. It is not sufficient to set breakpoints in **Down** mode and switch to **StandBy** mode. Exception: On-chip breakpoints and vector catch register can be set while the program is “running”. This mode is not yet available.

Up	Resets the target, sets the core to debug mode and stops the core. The nSRST signal will be pulsed and a software reset will be performed. After the execution of this command the core is stopped and all register are set to the default level. You need to execute **Register.Init** to force the debugger to read out the program counter and stack pointer address.
Prepare	Resets the target, initializes the JTAG interface, but does not connect to the ARM core. This debugger startup is used if no ARM core shall be debugged. It can be used if a user program or proprietary debugger uses the TRACE32 API (application programming interface) to access the JTAG interface via the TRACE32 debugger hardware.

SYStem.Option — Special setup

[SYStem.state window > Option]

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

SYStem.Option.BigEndian — Define byte order (endianness)

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

This option is normally not needed because Cortex-M is always little endian. But there are special chip designs where the debugger needs to handle the data as big endian.

SYStem.Option.CFLUSHAFTERBREAK — Flush data cache after break

[build 153369 - DVD 02/2023]

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

Default: OFF.

This command is used to clean and invalidate the data cache each time after a break.

SYStem.Option.CLEARHARDFAULT — Handle the HFSR[FORCED] bit

[build 154175 - DVD 02/2023]

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

Default: ON.

This command is used to clear or keep the HFSR[FORCED] bit set or cleared on each restart of the core.

The HFSR[FORCED] bit is set by the core to indicate a HARDFAULT exception entry. The debugger evaluates it for displaying the status “stopped by hardfault”. By default, the debugger clears the bit by the next restart of the core. To allow the target exception handler to evaluate this bit by itself the option can be set to OFF. “stopped by hardfault” will then be displayed as break reason as long as the bit is not cleared by the target application.

SYStem.Option.CoreSightRESet — Assert CPU reset via CTRL/STAT

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

Default: OFF.

The CPU is reset via the CTRL/STAT.CDBGRSTREQ bit. This feature is highly SoC specific and should only be used if this reset method is really implemented.

SYStem.Option.CORTEXMAHB — AHB-AP type of the Cortex-M

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

Default: ON.

This option needs to be turned off, if the Cortex-M core is accessed via a standard AHB Access Port provided by the ARM CoreSight design kit that needs to be handled different than the Cortex-M AHB Access Port.

SYStem.Option.CypressACQuire — Send acquire sequence

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

Send an acquire sequence after reset to put the MCU into test mode. Prevents the execution of user code from flash regions. This command is only available for certain Cypress chips.

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.

ON	Debug power is requested by the debugger on a debug session start, and the control bit is set to 1. The debug power is released at the end of the debug session, and the control bit is set to 0.
AlwaysON	Debug power is requested by the debugger on a debug session start, and the control bit is set to 1. The debug power is **not** released at the end of the debug session, and the control bit is set to 0.
OFF	Only for test purposes: Debug power is **not** requested and **not** checked by the debugger. The control bit is set to 0.

Use case:

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

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

SYStem.Option.DAP2DBGPWRUPREQ — Force debug power in DAP2

Format:	**SYStem.Option.DAP2DBGPWRUPREQ** [**ON** | **AlwaysON**]

Default: ON.

This option controls the DBGPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port 2 (DAP2) before and after the debug session. Debug power will always be requested by the debugger on a debug session start.

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

Use case:

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

To keep the debug interface active, it is recommended that **SYStem.Option.DAP2DBGPWRUPREQ** is set to **AlwaysON**.

SYStem.Option.DAPSYSPWRUPREQ — Force system power in DAP

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

Default: ON.

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

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

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

SYStem.Option.DAP2SYSPWRUPREQ Force system power in DAP2

Format: **SYStem.Option.DAP2SYSPWRUPREQ** [**AlwaysON** | **ON** | **OFF**]

Default: ON.

This option controls the SYSPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port 2 (DAP2) during and after the debug session

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

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 mapping on the DAP is different than the processor view, then this re-mapping command can be used

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

SYStem.Option.DEBUGPORTOptions — Options for debug port handling

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.DIAG — Activate more log messages

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

Default: OFF.

Adds more information to the report in the **SYStem.LOG.List** window.

Format:	**SYStem.Option.DisMode** *<option>*
<option>:	**AUTO** **ACCESS** **ARM** **THUMB** **THUMBEE**

Specifies the selected disassembler.

AUTO (default)	The information provided by the compiler output file is used for the disassembler selection. If no information is available it has the same behavior as the option ACCESS.
ACCESS	The selected disassembler depends on the T bit in the CPSR or on the selected access class. (e.g. `Data.List SR:0` for ARM mode or `Data.List ST:0` for THUMB mode).
ARM	Only the ARM disassembler is used (highest priority).
THUMB	Only the THUMB disassembler is used (highest priority).
THUMBEE	Only the THUMB disassembler is used which supports the Thumb-2 Execution Environment extension (highest priority).

SYStem.Option.DUALPORT Implicitly use run-time memory access

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

All TRACE32 windows that display memory are updated while the processor is executing code (e.g. **Data.dump**, **List.auto**, **PER.view**, **Var.View**). This setting has no effect if **SYStem.MemAccess** is disabled.

If only selected memory windows should update their content during runtime, leave **SYStem.Option.DUALPORT OFF** and use the access class prefix **E** or the format option **%E** for the specific windows.

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

[SYStem.state window> EnReset]

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.

From the view of the core, it is not necessary that nRESET (nSRST) becomes active at the start of a debug session (**SYStem.Up**), but there may be other logic on the target which requires a reset.

SYStem.Option.FORCESECure — Force secure memory access

[build 149151 - DVD 09/2022]

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

Default: OFF.

Forces default secure memory access for cores without TrustZone support. This option is only supported for cores with generic AHB-AP.

SYStem.Option.IMASKASM — Disable interrupts while single stepping

[SYStem.state window > IMASKASM]

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

[SYStem.state window > IMASKHLL]

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.

SYStem.Option.INTDIS Disable all interrupts

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

Default: OFF.

If this option is ON, all interrupts on the Arm core are disabled.

SYStem.Option.IntelSOC Slave core is part of Intel® SoC

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

Default: OFF.

Informs the debugger that the core is part of an Intel® SoC. When enabled, all IR and DR pre/post settings are handled automatically, no manual configuration is necessary.

Requires that the debugger for this core is slave in a multicore setup with x86 as the master debugger and that **SYStem.Option.CLTAPOnly** is enabled in the x86 debugger.

SYStem.Option.LOCKRES Go to "Test-Logic Reset" when locked

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

This command is only available on obsolete ICD hardware. The state machine of the JTAG TAP controller is switched to Test-Logic Reset state (ON) or to Run-Test/Idle state (OFF) before a **SYStem.LOCK ON** is executed.

SYStem.Option.MDMAP Set debug option controlled by NXP MDM-AP

[build 121894 - DVD 09/2022]

Format:	**SYStem.Option.MDMAP** *<option>*
<option>:	**DestructiveReset** [**ON** | **OFF**] **FunctionalReset** [**ON** | **OFF**] **HaltAfterPoWeRUP** [**ON** | **OFF**] **DBGRSTFASTPAD** [**ON** | **OFF**] **DBGRSTSLOWPAD** [**ON** | **OFF**] **PORWDGDIS** [**ON** | **OFF**] **WFIFIX** [**ON** | **OFF**]

Allows to set different debug option controlled by the NXP MDM-AP inside devices, where it is implemented.

DestructiveReset [**ON** | **OFF**]	Default: OFF. Generates a destructive reset during **SYSem.Up** or **SYStem.Mode.Go**.
FunctionalReset [**ON** | **OFF**]	Default: OFF. Generates a functional (warm) reset during **SYSem.Up** or **SYStem.Mode.Go**.
HaltAfterPoWeRUP [**ON** | **OFF**]	Default: OFF. Can be used to stop the master core on the first instruction after reset from a power-up transition using **SYStem.Mode.StandBy**. This ensures, that no code has been executed on the target, when first powering on the target board.
DBGRSTFASTPAD [**ON** | **OFF**]	Default: OFF. Turning on the fast IO pins using for tracing.

DBGRSTSLOWPAD [**ON** | **OFF**]	Default: OFF. Turning on the slow IO pins using for tracing.
PORWDGDIS [**ON** | **OFF**]	Default: OFF. Disabling the power watchdog inside the device.
WFIFIX [**ON** | **OFF**]	Default: ON. Workaround for WFI/WFE entrance of Cortex-M7 cores in some NXP S32 devices. In case the debugger is disconnected from the target using **SYStem.Down**, the set WFIFIX option ensures, that the Cortex-M7 still can wake-up correctly from WFI/WFE state.

SYStem.Option.MMUSPACES

Enable space IDs

Only available for CPUs with MMUs

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

Default: OFF.

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

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

NOTE:	**SYStem.Option.MMUSPACES** should not be set to **ON** if only one translation table is used on the target. If a debug session requires space IDs, you must observe the following sequence of steps: 1. Activate **SYStem.Option.MMUSPACES**. 2. Load the symbols with **Data.LOAD**. Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

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

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

SYStem.Option.NoRunCheck No check of the running state

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

Default: OFF.

If this option is ON, it advises the debugger not to do any running check. In this case the debugger does not even recognize that there will be no response from the processor. Therefore there always is the message “running”, independent of whether the core is in power down or not. This can be used to overcome power saving modes in case users know when a power saving mode happens and that they can manually de-activate and re-activate the running check.

NOTE:	This command will affect the setting of **SYStem.POLLING** *<stopped_mode>*.

SYStem.Option.OVERLAY Enable overlay support

Format:	**SYStem.Option.OVERLAY** [**ON** | **OFF** | **WithOVS**]

Default: OFF.

ON
Activates the overlay extension and extends the address scheme of the debugger with a 16 bit virtual overlay ID. Addresses therefore have the format *<overlay_id>***:***<address>*. This enables the debugger to handle overlaid program memory.

OFF
Disables support for code overlays.

WithOVS
Like option **ON**, but also enables support for software breakpoints. This means that TRACE32 writes software breakpoint opcodes to both, the *execution area* (for active overlays) and the *storage area*. This way, it is possible to set breakpoints into inactive overlays. Upon activation of the overlay, the target’s runtime mechanisms copies the breakpoint opcodes to the execution area. For using this option, the storage area must be readable and writable for the debugger.

Example:

```
SYStem.Option.OVERLAY ON
List.auto 0x2:0x11c4                    ; List.auto <overlay_id>:<address>
```

SYStem.Option.PALLADIUM — Extend debugger timeout

Format:	**SYStem.Option.PALLADIUM** [**ON** | **OFF**] (deprecated) **Use SYStem.CONFIG.DEBUGTIMESCALE instead.**

Default: OFF.

The debugger uses longer timeouts as might be needed when used on a chip emulation system like the Palladium from Cadence.

This option will only extend some timeouts by a fixed factor. It is recommended to extend all timeouts. This can be done with **SYStem.CONFIG.DEBUGTIMESCALE**.

SYStem.Option.PSOCswdACQuire — Debug port acquire for PSOC5

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

Default: OFF.

Allows switching USB pins into SWD mode using a Debug Port Acquire key on some Cypress PSOC5 devices during SYStem.Mode Up/Go/Prepare. This command is only available if it is supported by the selected CPU.

SYStem.Option.PWRDWNRecover — Mode to handle special power recovery

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

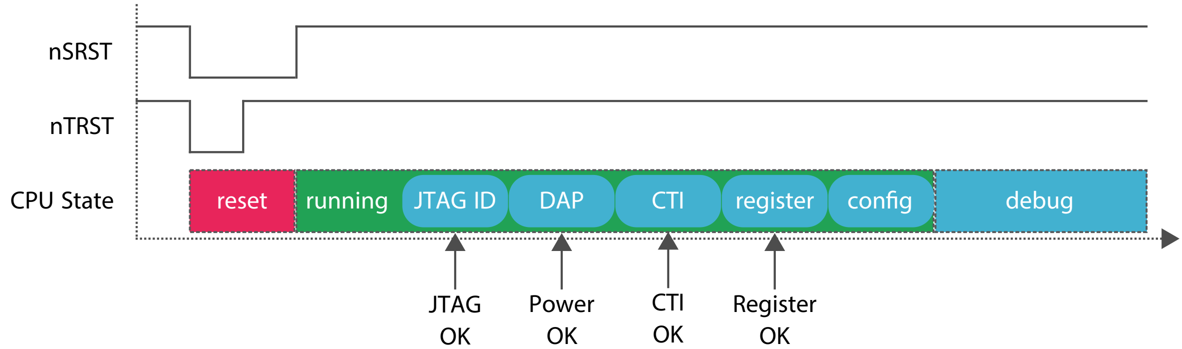
Some power saving states of Cortex-M controller additionally turn off the debug interface. The debugger usually would go into **SYStem.Down** state with a “emulation debug port fail” error message. Turning on this option the debugger assumes that the target has entered a power saving state and permanently tries to reconnect to the device, so that the debug session will not go lost. Additionally the debugger tries to restore all debug and trace registers, if possible.

Attention: This option will turn off basic debug connectivity checks. If there are problems with the debug port, then they might not be detected. Instead of this false power down messages will be displayed.

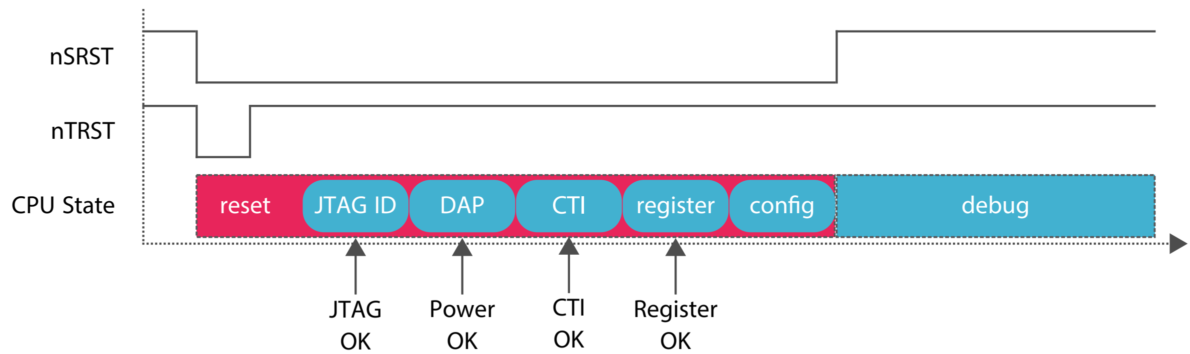
Format:	**SYStem.Option.ResBreak** [**ON** | **OFF**]

Default: ON.

This option has to be disabled if the nTRST line is connected to the nRESET / nSRST line on the target. In this case the CPU executes some cycles while the **SYStem.Up** command is executed. The reason for this behavior is the fact that it is necessary to halt the core (enter debug mode) by a JTAG sequence. This sequence is only possible while nTRST is inactive. In the following figure the marked time between the deassertion of reset and the entry into debug mode is the time of this JTAG sequence plus a time delay selectable by **SYStem.Option.WaitReset** (default = 3 msec).



If nTRST is available and not connected to nRESET/nSRST it is possible to force the CPU directly after reset (without cycles) into debug mode. This is also possible by pulling nTRST fixed to VCC (inactive), but then there is the problem that it is normally not ensured that the JTAG port is reset in normal operation. If the ResBreak option is enabled the debugger first deasserts nTRST, then it executes a JTAG sequence to set the DBGRQ bit in the ICE breaker control register and then it deasserts nRESET/nSRST.



SYStem.Option.RESetREGister Generic software reset

Format:	**SYStem.Option.RESetRegister NONE** **SYStem.Option.RESetRegister** *<address>* *<mask>* *<assert_value>* *<deassert_value>* [**/***<width>*]
<width>:	**Byte** | **Word** | **Long** | **Quad**

Specifies a register on the target side, which allows the debugger to assert a software reset, in case no nReset line is present on the JTAG header. The reset is asserted on **SYStem.Up**, **SYStem.Mode.Go**, **SYStem.Mode Prepare** and **SYStem.RESetOut**. The specified address needs to be accessible during runtime (for example E, DAP, AXI, AHB, APB).

<address>	Specifies the address of the target reset register.
<mask>	The *<assert_value>* and *<deassert_value>* are written in a read-modify-write operation. The mask specifies which bits are changed by the debugger. Bits of the mask value which are '1' are not changed inside the reset register.
<assert_value>	Value that is written to assert reset.
<deassert_value>	Value that is written to deassert reset.
<width>	Width used for register access. See also **"Keywords for <width>"** (general_ref_d.pdf).

SYStem.Option.RisingTDO Target outputs TDO on rising edge

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

Default: OFF.

Bug fix for chips which output the TDO on the rising edge instead of on the falling.

SYStem.Option.SELECTDAP — Select Cortex-M DAP

Format:	**SYStem.Option SELECTDAP** [**DAP** | **DAP2**]

Selects if the Cortex-M core is debugged via the DAP (default) or DAP2. For debugging via DAP2 a second DAP need to be present in the chip and need to be configured by **SYStem.CONFIG**.

SYStem.Option.SOFTLONG — Use 32-bit access to set breakpoint

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

Default: OFF.

Instructs the debugger to use 32-bit accesses to patch the software breakpoint code.

SYStem.Option.SOFTWORD — Use 16-bit access to set breakpoint

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

Default: OFF.

Instructs the debugger to use 16-bit accesses to patch the software breakpoint code.

SYStem.Option.STEPSOFT — Use software breakpoints for ASM stepping

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

Default: OFF.

If set to ON, software breakpoints are used for single stepping on assembler level (advanced users only).

SYStem.Option.SYSPWRUPREQ   Force system power

Format:	**SYStem.Option.SYSPWRUPREQ** [**ON** | **OFF**] (deprecated) **Use SYStem.Option.DAPSYSPWRUPREQ instead.**

Default: ON.

This option controls the SYSPWRUPREQ bit of the CTRL/STAT register of the Debug Access Port (DAP). If the option is ON, system power will be requested by the debugger on a debug session start.

This option is for target processors having a Debug Access Port (DAP).

SYStem.Option.SYSRESETREQ   Allow system reset via the AIRC register

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

Default: depends on the selected CPU.

This option allows the debugger to assert a software reset using the SYSRESETREQ flag inside the Application Interrupt and Reset Control Register (AIRCR) of the Cortex-M core. Its effect depends on the implementation inside the microcontroller or SOC.

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.

SYStem.Option.VECTRESET — Allow local reset via the AIRC register

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

Default: OFF.

Allows the debugger to assert a local software reset using the VECTRESET flag inside the Application Interrupt and Reset Control Register (AIRCR) of the Cortex-M core. Its system wide effect depends on the implementation inside the microcontroller or SOC.

This option is not available for ARMv6-M core.

SYStem.Option.WaitIDCODE — IDCODE polling after deasserting reset

Format: **SYStem.Option.WaitIDCODE** [**ON** | **OFF** | *<time>*]

Default: OFF = disabled.

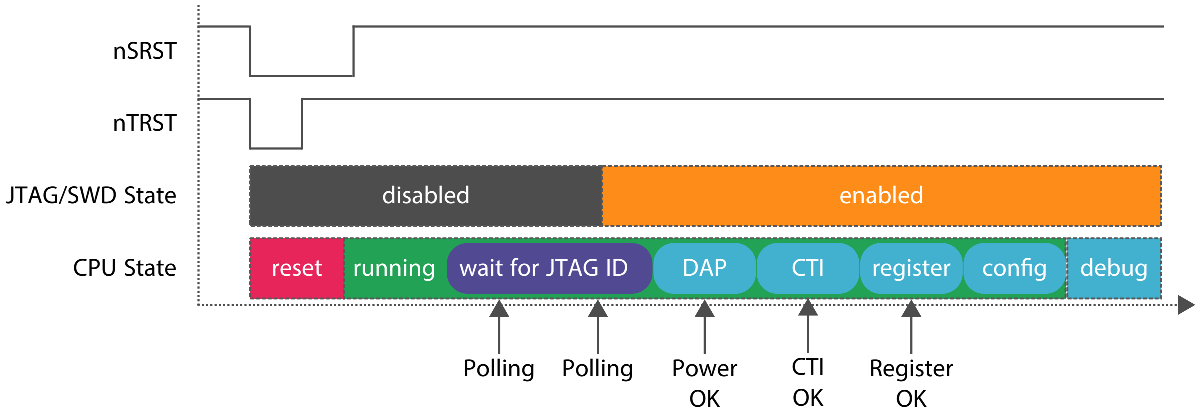
Allows to add additional busy time after reset. The command is limited to systems that use an Arm DAP.

If **SYStem.Option.WaitIDCODE** is enabled and **SYStem.Option.ResBreak** is disabled, the debugger starts to busy poll the JTAG/SWD IDCODE until it is readable. For systems where JTAG/SWD is disabled after RESET and e.g. enabled by the BootROM, this allows an automatic adjustment of the connection delay by busy polling the IDCODE.

After deasserting nSRST and nTRST the debugger waits the time configured by **SYStem.Option.WaitReset** till it starts to busy poll the JTAG/SWD IDCODE. As soon as the IDCODE is readable, the regular connection sequence continues.

ON	1 second busy polling
OFF	Disabled
<time>	Configurable polling time, max. 30 sec, use ’us’, ’ms, ’s’ as units.

Example: The following figure shows a scenario with **SYStem.Option.ResBreak** disabled and **SYStem.Option.WaitIDCODE** enabled. The polling mechanism tries to minimize the delay between the JTAG/SWD disabled and debug state.



SYStem.Option.WaitReset Wait with JTAG activities after deasserting reset

[SYStem.state window > WaitReset]

Format:	**SYStem.Option.WaitReset** [**ON** | **OFF** | *<time>*]

Default: OFF = 3 msec.

Allows to add additional wait time after reset.

ON	1 sec delay
OFF	3 msec delay
<time>	Selectable time delay, min. 50 usec, max. 30 sec, use 'us', 'ms, 's' as units.

If **SYStem.Option.ResBreak** is enabled, **SYStem.Option.WaitReset** should be set to **OFF**.

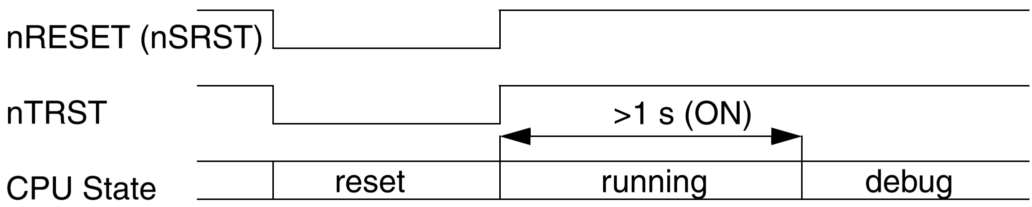
If **SYStem.Option.ResBreak** is disabled, **SYStem.Option.WaitReset** can be used to specify a waiting time between the deassertion of nSRST and nTRST and the first JTAG activity. During this time the core may execute some code, e.g to enable the JTAG port.

If **SYStem.Option.WaitReset** is disabled (**OFF**) and **SYStem.Option.ResBreak** is disabled, the debugger waits 3 ms after the deassertion of nSRST and nTRST before the first JTAG/SWD activity.

If **SYStem.Option.WaitReset** *<time>* is specified and **SYStem.Option.ResBreak** is disabled, the debugger waits the specified *<time>* after the deassertion of nSRST and nTRST before the first JTAG/SWD activity.

If **SYStem.Option.WaitReset** is enabled (**ON**) and **SYStem.Option.ResBreak** is disabled, the debugger waits for at least 1 s, then it waits until nSRST is released from target side; the max. wait time is 35 s (see picture below).

If the chip additionally supports soft reset methods then the wait time can happen more than once.



SYStem.Option.WakeUpACKnowledge — Set acknowledge after wake-up

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

Some targets additionally need an acknowledge by the debugger after a wake-up to be sure that debug and trace are working correctly after leaving a deep sleep or power off state. Additionally to get this option to take effect is, to set **SYStem.Option.PWRDWNRecover ON**.

Attention: Depending on the target setting this option may have an impact to the clock and power management of the chip. The target software might behave differently, when this option is set.

SYStem.RESetOut — Performs a reset

Format:	**SYStem.RESetOut**

This command asserts the nSRST line on the JTAG connector and performs a software reset. This command can be used independent if the core is running or halted.

SYStem.state — Display SYStem.state window

Format:	**SYStem.state**

Displays the **SYStem.state** window for Cortex-M.

ARM Specific Benchmarking Commands

The **BMC** (**B**ench**M**ark **C**ounter) commands provide control of counters in the data watchpoint and trace unit (DWT). The DWT can generate statistics on the operation of the processor and the memory system.

The counters of the DWT can be read at run-time, but the limited counter size (8-bit) leads to quick counter overflows. A meaningful benchmarking analysis is possible if you let the ITM emit a trace packet each time the counter overflows.

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

For information about *architecture-specific* **BMC** commands, see command descriptions below.

BMC.OFF — Disable benchmark counters

[build 151054 - DVD 02/2023]

Format:	**BMC.OFF**

Disables TRACE32 BMC functionality.

BMC.ON — Enable benchmark counters

[build 151054 - DVD 02/2023]

Format:	**BMC.ON**

Enables TRACE32 BMC functionality.

BMC.SELect — Select counter for statistic analysis

Format:	**BMC.SELect** *<counter>*
<counter>:	**CPI** | **EXC** | **SLP** | **LSU** | **FLD** | **CYC**

The exported event counter values can be combined with the exported instruction flow in order to get a clearer understanding of the program behavior. The command **BMC.SELect** allows to specify which counter is combined with the instruction flow to get a statistical evaluation.

Please refer to **"BenchMarkCounter"**, page 32 for information about the different counters.

Format:	**BMC.Trace** [**ON** | **OFF**]

Default: OFF.

Switches the ITM on in order to output the benchmark counter values through the instrumentation trace.

ARM specific TrOnchip Commands

Deprecated vs. New Commands

NOTE:	A number of commands from the **TrOnchip** command group have been renamed to **Break.CONFIG.***<sub_cmd>*. In addition, these **Break.CONFIG** commands are now *architecture-independent* commands, and as such they have been moved to general_ref_b.pdf.

Previously in this manual:	Now in general_ref_b.pdf:
TrOnchip.CONVert (deprecated)	**Break.CONFIG.InexactAddress**
TrOnchip.MatchASID (deprecated)	**Break.CONFIG.MatchASID**
TrOnchip.VarCONVert (deprecated)	**Break.CONFIG.VarConvert**

For information about *architecture-specific* **TrOnchip** commands, refer to the command descriptions in this chapter.

TrOnchip.state

Display on-chip trigger window

Format:	**TrOnchip.state**

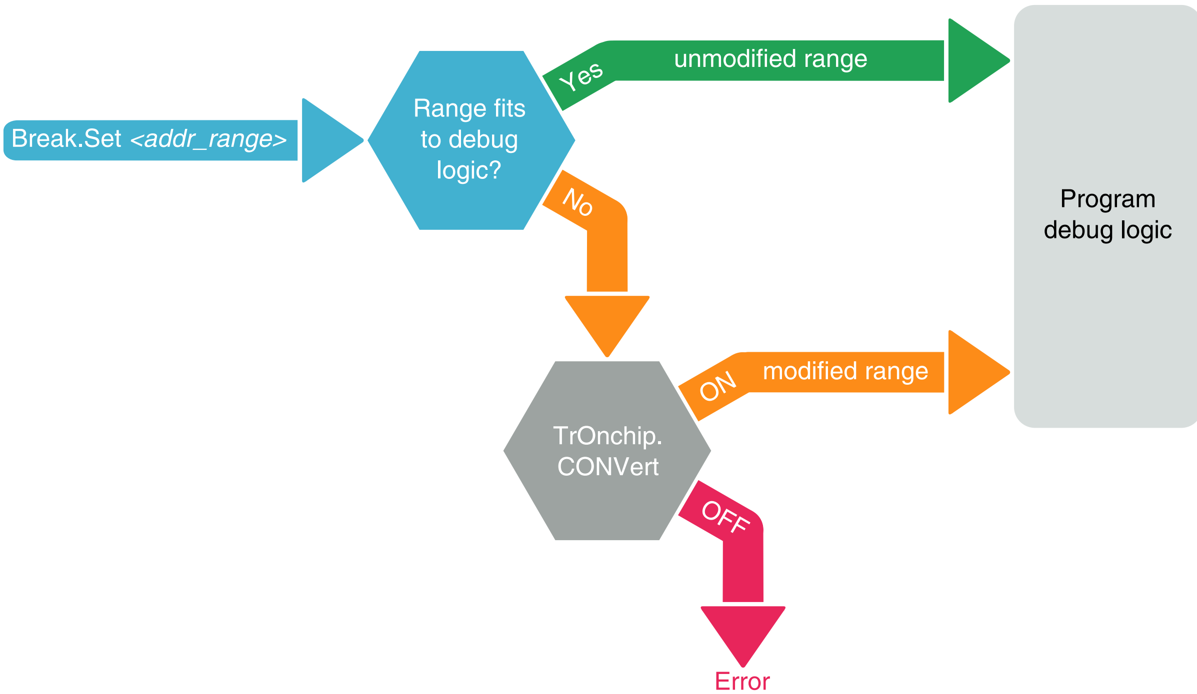
Opens the **TrOnchip.state** window.

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

OFF (default)	Stop the program execution at on-chip breakpoint if the address matches. Trace filters and triggers become active if the address matches.
ON	Stop the program execution at on-chip breakpoint if both the address and the ASID match. Trace filters and triggers become active if both the address and the ASID match.

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

Controls for all on-chip read/write breakpoints whether the debugger is allowed to change the user-defined address range of a breakpoint (see **Break.Set** *<address_range>* in the figure below).



The debug logic of a processor may be implemented in one of the following three ways:

1. The debug logic does not allow to set range breakpoints, but only single address breakpoints. Consequently the debugger cannot set range breakpoints and returns an error message.

2. The debugger can set any user-defined range breakpoint because the debug logic accepts this range breakpoint.

3. The debug logic accepts only certain range breakpoints. The debugger calculates the range that comes closest to the user-defined breakpoint range (see “modified range” in the figure above).

The **TrOnchip.CONVert** command covers case 3. For case 3) the user may decide whether the debugger is allowed to change the user-defined address range of a breakpoint or not by setting **TrOnchip.CONVert** to **ON** or **OFF**.

ON (default)	If **TrOnchip.Convert** is set to **ON** and a breakpoint is set to a range which cannot be exactly implemented, this range is automatically extended to the next possible range. In most cases, the breakpoint now marks a wider address range (see “modified range” in the figure above).
OFF	If **TrOnchip.Convert** is set to **OFF**, the debugger will only accept breakpoints which exactly fit to the debug logic (see “unmodified range” in the figure above). If the user enters an address range that does not fit to the debug logic, an error will be returned by the debugger.

In the **Break.List** window, you can view the requested address range for all breakpoints, whereas in the **Break.List /Onchip** window you can view the actual address range used for the on-chip breakpoints.

TrOnchip.RESERVE — Reserve on-chip breakpoint comparators

Format: **TrOnchip.RESERVE FP**<*x*> [**ON** | **OFF**]

Reserve on-chip breakpoint comparators to be used by the target application.

ON	The on-chip breakpoint is used by the target application.
OFF (default)	The on-chip breakpoint can be used by the debugger.

TrOnchip.RESet — Reset on-chip trigger settings

Format: **TrOnchip.RESet**

Resets all TrOnchip settings.

Format:	**TrOnchip.Set** *<exception>* [**ON** | **OFF**]
<exception>:	**SFERR** **HARDERR** **INTERR** **BUSERR** **STATERR** **CHKERR** **NOCPERR** **MMERR** **CORERESET**

Default: ON.

Sets/resets the corresponding bits for vector catching. The bit causes a debug entry when the specified vector is committed for execution. The availability of the vector catch type depends on the core type.

SFERR	Debug trap on secure fault.
HARDERR	Debug trap on hard fault.
INTERR	Debug trap on interrupt/exception service errors. These are a subset of other faults and catches before BUSERR or HARDERR.
BUSERR	Debug trap on normal bus error.
STATERR	Debug trap on usage fault state errors.
CHKERR	Debug trap on usage fault enabled checking errors.
NOCPERR	Debug trap on usage fault access to coprocessor which is not present or marked as not present in CAR register.
MMERR	Debug trap on memory management faults.
CORERESET	Reset vector catch. Halt running system if core reset occurs.

TrOnchip.StepVector — Step into exception handler

Format:	**TrOnchip.StepVector** [**ON** | **OFF**]

Default: OFF.

ON	Step into exception handler.
OFF	Step over exception handler.

TrOnchip.StepVectorResume — Catch exceptions and resume single step

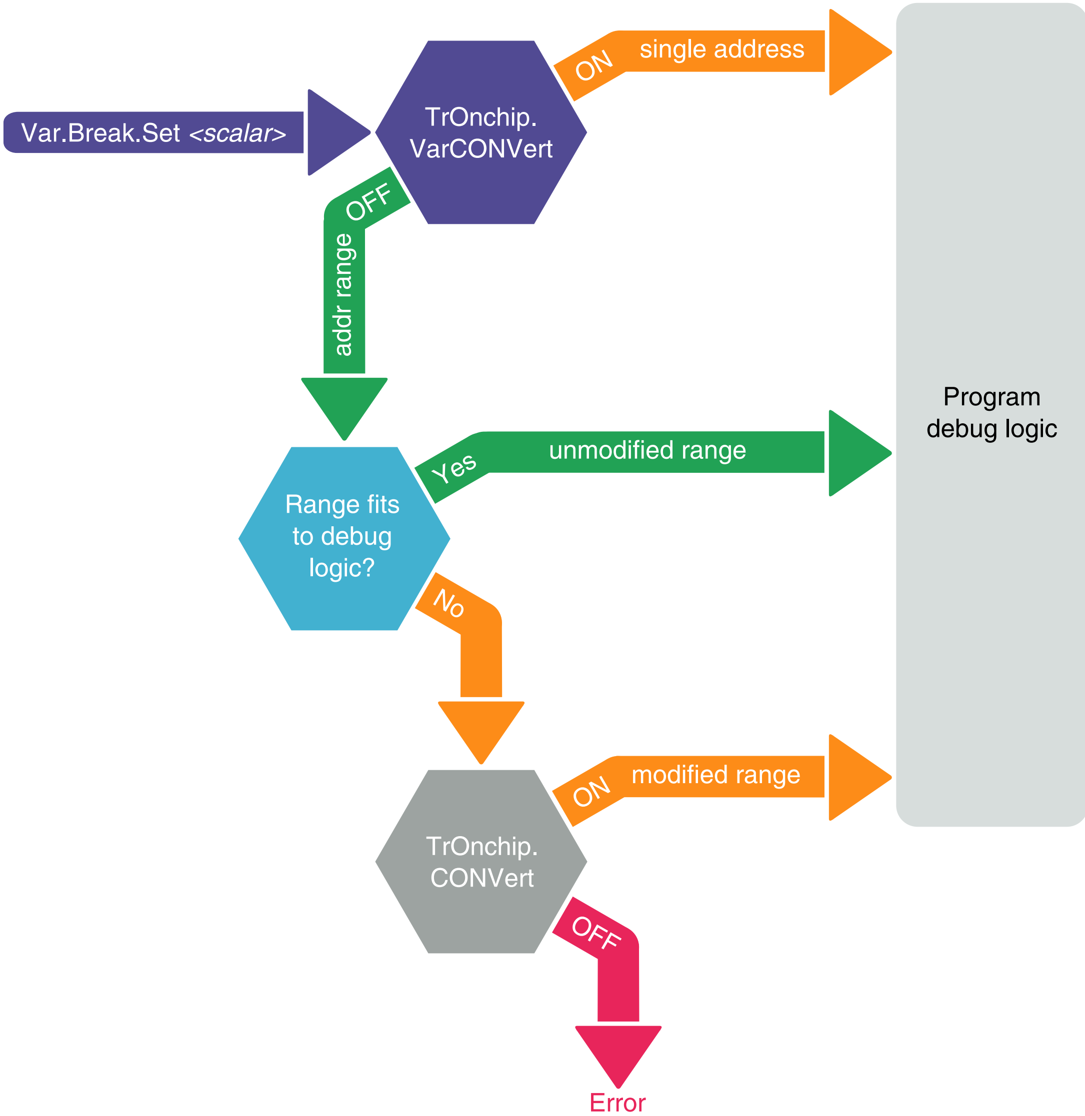
Format:	**TrOnchip.StepVector** [**ON** | **OFF**]

Default: OFF.

When this command is set to ON, the debugger will catch exceptions and resume the single step.

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

Controls for all scalar variables whether the debugger sets an HLL breakpoint with **Var.Break.Set** only on the start address of the scalar variable or on the entire address range covered by this scalar variable.



ON	If **TrOnchip.VarCONVert** is set to **ON** and a breakpoint is set to a scalar variable (int, float, double), then the breakpoint is set only to the start address of the scalar variable. • Allocates only one single on-chip breakpoint resource. • Program will not stop on accesses to the variable's address space.
OFF (default)	If **TrOnchip.VarCONVert** is set to **OFF** and a breakpoint is set to a scalar variable (int, float, double), then the breakpoint is set to the entire address range that stores the scalar variable value. • The program execution stops also on any unintentional accesses to the variable's address space. • Allocates **up to two** on-chip breakpoint resources for a single range breakpoint. **NOTE**: The address range of the scalar variable may not fit to the debug logic and has to be converted by the debugger, see **TrOnchip.CONVert**.

In the **Break.List** window, you can view the requested address range for all breakpoints, whereas in the **Break.List /Onchip** window you can view the actual address range used for the on-chip breakpoints.

JTAG Connection

Pinout of the 20-pin Debug Cable:

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	VSUPPLY (not used)
TRST-	3	4	GND
TDI	5	6	GND
TMS|TMSC|SWDIO	7	8	GND
TCK|TCKC|SWCLK	9	10	GND
RTCK	11	12	GND
TDO|-|SWO	13	14	GND
RESET-	15	16	GND
DBGRQ	17	18	GND
DBGACK	19	20	GND

For details on logical functionality, physical connector, alternative connectors, electrical characteristics, timing behavior and printing circuit design hints refer to **"Arm Debug and Trace Interface Specification"** (app_arm_target_interface.pdf).