LAUTERBACH
DEVELOPMENT TOOLS

Arm Debugger

Release 02.2025

MANUAL

Arm Debugger

Arm Debugger

Version 13-Feb-2025

History

06-Mar-2024	New command SYStem.Option.SLaVeSOFTRESet.
19-Aug-2022	Link to manual **XCP Debug Back-End** added in chapter' Brief Overview of Documents for New Users'.
15-Jun-2022	New subchapter 'XCP Specific Commands', describes the XCP subcommands of SYStem.CONFIG.

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 functions for Cortex-A/R (Armv7, 32-bit), as well as for the legacy architectures Arm7, Arm9 and Arm11.

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

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Debugger Tutorial"** (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.
- **"OS Awareness Manuals"** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

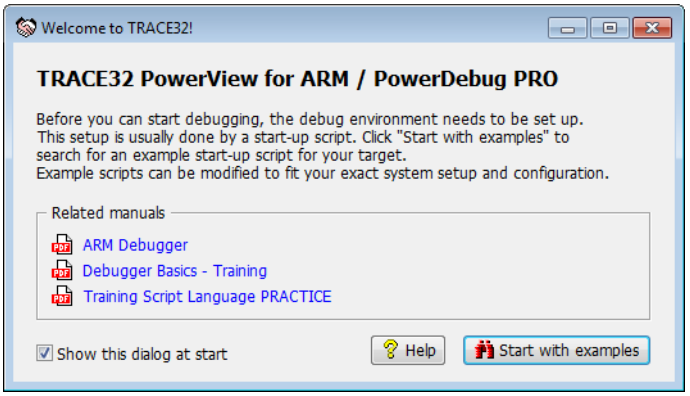
Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:

  - Choose **Help** menu > **Processor Architecture Manual**.

- 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).
- This manual does not cover the Cortex-M processor architecture. If you are using this processor architecture, please refer to **"Cortex-M Debugger"** (debugger_cortexm.pdf) for details.
- **"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**):



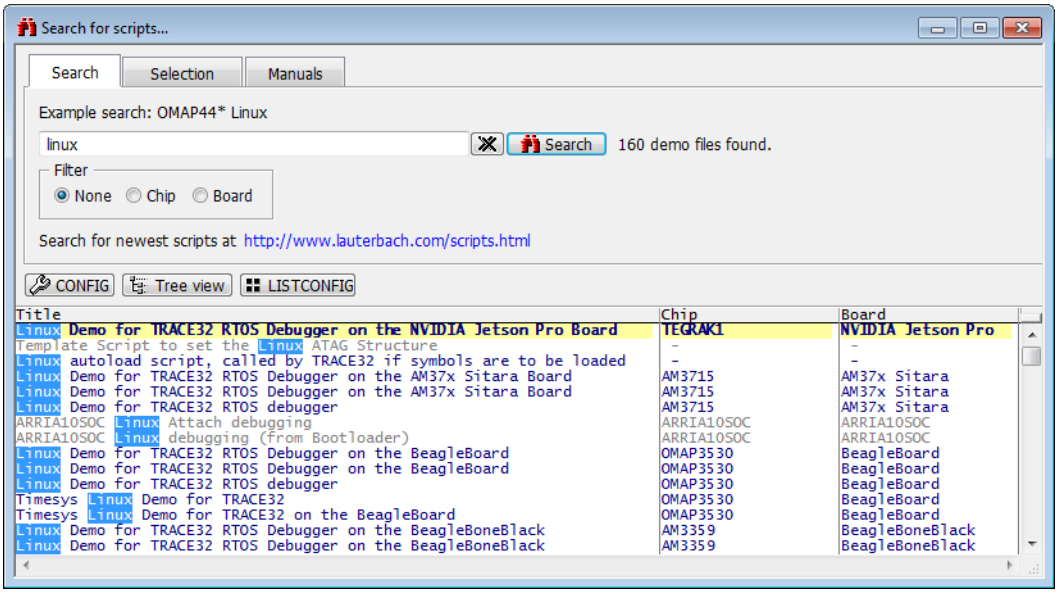
Demo and Start-up Scripts

Lauterbach provides ready-to-run PRACTICE start-up scripts for public known architecture 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.

Starting up the debugger is done as follows:

1. Reset the debugger.

```
RESet
```

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

2. Select the chip or core you intend to debug.

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

Based on the selected chip the debugger sets the **SYStem.CONFIG** and **SYStem.Option** commands the way which should be most appropriate for debugging this chip. Ideally no further setup is required.

If you select a Cortex-A or Cortex-R core instead of a chip (e.g. “SYStem.CPU CortexR4”) then you need to specify the base address of the debug register block:

```
SYStem.CONFIG.CCOREDEBUG.Base <address>
```

3. Connect to target.

```
SYStem.Up
```

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

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

```
SYStem.Mode Attach
```

instead. A “Break” will halt the processor.

4. Load the program you want to debug.

```
Data.LOAD armle.axf
```

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

An example of a 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 ARM940T                ; Select the core type

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

SYStem.Up                         ; Reset the target and enter debug mode

Data.LOAD armle.axf               ; Load the application

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

Register.Set r13 0x8000           ; Set the stack pointer to address 8000

PER.view                          ; Show clearly arranged peripherals
                                  ; in window                      *)

List.Mix                          ; Open source code window        *)

Register.view /SpotLight          ; Open register window           *)

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

Var.Watch var1 var2               ; Open watch window for variables *)

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

Break.Set 0x101000 /Program       ; 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.

FAQ

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

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 time out” 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 view 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 target 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.
- Your core needs adaptive clocking. Use the RTCK mode: **SYStem.JtagClock RTCK**.
- The core is used in a multicore system and the appropriate multicore settings for the debugger are missing. See for example **SYStem.CONFIG.IRPRE**. This is the case if you get a value IR_Width > 5 when you enter “DIAG 3400” and “AREA”. If you get IR_Width = 4 (Arm7, Arm9, Cortex) or IR_Width = 5 (Arm11), then you have just your core 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\hardware if there is an suitable script file *.cmm for your target.

Trace Extensions

There are two types of trace extensions available on the Arm:

- **Arm-ETM:** an Embedded Trace Macrocell or Program Trace Macrocell is integrated into the core. The Embedded Trace Macrocell provides program and data flow information plus trigger and filter features. The Program Trace Macrocell provide similar features but no data trace. The TRACE32 does not distinguish between ETM and PTM. The **ETM** command group is used for both.

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

  Please note that in case of CoreSight ETM/PTM you need to inform the debugger about the CoreSight trace system on the chip. If you can select the chip you are using (e.g. 'SYStem.CPU OMAP4430') then this is automatically done. If you select a core (e.g. 'SYStem.CPU CortexA9') then you need to configure the debugger in your start-up script by using commands like:

  - **SYStem.CONFIG.ETM.Base**
  - **SYStem.CONFIG.FUNNEL.Base**
  - **SYStem.CONFIG.TPIU.Base**
  - **SYStem.CONFIG.FUNNEL.ATBSource**
  - **SYStem.CONFIG.TPIU.ATBSource**

  In case a HTM or ITM/STM module is available and shall be used you need also settings for that.

- **Arm7 Bus Trace:** the Preprocessor for Arm7 family samples the external address and data bus. The features for the Bus Trace are described in this book.

  The commands for the Arm7 bus trace are:

  - **SYStem.Option.AMBA**
  - **SYStem.Option.NODATA**
  - **TrOnchip.TEnable** and **TrOnchip.TCYcle**

Symmetric Multiprocessing

A multi-core system used for **Asymmetric Multiprocessing (AMP)** has specialized cores which are used for specific tasks. To debug such a system you need to open separate TRACE32 graphical user interfaces (GUI) one for each core. On each GUI you debug the application which is assigned to this core and will never be executed on another core. The GUIs can be synchronized regarding program start and halt in order to debug the cores interaction.

ARM11 MPCore and Cortex-A9 MPCore are examples for multi-core architectures which allow **Symmetric Multiprocessing (SMP)**. The included cores of identical type are connected to a single shared main memory. Typically a proper SMP real-time operating system assigns the tasks to the cores. You will not know on which core the task you are interested in will be executed.

To debug an SMP system, you need to start only one TRACE32 PowerView GUI.

The selection of the proper SMP chip (e.g. 'CNS3420' or 'OMAP4430') causes the debugger to connect to all included SMP-able cores on start-up (e.g. by 'SYStem.Up'). If you have an SMP-able core type selected (e.g. 'ARM11MPCore' or 'CortexA9MPCore') you need to specify the number of cores you intend to SMP-debug by **SYStem.CONFIG CoreNumber** *<number>*.

On a selected SMP chip (e.g. 'CNS3420' or 'OMAP4430') the CONFIG parameters of all cores are typically known by the debugger. For an SMP-able core type you need to set them yourself (e.g. DAPIRPRE, COREDEBUG.Base, ...). Where needed multiple parameters are possible (e.g. 'SYStem.CONFIG.COREDEBUG.Base 0x80001000 0x80003000'.

System options and selected JTAG clock affect all cores.

All cores will be started, stepped and halted together. An exception is the assembler single-step which will affect only one core.

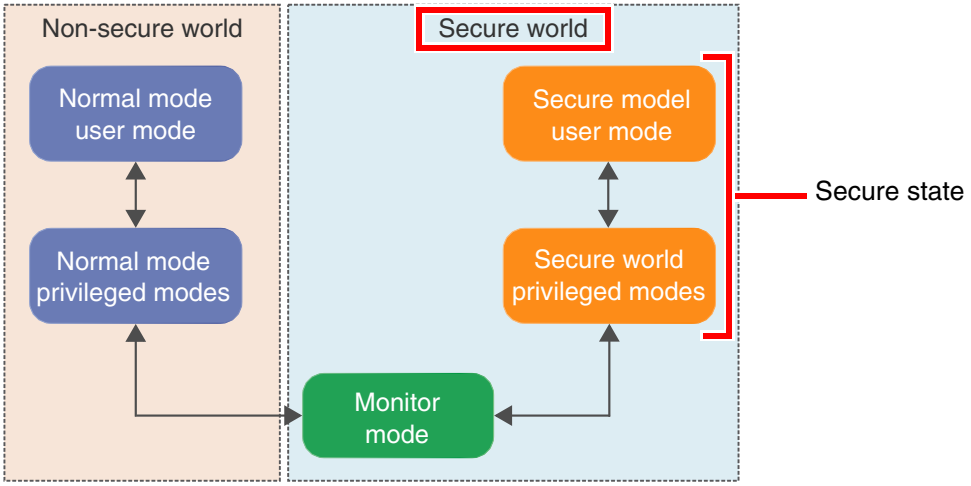
TRACE32 takes care that software and on-chip breakpoints will have effect on whatever core the task will run.

When the task halts, e.g. due to a breakpoint hit, the TRACE32 PowerView GUI shows the core on which the debug event has happened. The core number is shown in the state line at the bottom of the main window. You can switch the GUIs perspective to the other cores when you right-click on the core number there. Alternatively you can use the command **CORE.select** *<number>*.

TrustZone Technology

The Cortex-A and ARM1176 processor integrate Arm's TrustZone technology, a hardware security extension, to facilitate the development of secure applications.

It splits the computing environment into two isolated worlds. Most of the code runs in the 'non-secure' world, whereas trusted code runs in the 'secure' world. There are core operations that allow you to switch between the secure and non-secure world. For switching purposes, TrustZone introduces a new secure 'monitor' mode. Reset enters the secure world:



Only when the core is in the secure world, core and debugger can access the secure memory. There are some CP15 registers accessible in secure state only, and there are banked CP15 registers, with both secure and non-secure versions.

Debug Permission

Debugging is strictly controlled. It can be enabled or disabled by the SPIDEN (Secure Privileged Invasive Debug Enable) input signal and SUIDEN (Secure User Invasive Debug Enable) bit in SDER (Secure Debug Enable Register):

- SPIDEN=0, SUIDEN=0: debug in non-secure world, only
- SPIDEN=0, SUIDEN=1: debug in non-secure world and secure user mode
- SPIDEN=1: debug in non-secure and secure world

SPIDEN is a chip internal signal and it's level can normally not be changed. The SUIDEN bit can be changed in secure privileged mode, only.

Debug mode can not be entered in a mode where debugging is not allowed. Breakpoints will not work there. A **Break** command or a **SYStem.Up** will work the moment a mode is entered where debugging is allowed.

Checking Debug Permission

The DBGDSCR (Debug Status and Control Register) bit 16 shows the signal level of SPIDEN. In the SDER (Secure Debug Enable Register) you can see the SUIDEN flag assuming you are in the secure state which allows reading the SDER register.

Checking Secure State

In the peripheral file, the DBGDSCR register bit 18 (NS) shows the current secure state. You can also see it in the **Register.view** window if you scroll down a bit. On the left side you will see 'sec' which means the core is in the secure state, 'nsec' means the core is in non-secure state. Both reflect the bit 0 (NS) of the SCR (Secure Control Register). However SCR is only accessible in secure state.

In monitor mode, which is also indicated in the **Register.view** window, the core is always in secure state independent of the NS bit (non-secure bit) described above. However, in monitor mode, you can access the secure CP15 register if NS=secure. And you can access the non-secure CP15 register if NS=non-secure.

Changing the Secure State from within TRACE32

From the TRACE32 PowerView GUI, you can switch between secure mode (0) and non-secure mode (1) by toggling the 'sec', 'nsec' indicator in the **Register.view** window or by executing this command:

```
Register.Set NS 0    ;secure mode
Register.Set NS 1    ;non-secure mode
```

It sets or clears the NS (Non-Secure) bit in the SCR register. You will get a 'emulator function blocked by device security' message in case you are trying to switch to secure mode although debugging is not allowed in secure mode.

This way you can also inspect the register of the other world. Please note that a change in state affects program execution. Remember to set the bit back to its original value before continuing the application program.

Accessing Memory

If you do not specify otherwise, the debugger shows you the memory of the secure state the core is currently in.

- The access class 'Z:' indicates secure mode ('Z' -> trustZone, 'S' -> Supervisor)
- The access class 'N:' indicates non-secure mode.

By preceding an address with the 'Z:' and 'N:' access class, you can force a certain memory view for all memory operations.

Accessing Coprocessor CP15 Register

The peripheral file and 'C15:' access class will show you the CP15 register bank of the secure mode the core is currently in. When you try to access registers in non-secure world which are accessible in secure world only, the debugger will show you '????????'.

You can force to see the other bank by using access class "ZC15:" for secure, "NC15:" for non-secure respectively.

Accessing Cache and TLB Contents

Reading cache and TLB (Translation Look-aside Buffer) contents is only possible if the debugger is allowed to debug in secure state. You get a 'function blocked by device security' message otherwise.

However, a lot of devices do not provide this debug feature at all. Then you get the message 'function not supported by this device'.

Vector Catch Register and Secure Modes

Vector catch debug events (**TrOnchip.Set** …) can individually be activated for secure state, non-secure state, and monitor mode.

Breakpoints and Secure Modes

The security concept of the Arm architecture allows to specify breakpoints that cause a halt event only for a certain secure mode (secure/non-secure/hypervisor).

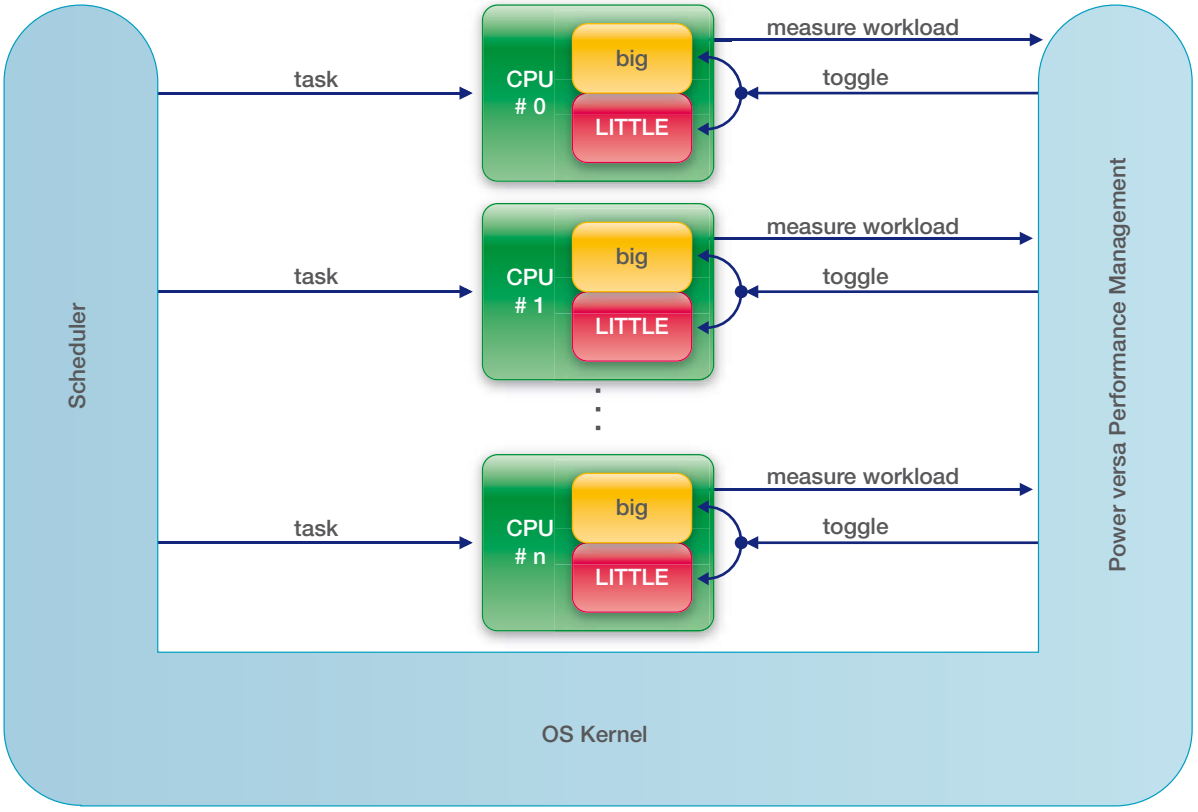
Software breakpoints will be set in secure or non-secure memory depending on the current secure mode of the core. Alternatively, software breakpoints can explicitly be placed in a certain secure mode by preceding an address with the access class "Z:" (secure) or "N:" (non-secure).

On-chip breakpoints will halt the core in any secure mode by default. **Break.CONFIG.MatchZone ON** enables the breakpoints to be conditional on the security state of the processor.

Please refer to the chapter about **secure, non-secure and hypervisor breakpoints** to get additional information.

big.LITTLE

Arm big.LITTLE processing is an energy savings method where high-performance cores get paired together in a cache-coherent combination. Software execution will dynamically be transitioned between these cores depending on performance needs.



The OS kernel scheduler sees each pair as a single virtual core. The big.LITTLE software works as an extension to the power-versa-performance management. It can switch the execution context between the big and the LITTLE core.

Qualified for pairing is Cortex-A15 (as 'big') and Cortex-A7 (as 'LITTLE').

Debugger Setup

Example for a symmetric big.LITTLE configuration (2 Cortex-A15, 2 Cortex-A7):

```
SYStem.CPU CORTEXA15A7
SYStem.CONFIG CoreNumber 4.
CORE.ASSIGN BIGLITTLE 1. 2. 3. 4.
SYStem.CONFIG.COREDEBUG.Base <CA15_1> <CA7_2> <CA15_3> <CA7_4>
```

Example for a non-symmetric big.LITTLE configuration (1 Cortex-A15, 2 Cortex-A7):

```
SYStem.CPU CORTEXA15A7
SYStem.CONFIG CoreNumber 4.
CORE.ASSIGN BIGLITTLE 1. 2. NONE 4.
SYStem.CONFIG.COREDEBUG.Base <CA15_1> <CA7_2> <dummy_3> <CA7_4>
```

Consequence for Debugging

The shown core numbers are extended by 'b' = 'big' or 'l' = 'LITTLE'.

The core status (active or powered down) can be checked with **CORE.SHOWACTIVE** or in the state line of the TRACE32 main window, where you can switch between the cores.

The debugger assumes that one core of the pair is inactive.

The OS Awareness sees each pair as one virtual core.

The peripheral file respects the core type (Cortex-A15 or Cortex-A7).

Requirements for the Target Software

The routine (OS on target) which switches between the cores needs to take care of (copying) transferring the on-chip debug settings to the core which wakes up.

This needs also to be done when waking up a core pair. In this case you copy the settings from an already active core.

big.LITTLE MP

Another logical use-model is ('MP' = Multi-Processing). It allows both the big and the LITTLE core to be powered on and to simultaneously execute code.

From the debuggers point of view, this is not a big.LITTLE system in the narrow sense. There are no pairs of cores. It is handled like a normal multicore system but with mixed core types.

Therefore for the setup, we need **SYStem.CPU CORTEXA15A7**, but we use **CORE.ASSIGN** instead of **CORE.ASSIGN BIGLITTLE**.

Example for a symmetric big.LITTLE MP configuration (2 Cortex-A15, 2 Cortex-A7):

```
SYStem.CPU CORTEXA15A7
SYStem.CONFIG CoreNumber 4.
CORE.ASSIGN 1. 2. 3. 4.
SYStem.CONFIG.COREDEBUG.Base <CA15_1> <CA7_2> <CA15_3> <CA7_4>
```

Breakpoints

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is patched by a breakpoint code.

While software breakpoints are used one of the two ICE breaker units is programmed with the breakpoint code (on Arm7 and Arm9, except ARM9E variants). This means whenever a software breakpoint is set only one ICE unit breakpoint is remaining for other purposes. There is no restriction in the number of software breakpoints.

On-chip Breakpoints for Instructions

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the CPU. For the Arm architecture the on-chip breakpoints are provided by the “ICEbreaker” unit. on-chip breakpoints are usually needed for instructions in FLASH/ROM.

With the command **MAP.BOnchip** *<range>* it is possible to tell the debugger where you have ROM / FLASH on the target. If a breakpoint is set into a location mapped as BOnchip one ICEbreaker unit is automatically programmed.

On-chip Breakpoints for Data

To stop the CPU after a read or write access to a memory location on-chip breakpoints are required. In the Arm notation these breakpoints are called watchpoints. A watchband may use one or two ICEbreaker units.

The number of on-chip breakpoints for data accesses can be extended by using the ETM Address and Data comparators. Refer to **ETM.StoppingBreakPoints**.

Overview

- **On-chip breakpoints:** Total amount of available on-chip breakpoints.
- **Instruction breakpoints:** Number of on-chip breakpoints that can be used to set program breakpoints into ROM/FLASH/EPROM.
- **Read/Write breakpoints:** Number of on-chip breakpoints that can be used as Read or Write breakpoints.
- **Data breakpoint:** Number of on-chip data breakpoints that can be used to stop the program when a specific data value is written to an address or when a specific data value is read from an address.

	Program Breakpoints	Read/Write Breakpoints	Data Value Breakpoints
ARM7 **ARM9**	**Onchip breakpoints:** up to 2, but address range only as bit mask (Reduced to 1 if software breakpoints are used) **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** up to 2, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip Breakpoint:** up to 2, but address range only as bit mask **ETM breakpoints:** up to 2 data value breakpoints for exact address ranges
ARM11	**Onchip breakpoints:** 6, but only single addresses **ETM breakpoints:** up to 2 exact address ranges possible	**Onchip breakpoints:** 2, but only single addresses **ETM breakpoints:** up to 2 exact address ranges possible	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** up to 2 data value breakpoints for exact address ranges
Cortex-A5	**Onchip breakpoints:** 3, but only single addresses **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** 2, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** up to 2 data value breakpoints for exact address ranges
Cortex-A7 **Cortex-R7**	**Onchip breakpoints:** 6, but only single addresses **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** 4, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** up to 2 data value breakpoints for exact address ranges
Cortex-A8	**Onchip breakpoints:** 6, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** 2, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** up to 2 data value breakpoints for exact address ranges

	Program Breakpoints	Read/Write Breakpoints	Data Value Breakpoints
Cortex-R4 **Cortex-R5**	**Onchip breakpoints:** 2..8, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** 1..8, but address range only as bit mask **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** up to 2 data value breakpoints for exact address ranges
Cortex-A9 **Cortex-A15** **Cortex-A17**	**Onchip breakpoints:** 6, but only single addresses **ETM breakpoints:** 2 exact address ranges	**Onchip breakpoints:** 4, but address range only as bit mask **ETM breakpoints:** —	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** —

Hardware Breakpoints (Bus Trace only)

When a Preprocessor for Arm7 family is used, hardware breakpoints are available to filter the trace information. Refer to **TrOnchip.TEnable** for more information.

If a hardware breakpoint is used the resources to set the breakpoint are provided by the TRACE32 development tool.

Example for Standard Breakpoints

Assume you have a target with

- FLASH from `0x0--0xfffff`
- RAM from `0x100000--0x11ffff`

The command to configure TRACE32 correctly for this configuration is:

```
MAP.BOnchip 0x0--0xfffff
```

The following standard breakpoint combinations are possible.

1. Unlimited breakpoints in RAM and one breakpoint in ROM/FLASH

```
Break.Set 0x100000 /Program          ; Software breakpoint 1
Break.Set 0x101000 /Program          ; Software breakpoint 2
Break.Set addr /Program              ; Software breakpoint 3
Break.Set 0x100 /Program             ; On-chip breakpoint
```

2. Unlimited breakpoints in RAM and one breakpoint on a read or write access

```
Break.Set 0x100000 /Program          ; Software breakpoint 1
Break.Set 0x101000 /Program          ; Software breakpoint 2
Break.Set addr /Program              ; Software breakpoint 3
Break.Set 0x108000 /Write            ; On-chip breakpoint
```

3. Two breakpoints in ROM/FLASH

```
Break.Set 0x100 /Program             ; On-chip breakpoint 1
Break.Set 0x200 /Program             ; On-chip breakpoint 2
```

4. Two breakpoints on a read or write access

```
Break.Set 0x108000 /Write            ; On-chip breakpoint 1
Break.Set 0x108010 /Read             ; On-chip breakpoint 2
```

5. One breakpoint in ROM/FLASH and one breakpoint on a read or write access

```
Break.Set 0x100 /Program              ; On-chip breakpoint 1

Break.Set 0x108010 /Read              ; On-chip breakpoint 2
```

TRACE32 will set any breakpoint to work in any secure and non-secure mode. As of build 59483, TRACE32 distinguishes between secure, non-secure, and hypervisor breakpoints. The support for these kinds of breakpoints is disabled per default, i.e. all breakpoints are set for all secure/non-secure modes.

Enable and Use Secure, Non-Secure and Hypervisor Breakpoints

To make use of this feature, you have to enable the symbol management for Arm zones first with the **SYStem.Option.ZoneSPACES** command:

```
SYStem.Option.ZoneSPACES ON     ; enable symbol management for Arm zones
```

Usually TRACE32 will then set the secure/non-secure breakpoint automatically if it has enough information about the secure/non-secure properties of the loaded application and its symbols. This means the user has to tell TRACE32 if a program code runs in secure/non-secure or hypervisor mode when the code is loaded:

```
Data.LOAD.ELF armf Z:    ; Load application, symbols for secure mode

Data.LOAD.ELF armf N:    ; Load application, symbols for non-secure mode

Data.LOAD.ELF armf H:    ; Load application, symbols for hypervisor mode
```

Please refer to the **SYStem.Option.ZoneSPACES** command for additional code loading examples.

Now breakpoints can be uses as usual, i.e. TRACE32 will automatically take care of the secure type when a breakpoint is set. This depends on how the application/symbols were loaded:

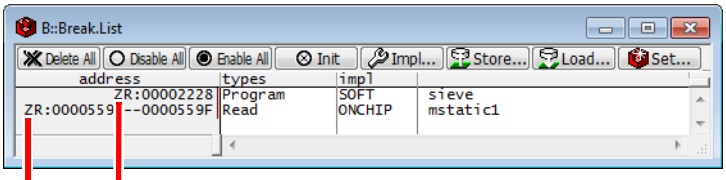
```
Break.Set main            ; Set breakpoint on main() function, Z:, N: or
                          ; H: access class is automatically set

Var.Break.Set struct1     ; Set Read/Write breakpoints to the whole
                          ; structure struct1. The breakpoint is either
                          ; a secure/non-secure or hypervisor type.
```

Example 1 - Load Secure Application and Set Breakpoints

```
SYStem.Option.ZoneSPACES ON    ; Enable symbol management

// Load demo application and tell TRACE32 that it is secure
Data.LOAD.ELF ~~/demo/arm/compiler/arm/armle.axf Z:

// Set a breakpoint on the sieve() function start
Break.Set sieve

// Set a read breakpoint to the global variable mstatic1
Var.Break.Set mstatic1 /Read

Break.List                     ; Show breakpoints
```

First the symbol management is enabled. An application is loaded and TRACE32 is advised by the access class “Z:” at the end of the **Data.LOAD.Elf** command that this application runs in secure mode.

As a next step, two breakpoints are set but the user does not need to care about any access classes. The **Break.List** window shows that the breakpoints are automatically configured to be of the secure type. This is shown by the “Z:” access class that is set at the beginning of the breakpoint addresses:



Secure breakpoint(s)

Set Breakpoints and Enforce Secure Mode

TRACE32 allows the user to specify whether a breakpoint should be set for secure, non-secure or hypervisor mode. This means the user has to specify an access class when the breakpoint is set:

```
Break.Set Z:main           ; Enforce secure breakpoint on main()

Break.Set N:main           ; Enforce non-secure breakpoint on main()

Break.Set H:main           ; Enforce hypervisor breakpoint on main()
```

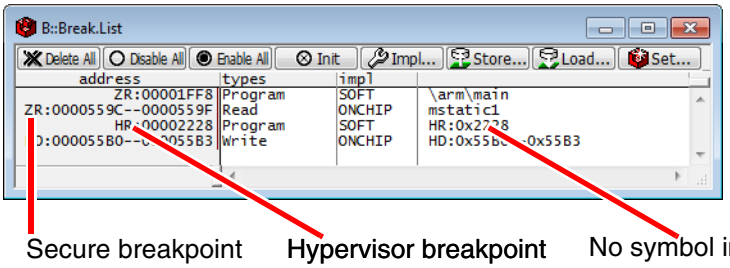
Breakpoints on variables need the variable name and the access class to be enclosed in round brackets:

```
Var.Break.Set (Z:struct1) ; Enforce secure read/write breakpoint

Var.Break.Set (N:struct1) ; Enforce non-secure read/write breakpoint

Var.Break.Set (H:struct1) ; Enforce hypervisor read/write breakpoint
```

```
SYStem.Option.ZoneSPACES ON     ; Enable symbol management

// Load demo application and tell TRACE32 that it is secure
Data.LOAD.ELF ~~/demo/arm/compiler/arm/armle.axf Z:

// Set secure breakpoint (auto-configured) on function main()
Break.Set main

// Explicitly set hypervisor breakpoint on function sieve()
Break.Set H:sieve

// Set secure read breakpoint (auto-configured) on variable mstatic1
Var.Break.Set mstatic1 /Read

// Explicitly set hypervisor write breakpoint on variable vtdef1
Var.Break.Set (H:vtdef1) /Write

Break.List                      ; Show breakpoints
```

First, the symbol management is enabled. An application is loaded and TRACE32 is advised by the “Z:” at the end of the **Data.LOAD.Elf** command that this application runs in secure mode.

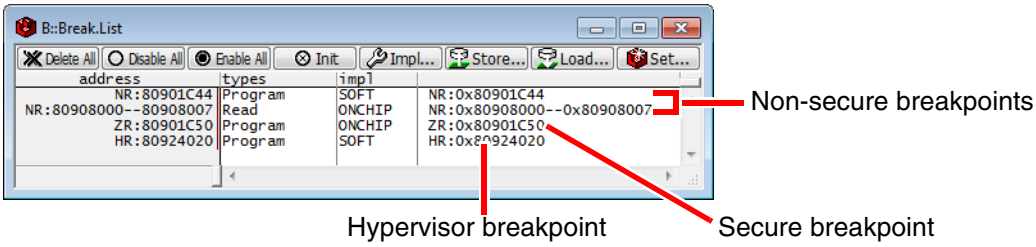
As a next step, four breakpoints are set. Two of them do not have any access class specified, so TRACE32 will use the symbol information to make it a secure breakpoint. The other two breakpoints are defined as hypervisor breakpoints using the “H:” access class. In this case the symbol information is explicitly overwritten. The **Break.List** now shows a mixed breakpoint setup:

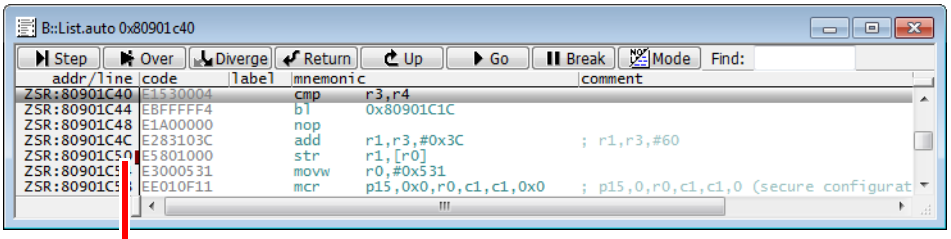


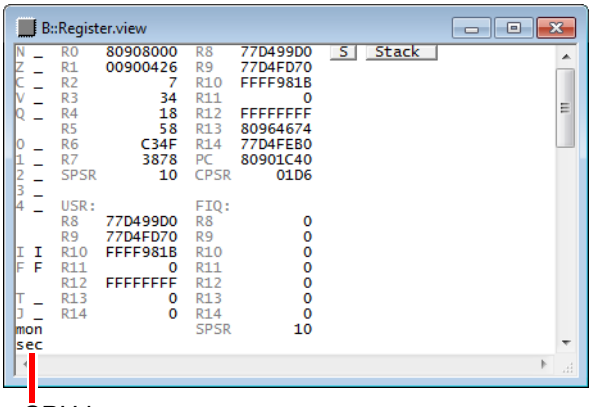
NOTE:	If a breakpoint is explicitly set in another mode, there might be no symbol information loaded for this mode. This means that the **Break.List** can only display the address of the breakpoint but not the corresponding symbol.

Summary of Breakpoint Configuration

TRACE32 can show you a summary of the set breakpoints in a **Break.List** window. Furthermore, which breakpoint will be active is also indicated in the **List.auto** window. A **Register.view** window will show you the current secure state of the CPU. This example uses only addresses and no symbols. The use of symbols is also possible as shown in Example 1 and Example 2:







NOTE:	The CPU might stop at a software breakpoint although there is not breakpoint shown in the **List.auto** window. This happens because all software breakpoints are always written at the given memory address.

Configuration of the Target CPU

The configuration of the onchip breakpoints will be placed in the breakpoint/watchpoint registers of the Arm CPU. If **Break.CONFIG.MatchZone** is **ON**, the debugger takes care of the correct values in the configuration register so that the breakpoint becomes only active when the CPU operates in the given secure/non-secure mode.

Complex Breakpoints

To use the advanced features of the ICE breaker unit the **TrOnchip** command group is possible. These commands provide full access to both ICE breaker units called A and B in the TRACE32 system. For an example of complex breakpoint usage please refer to the chapter **TrOnchip Example**. Most features can also be used by setting advanced breakpoints (e.g. task selective breakpoints, exclude breakpoints). Ranged breakpoints use multiple breakpoint resources to better fit the range when the resources are available.

Direct ICE Breaker Access

It is possible to program the complete ICE breaker unit directly, by using the access class ICE. E.g. the command `Data.Set ICE:10 %Long 12345678` writes the value 12345678 to the Watchpoint 1 Address Value Register. The following table lists the addresses of the relevant registers.

Address	Register
ICE:8	Watchpoint 0 Address Value
ICE:9	Watchpoint 0 Address Mask
ICE:0A	Watchpoint 0 Data Value
ICE:0B	Watchpoint 0 Data Mask
ICE:0C	Watchpoint 0 Control Value
ICE:0D	Watchpoint 0 Control Mask
ICE:10	Watchpoint 1 Address Value
ICE:11	Watchpoint 1 Address Mask
ICE:12	Watchpoint 1 Data Value
ICE:13	Watchpoint 1 Data Mask
ICE:14	Watchpoint 1 Control Value
ICE:15	Watchpoint 1 Control Mask

For more details please refer to the Arm data sheet. It is recommended to use the **Break.Set** or **TrOnchip** commands instead of direct programming, because then no special ICEbreaker knowledge is required.

Example for ETM Stopping Breakpoints

The default on-chip breakpoints either allow you to just set an instruction breakpoint on a single address or to apply a mask to get a rough range. In case of a mask, the given range is extended to the next range limits that fit the mask, i.e. the breakpoint may cover a wider address range than initially anticipated.

ETM stopping breakpoints allow you to set a true address range for instructions, i.e. the end and the start address of the breakpoint really match your expectations. This only works if the CPU provides an ETM with the necessary resources, e.g. the address comparators.

Prerequisites for ETM stopping breakpoints:

- Make sure that an ETM base address is configured. Otherwise TRACE32 will assume that there is no ETM.

```
SYStem.CONFIG ETM Base DAP:<etm_base>       ; Make ETM available
```

- If your CPU has its own CTI, it is recommended that you specify the CTI as well. Dependant on the specific core implementation, the CTI might be needed to receive the ETM stop events:

```
SYStem.CONFIG CTI Base DAP:<cti_base>
```

It's recommended to add both configuration commands to your PRACTICE start-up script (*.cmm). If you selected a known SoC, e.g. with **SYStem.CPU** *<cpu>*, these settings are already configured.

To set ETM stopping breakpoints:

1. Activate the ETM Stopping breakpoints support:

```
ETM.StoppingBreakpoints ON
```

2. Set the instruction range breakpoints, e.g.:

```
Break.Set func10                ; Set address range breakpoint on
                                ; the address range of function
                                ; func10

Break.Set 0xEC009008++0x58      ; Set address range breakpoint with
                                ; precise start and end address
```

The **Break.List** window provides an overview of all set breakpoints.

For more information, see **ETM.StoppingBreakPoints** in **"Arm ETM Trace"** (trace_arm_etm.pdf).

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:

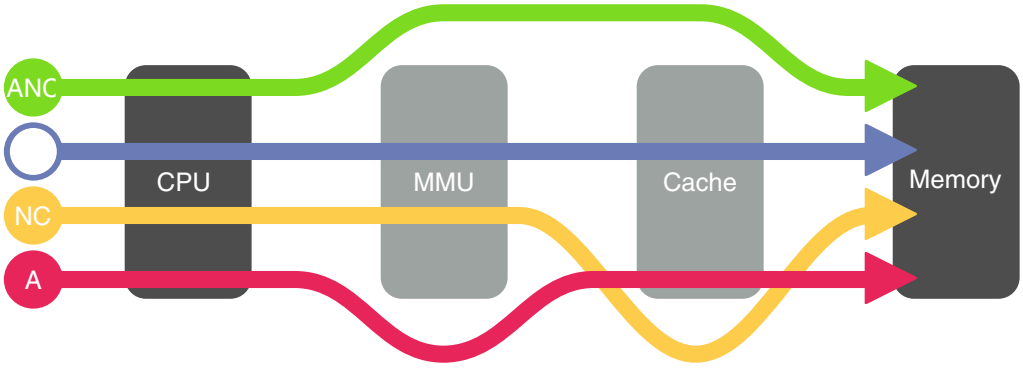
Description of the Individual Access Classes

Access Class	Description
A	Absolute addressing (physical address)
AHB, AHB2	See DAP description in this table.
APB, APB2	See DAP description in this table.
AXI, AXI2	See DAP description in this table.
C	"Current". Do not use this access class. It might be shown by the debugger if it is unknown what access class shall be used. Then the access class derives from the current processor mode.
C14	Access to C14-Coprocessor register. Its recommended to only use this in AArch32 mode.
C15	Access to C15-Coprocessor register. Its recommended to only use this in AArch32 mode.
D	Data Memory

Access Class	Description
DAP, DAP2, AHB, AHB2, ..., APB, APB2, ..., AXI, AXI2, ...	Memory access through bus masters that are called Memory Access Ports (MEM-AP) provided by a Debug Access Port (DAP). The DAP is a CoreSight component that is mandatory on Cortex-based devices. The MEM-APs are addressed and configured differently depending on whether you have a CoreSight SoC-400 or CoreSight SoC-600 based system on chip. Which bus master (MEM-AP) is used by which access class (e.g. AHB) is defined by assigning a MEM-AP number (SoC-400) or base address (SoC-600) to the access class: **CoreSight SoC-400:** SYStem.CONFIG.DEBUGAP1.Port *<mem_ap#>* -> “DAP” SYStem.CONFIG.AHBAP1.Port *<mem_ap#>* -> “AHB” SYStem.CONFIG.APBAP1.Port *<mem_ap#>* -> “APB” SYStem.CONFIG.AXIAP1.Port *<mem_ap#>* -> “AXI” **CoreSight SoC-600:** SYStem.CONFIG.AHBAP1.Base *<address>* -> “AHB” SYStem.CONFIG.APBAP1.Base *<address>* -> “APB” SYStem.CONFIG.AXIAP1.Base *<address>* -> “AXI” For an example, see **Configuration examples for memory access ports and a CoreSight component**. “DAP” should be the memory access port where the debug registers are located, which is typically an APB MEM-AP (AHB MEM-AP on a Cortex-M). This is because it is the default access class in debugger configurations (SYStem.CONFIG ...) if you use an address without access class. “DAP” is not available for a SoC-600 system. For a CoreSight SoC-400 system, a second set of access classes (DAP2, AHB2, APB2, AXI2) and configuration commands (e.g., SYStem.CONFIG.DEBUGAP2.Port *<mem_ap#>*) is available in case there are two DAPs that need to be controlled by the debugger. For a CoreSight SoC-600 system, more than two access classes of the same type are possible and configurable (APB, APB2, APB3, ...), all controlled by the same DAP.
DP	The Debug Port access class is used to address the first memory bus (APB) that you directly access with a SoC-600 DAPs debug interface (JTAG, SWD).
E	Run-time memory access (see **SYStem.MemAccess**)
M Armv8-A only	EL3 Mode (TrustZone devices). This access class only refers to the 64-bit EL3 mode. It does not refer to the 32-bit monitor mode. If an Armv8 based device is in 32-bit only mode, any entered “M” access class will be converted to a “ZS” access class.
O Armv9 only	EL3 Mode (Realm Management Extension). This access class only refers to the 64-bit EL3 mode (root access).

Access Class	Description
L Armv9 only	Realm access. This access class can be used with EL0, EL1, EL2 64-bit modes.
H	EL2/Hypervisor Mode (devices having Virtualization Extension)
I	Intermediate address. Available on devices having Virtualization Extension.
J	Java Code (8-bit)
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)
SPR Armv8/Armv9 only	Access to System Register, Special Purpose Registers and System Instructions. Its recommended to only use this in AArch64 mode.
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. The access class “NC” in the yellow path means “no cache”, so it will bypass the cache but not the MMU, i.e. a virtual access is happening.

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.

The blue path is an example of a virtual access which is done when no access class is specified.

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 32-bit Arm code. Furthermore, the access is translated by the MMU, so you have to pick the correct CPU mode to avoid a translation fail. In our example it should be necessary to access the memory in Arm supervisor mode. To ensure a secure access, use the access class specifier “Z”. To switch the CPU to supervisor mode during the access, use the access class specifier “S”. And to make the debugger disassemble the memory content as 32-bit Arm code use “R”. When you put all three access class specifiers together, you will obtain the access class combination “ZSR”.

```
List.Mix ZSR:0x10000000   // View 32-bit Arm 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.

```
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 “ZSR” again, it would not be possible to specify the access class combination as “SZR” or “RZS”, etc. 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.

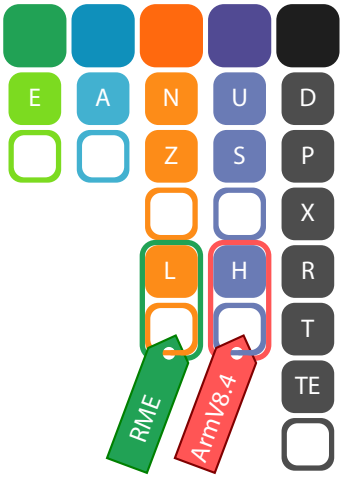
The illustrations below will show you how to combine access class specifiers for frequently-used access class combinations.

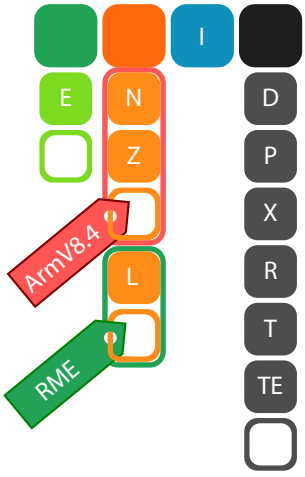
Rules to create a valid access class combination:

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

Memory Access through CPU (CPU View)

The debugger uses the CPU to access memory 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. Some options are only available since Armv8.4.





Example combinations:

AD	View physical data (current CPU mode)
AH	View physical data or program code while CPU is in hypervisor mode
ED	Access data at run-time
NUX	View A64 instruction code at non-secure virtual address location, e.g. code of the user application.
ZSD	View data in secure supervisor mode at virtual address location
AZHD	Physical secure hypervisor access. ArmV8.4-A only.
ZI	Secure intermediate access. ArmV8.4-A only.

Illegal access class combinations when ArmV8.4-A secure hypervisor is not implemented:

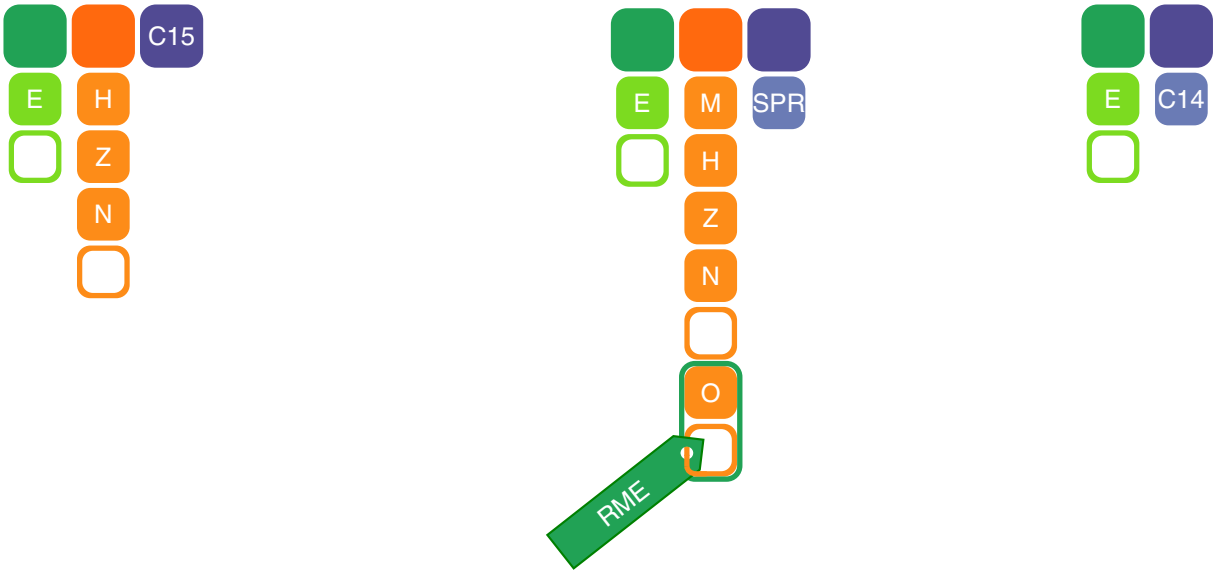
ZH, NH	Illegal; Secure hypervisor is not supported by CPU
ZI, NI	Illegal; Secure intermediate addresses are not supported by CPU

Illegal access class combinations when ArmV8.4-A secure hypervisor is implemented:

ZHR, NHR **ZHT, NHT** **ZHTE, NHTE**	The ArmV8.4-A extension does not include a secure AArch32 hypervisor. Therefore any 32-bit access class specifiers (R, T, TE) are illegal in combination with “NH” or “ZH”.
ZIR, NIR **ZIT, NIT** **ZITE, NITE**	The ArmV8.4-A extension does not include a secure AArch32 intermediate addresses. Therefore any 32-bit access class specifiers (R, T, TE) are illegal in combination with “NH” or “ZH”.

Peripheral Register Access

This is used to access core ID and configuration/control registers.



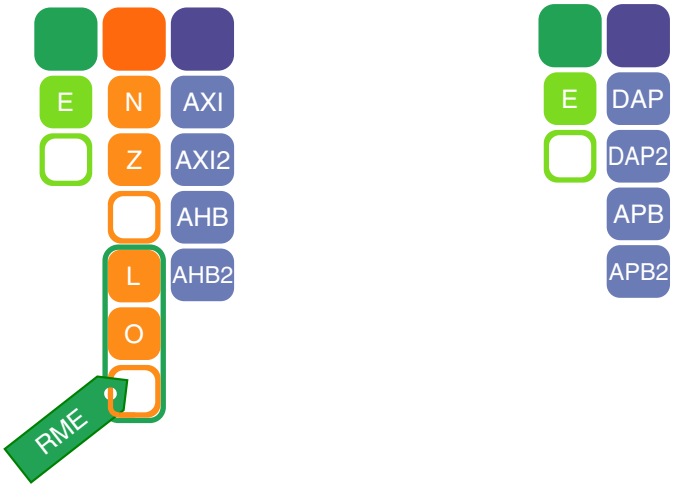
Example combinations:

NC15	Access non-secure banked coprocessor 15 register (AArch32 mode)
C15	Access coprocessor 15 register in current secure mode (AArch32 mode)
SPR	Access system register (AArch64 mode)
MSPR	Access system registers in EL3 (AArch64) mode

HSPR Access system registers in EL2 (AArch64) mode

ZSPR Access system registers in secure EL1 (AArch64) mode

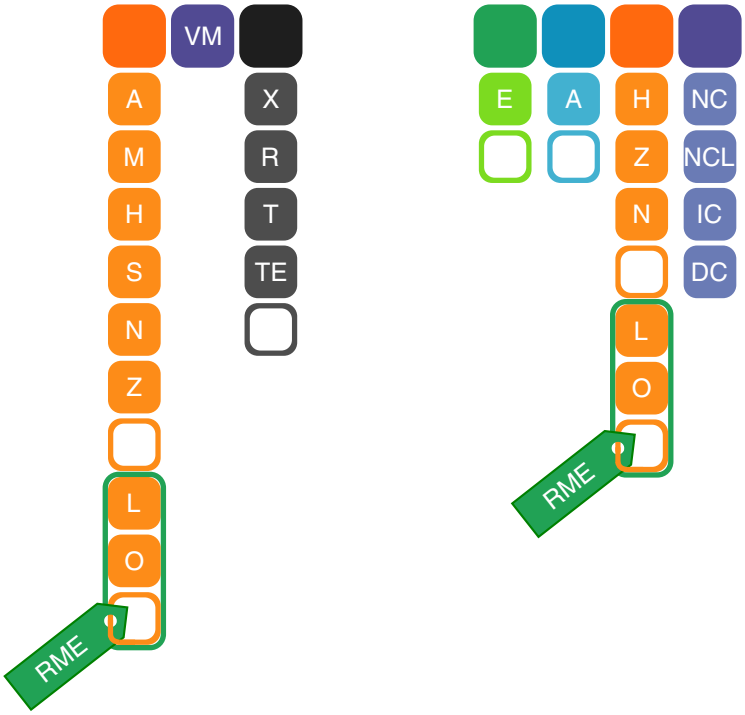
These accesses are typically used to access the CoreSight busses APB, AHB and AXI directly through the DAP bypassing the CPU. For example, this could be used to view physical memory at run-time.



Example combinations:

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

Used to access the TRACE32 virtual memory (VM:) or the data and instruction caches or to bypass them.



Example combinations:

VM	Access virtual memory using current CPU context
AVM	Access virtual memory ignoring current CPU context
HVMR	Access virtual memory that is banked in hypervisor mode and disassemble memory content as 32-bit Arm instruction code
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:

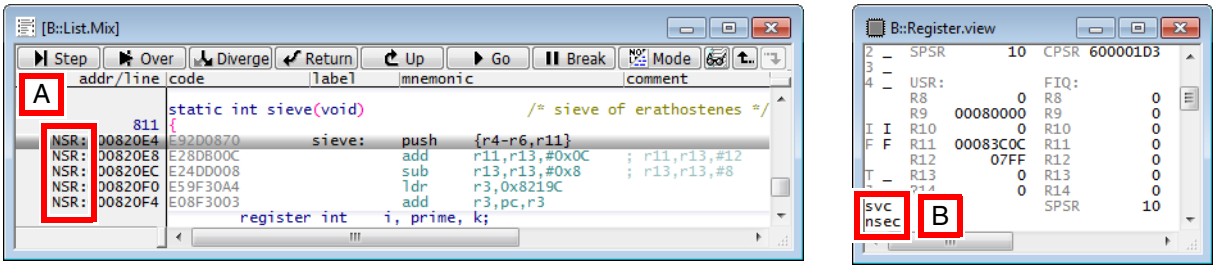
- The current CPU context (architecture specific)
- The used window type (e.g. **Data.dump** window for data or **List.Mix** window for code)
- Symbol information of the loaded application (e.g. combination of code and data)
- Segments that use different instruction sets
- Debugger specific settings (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)	**NSR**:	**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 32-bit 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 "NSR".

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

The following coprocessors can be accessed if available in the processor:

Coprocessor 14. Please refer to the chapter **Virtual Terminal** and to your Arm documentation for details. On Cortex-A and Cortex-R the debug register can be accessed by 'C14' access class and the address is the address offset in the debug register block divided by 4. Recommended is to use the 'DAP:' or 'EDAP:' access class, but then the address is the address offset plus the base address of the debug register block which is 0xd4011000.

Coprocessor 15, which allows the control of basic CPU functions. This coprocessor can be accessed with the access class C15. For the detailed definition of the CP15 registers, please refer to the Arm data sheet. The CP15 registers can also be controlled in the **PER** window.

The TRACE32 address is composed of the CRn, CRm, op1, op2 fields of the corresponding coprocessor register command

```
<MCR|MRC> p15, <op1>, Rd, CRn, CRm, <op2>

BIT0-3:CRn, BIT4-7:CRm, BIT8-10:<op2>, BIT12-14:<op1>, Bit16=0 (32-bit
access)


<MCRR|MRRC> p15, <op1>, <Rd1>, <Rd2>, <CRm>

BIT0-3: -, BIT4-7:CRm, BIT8-10: -, BIT12-14:<op1>, Bit16=1 (64-bit access)
```

is the corresponding TRACE32 address (one nibble for each field).

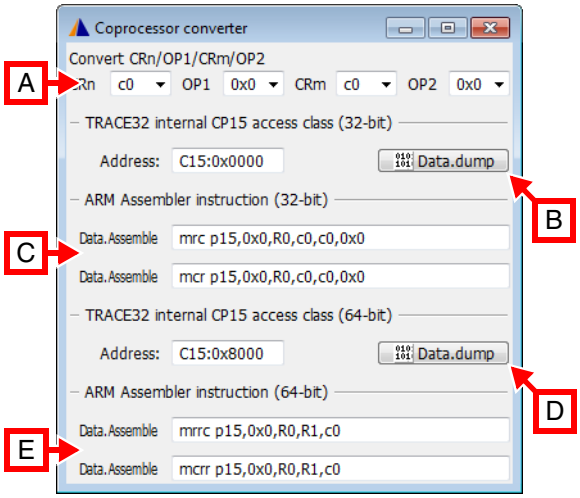
Coprocessor Converter Dialog

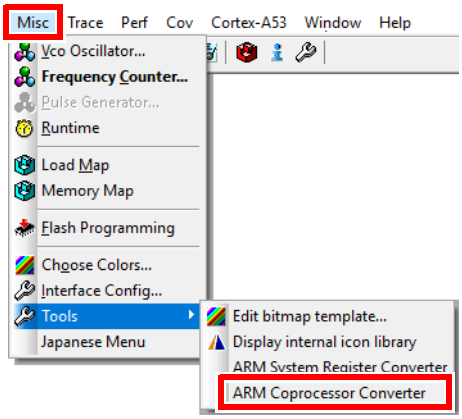
The demo directory offers a **Coprocessor converter** dialog, which assists in calculating the C15 address class offsets.

To display the **Coprocessor converter** dialog, run this command:

```
DO ~~/demo/arm/etc/coprocessor/coprocessor_converter.cmm
```

Alternatively, you can open the converter from the **Misc** menu:





A Edit Coprocessor parameters here.

B Open **Data.dump** window at current 32-bit Coprocessor address.

C Assemble MRC/MCR instruction at current PC location.

D Open **Data.dump** window at current 64-bit Coprocessor address.

E Assemble MRRC/MCRR instruction at current PC location.

On Cortex-A/R or ARM11 you can access other available coprocessors by using the same addressing scheme. The access class is then e.g. “C10:” instead of “C15”. You need to secure that access to this coprocessor is permitted in the Coprocessor Access Control Register.

The “C15:” access class provides the view of the mode the core currently is in. On devices having “TrustZone” (ARM1176, Cortex-A) there are some banked CP15 register, one for secure and one for non-secure mode. With “ZC15:” and “NC15:” you can access the secure / non-secure bank independent of the current core mode. On devices having a “Hypervisor” mode (e.g. Cortex-A7, -A15) there are CP15 register which are only available in hypervisor mode or in monitor mode with NS bit set. With “HC15:” you can access these register independent of the current core mode.

Coprocessor access in per file

Usually per files use the “C1x” class to access coprocessors, and the “AD:” access class to access other peripherals that are directly memory mapped.

All these accesses may be done in non-secure or secure mode, dependant on the SoC implementation. The non-secure/secure access is automatically selected, so per default **PER** shows the content using the current CPU secure mode.

Sometimes coprocessors or peripherals might only show “????????” if the access is not possible in the current CPU secure mode.

In this case you can enforce the secure mode with the /**Secure** or /**NonSecure** option.

```
PER , /Secure                 // always use “ZC1x” or “AZD” in per file
PER , /NonSecure              // always use “NC1x” or “AND” in per file
```

The /**Secure** or /**NonSecure** can be combined with /**DualPort** option (run-time access), example:

```
PER , /DualPort /Secure
```

NOTE:	Non-intrusive run-time accesses are not possible for coprocessors. Peripherals that are directly memory mapped need to be mapped to either the AXI or AHB in a 1:1 fashion to make a non-intrusive run-time access possible. For more information about intrusive and non-intrusive run-time accesses, please see **Accessing Memory at Run-time**.

Accessing Memory at Run-time

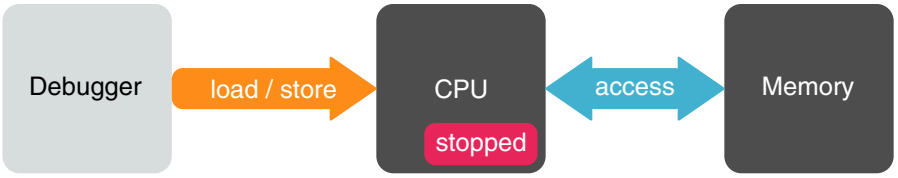
This sections describes how memory can be accessed at run-time. It gives an overview of all available methods for Arm based devices.

In this section:

Intrusive and Non-intrusive Run-time Access

Intrusive run-time access

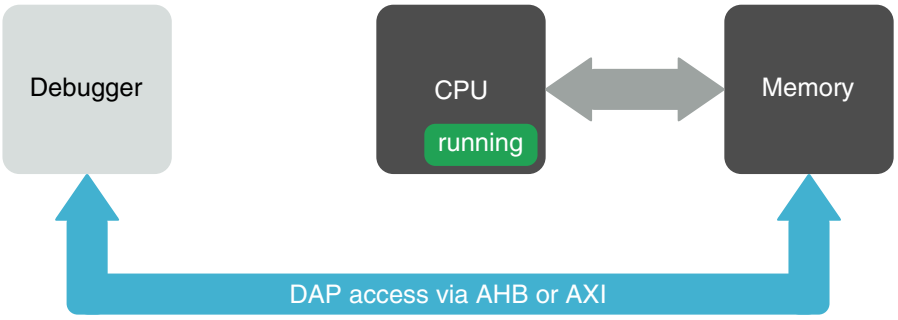
Intrusive means that the CPU is periodically stopped and restarted, so that the debugger can access the memory content through the CPU using load / store commands.



The debugger will see memory the same way the CPU does; however, real-time constraints may be broken.

Non-intrusive run-time access

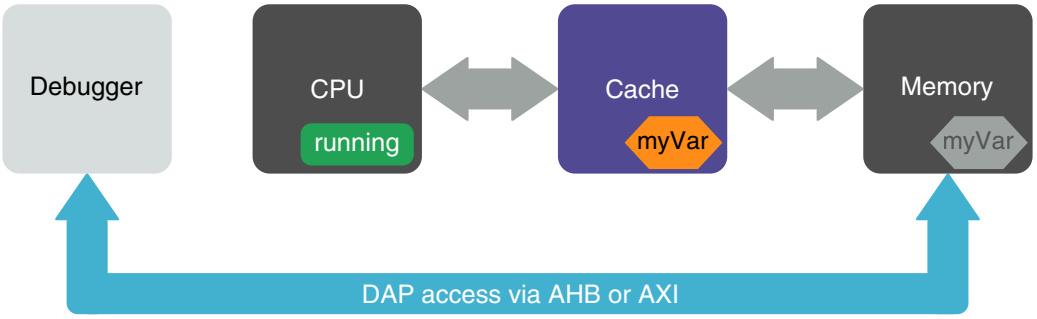
Non-intrusive means that the CPU is not stopped during the memory access.



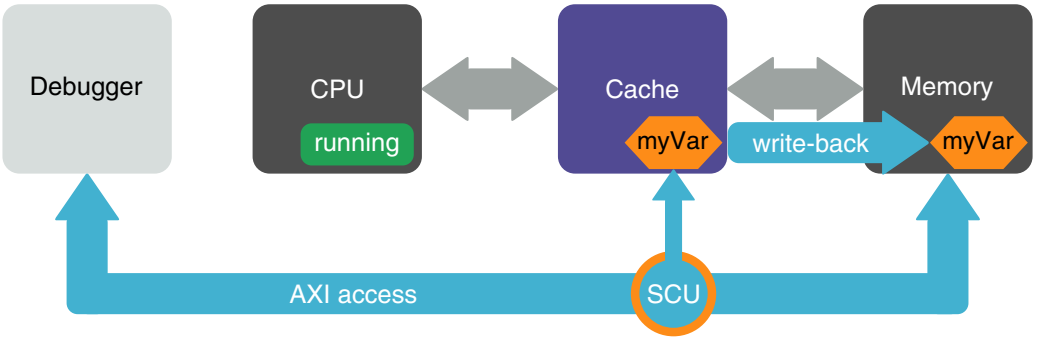
The debugger cannot read through the CPU while it is running and continuously accessing memory. Therefore the debugger has to use a DAP access, i.e. the AHB or AXI bus. The CPU is bypassed, which will equal a physical memory access. This way the real-time constrains are preserved. This access method only works if an AHB or AXI is present and if the busses are properly mapped to memory.

Cache Coherent Non-intrusive Run-time Access

A non-intrusive run-time access through the AHB/AXI bus will bypass caches. In the example below, “myVar” is only updated in the cache but not in memory. Hence its current state is invisible to the debugger.



An example of such a cache would be a write-back cache. For the debugger to see the current value of “myVar”, a run-time access has to trigger a cache flush, so that “myVar” is written back to memory.



In this example, the cache coherency is maintained by the Snoop Control Unit (SCU). During an AXI access, the SCU can be instructed to trigger a write of “myVar” back to memory. This feature is not supported for the AHB. It is implementation-defined whether this is available for AXI transactions.

Performing Intrusive and Non-intrusive Run-time Accesses with TRACE32

All of the previously mentioned access methods can be carried out in TRACE32.

To access memory at run-time, add the access class “E” as a prefix. “E” means run-time access and can be combined with most access classes that access memory. E.g. “Data.dump NSD:*<address>*” can be extended to “Data.dump ENSD:*<address>*”.

Intrusive run-time access

To activate intrusive memory accesses, use the command **SYStem.MemAccess.StopAndGo**.

```
SYStem.MemAccess.StopAndGo      ; Intrusive run-time memory access, CPU
                                ; is periodically stopped / restarted
Data.dump E:0x100               ; Intrusive access via CPU. Prefix "E"
Var.view %E myVar               ; is required to read 0x100 or myVar
```

Non-intrusive run-time access: Direct DAP access

You can directly specify an access to memory via the AHB or AXI bus using an access class. This requires that the AHB or AXI is defined as a valid access port. If you select a known chip with **SYStem.CPU**, then TRACE32 configures this setting automatically. Please see the following example for the AXI:

```
SYStem.CONFIG MEMORYACCESSPORT 1.   ; Define memory access port and AXI
SYStem.CONFIG AXIACCESSPORT 1.      ; access port (e.g. port number 1)

Data.dump EAXI:<address>            ; Run-time access via AXI. Prefix "E"
Data.dump EAXI:myVar                ; is optional but recommended to read
                                    ; myVarn via the DAP
```

Non-intrusive run-time access: Indirect DAP access

It is not very convenient or even not always possible to use an AXI or AHB access class specifier. In most cases you should let the debugger decide which access to use. Use the command **SYStem.MemAccess** DAP to activate non-intrusive run-time accesses via AHB or AXI. TRACE32 will then redirect access to the AHB or AXI bus. This requires that the AHB or AXI is defined as a valid access port.

```
SYStem.CONFIG MEMORYACCESSPORT 1.   ; Define memory access port and AHB
// SYStem.CONFIG AHBACCESSPORT 1.   ; or AXI access port
SYStem.CONFIG AXIACCESSPORT 1.

SYStem.MemAccess DAP                ; Non-intrusive access via AHB / AXI

Data.dump E:0x100                   ; Run-time access via DAP. Prefix "E"
Var.view %E myVar                   ; is required to read 0x100 or myVar
```

Performing Cache Coherent Run-time Accesses with TRACE32

So far there is not guarantee that the run-time accesses via AHB / AXI will be coherent. This means, you might not see the current value of e.g. a variable because the value is in the cache but not updated in memory.

The AXI may allow you to select whether an access should be performed as a coherent transaction or not. To activate this feature, use **SYStem.Option.AXIACEEnable ON**

```
SYStem.CONFIG.MEMORYACCESSPORT 1.   ; Define memory access port and AXI
SYStem.CONFIG.AXIACCESSPORT 1.      ; access port (e.g. port number 1)

SYStem.Option.AXIACEEnable ON       ; Enable cache coherent transactions
SYStem.MemAccess DAP                ; Non-intrusive access via AXI

Data.dump E:0x100                   ; Run-time access via AXI. Prefix "E"
Var.view %E myVar                   ; is required to read 0x100 or myVar
```

NOTE:	• Support for cache coherent AXI transactions is implementation-defined. Therefore **SYStem.Option.AXIACEEnable ON** may be without effect. • The AHB does not provide such a coherency mechanism.

Coherent cache accesses without AXI coherency support

The AXI may not provide cache coherent transactions or there may only be an AHB available. In this case you can still perform non-intrusive cache-coherent run-time memory accesses. But this requires that you change the configuration of your target application in one of the following ways:

- Configure the address range of interest as “non-cacheable”
- Configure the address range of interest as “write-through”
- Configure the entire cache as “write-through” (global setting)
- Make the CPU periodically flush the cache lines of interest
- Disable the cache
- Use a monitor program that accesses the memory address range of interest through the cache (CPU view) and provides the result to the debugger, e.g. via shared memory or DCC. This requires a code instrumentation of the target application.

Additional Considerations

Non-intrusive run-time access with active MMU

If the run-time access involves virtual addresses that do not directly map to physical addresses, the debugger has to be made aware of the proper virtual-to-physical address translations. For more information about address translations, refer to the descriptions of the following commands:

TRANSlation.Create	If the CPU has never stopped, set the translation manually.
MMU.SCAN	Scan static page tables into the debugger while the CPU is stopped.
TRANSlation.TableWalk	Use if CPU stops and page tables are modified frequently (e.g. by OS).

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.

A semihosting call from the application causes an exception by a SVC (SWI) instruction together with a certain SVC number to indicate a semihosting request. The type of operation is passed in R0. R1 points to the other parameters. On Cortex-M semihosting is implemented using the BKPT instead of SVC instruction.

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”.

The debugger which needs to interface to the I/O facilities on the host provides two ways to handle a semihosting request which results in a SVC (SWI) or BKPT exception:

SVC (SWI) Emulation Mode

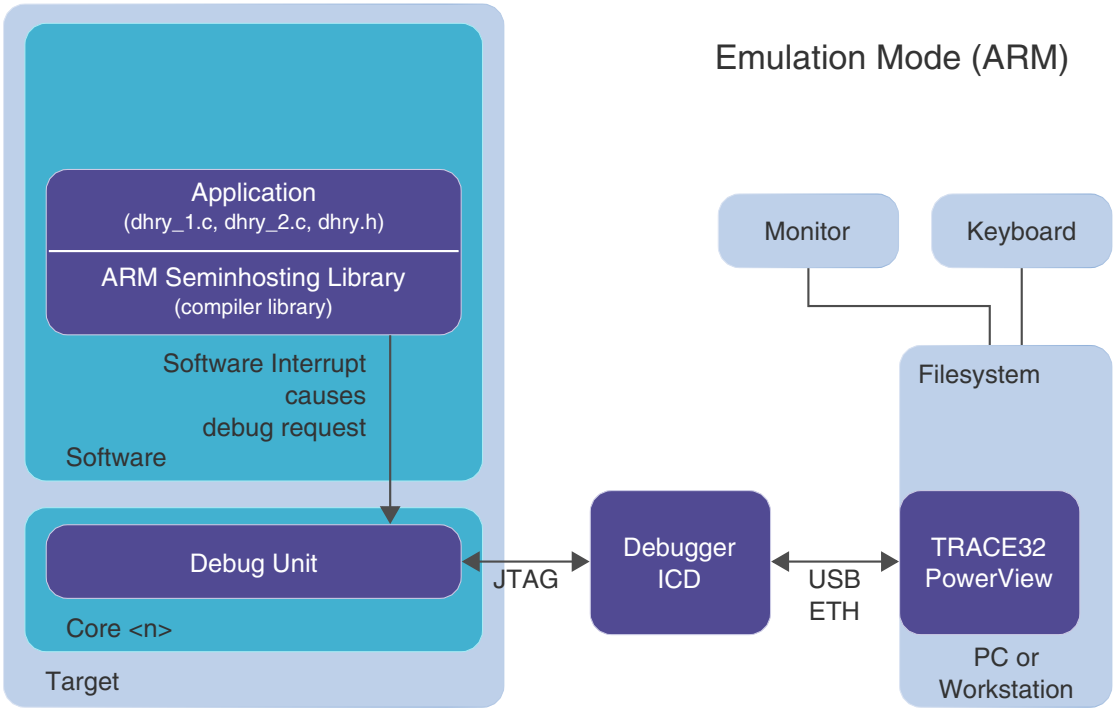
A breakpoint placed on the SVC exception entry stops the application. The debugger handles the request while the application is stopped, provides the required communication with the host, and restarts the application at the address which was stored in the link register R14 on the SVC exception call. Other as for the DCC mode the SVC parameter has to be 0x123456 to indicate a semihosting request.

This mode is enabled by **TERM.METHOD ARMSWI** [*<address>*] 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.

On Arm7 an on-chip or software breakpoint needs to be set at address 8 (SWI exception entry). On other Arm cores also the vector catch register can be used: **TrOnchip.Set SWI ON**. The Cortex-M does not need a breakpoint because it already uses the breakpoint instruction BKPT for the semihosting request.

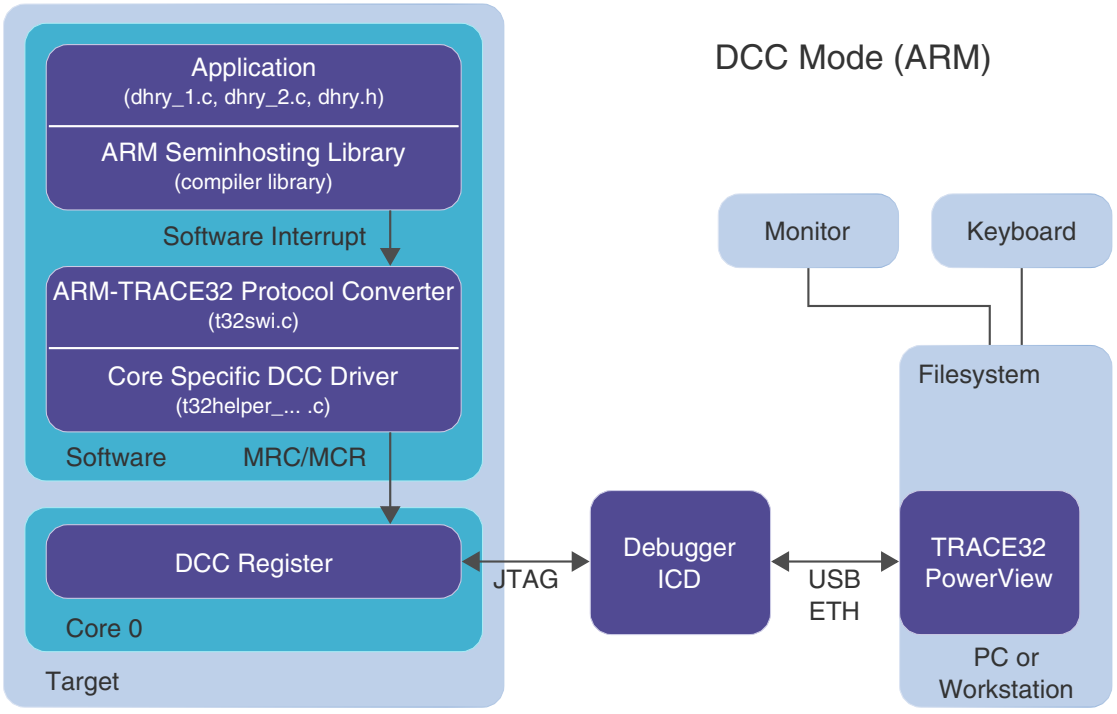
When using the *<address>* option of the **TERM.METHOD ARMSWI** *<address>*, any memory location with a breakpoint on it can be used as a semihosting service entry instead of the SVC call at address 8. The application just needs to jump to that location. After servicing the request the program execution continues at that address (not at the address in the link register R14). You could for example place a ’BX R14’ command at that address and hand the return address in R14. Since this method does not use the SVC command no parameter (0x123456) will be checked to identify a semihosting call.

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 example can be found in ~~/demo/arm/etc/semihosting_arm_emulation/swisoft_<*x*>.cmm.

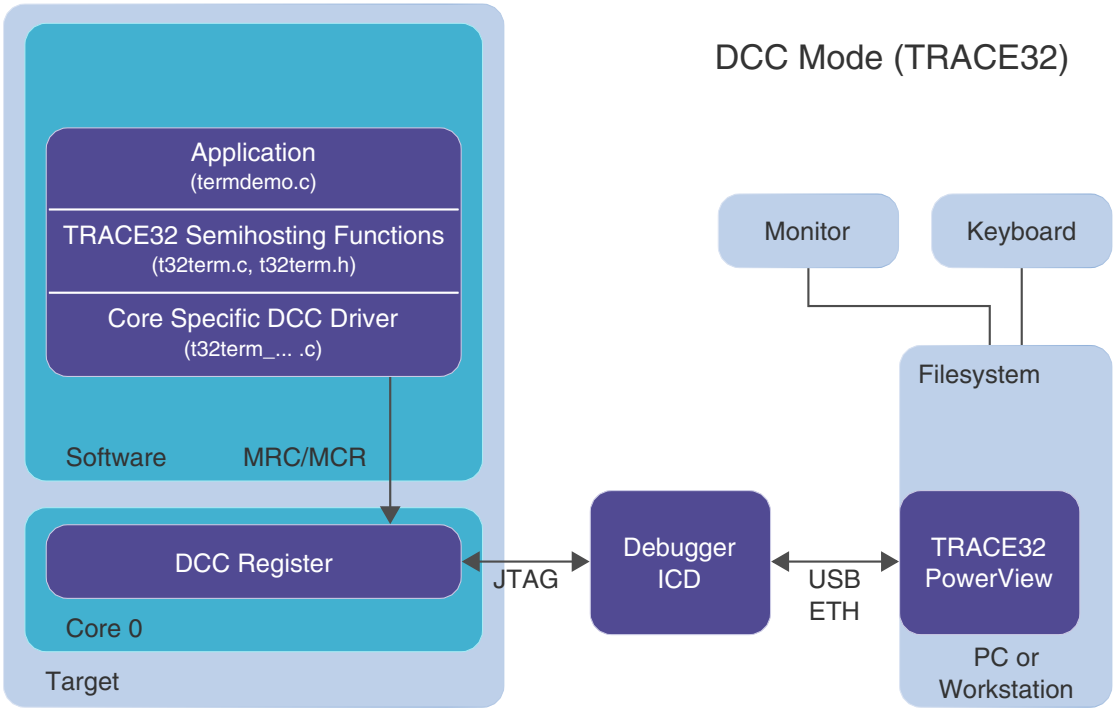


A semihosting exception handler will be called by the SVC (SWI) exception. It uses the Debug Communication Channel based on the JTAG interface to communicate with the host. The target application will not be stopped, but the semihosting exception handler needs to be loaded or linked to the application. The Cortex-M does not provide a DCC, therefore this mode can not be used.

This mode is enabled by **TERM.METHOD DCC3** 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 Arm C library reads these memory parameters by a SYS_HEAPINFO semihosting call and uses them for initialization. An example (swidcc_x.cmm) and the source of the Arm compatible semihosting handler (t32swi.c, t32helper_x.c) can be found in ~~/demo/arm/etc/semihosting_arm_dcc



In case the Arm library for semihosting is not used, you can alternatively use the native TRACE32 format for the semihosting requests. Then the SWI handler (t32swi.c) is not required. You can send the requests directly via DCC. Find examples and source codes in ~~/demo/arm/etc/semihosting_trace32_dcc



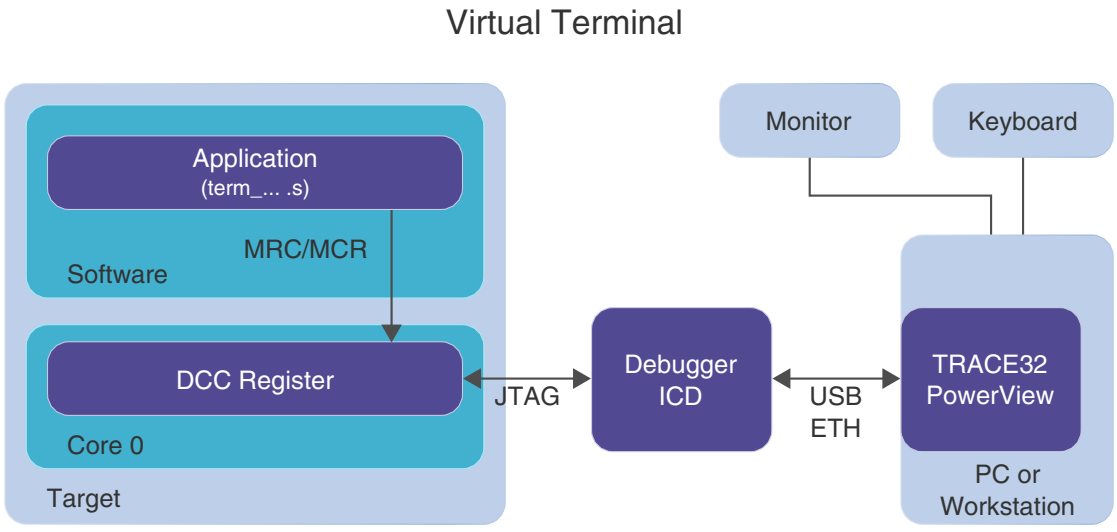
The command **TERM** opens a terminal window which allows to communicate with the Arm core over the Debug Communications Channel (DCC). All data received from the comms channel are displayed and all data inputs to this window are sent to the comms channel. Communication occurs byte wide or up to four bytes per transfer. The following modes can be used:

DCC	Use the DCC port of the JTAG interface to transfer 1 byte at once.
DCC3	Three byte mode. Allows binary transfers of up to 3 bytes per DCC transfer. The upper byte defines how many bytes are transferred (0 = one byte, 1 = two bytes, 2 = three bytes). This is the preferred mode of operation, as it combines arbitrary length messages with high bandwidth.
DCC4A	Four byte ASCII mode. Does not allow to transfer the byte 00. Each non-zero byte of the 32-bit word is a character in this mode.
DCC4B	Four byte binary mode. Used to transfer non-ASCII 32-bit data (e.g. to or from a file).

The **TERM.METHOD** command selects which mode is used (**DCC**, **DCC3**, **DCC4A** or **DCC4B**).

The communication mechanism is described e.g. in the ARM7TDMI data sheet in chapter 9.11. Only three move to/from coprocessor 14 instructions are necessary.

The TRACE32 ~~/demo/arm/etc/virtual_terminal directory contains examples for the different Arm families which demonstrate how the communication works.



Large Physical Address Extension (LPAE)

LPAE is an optional extension for the Armv7-AR architecture. It allows physical addresses above 32-bit. The instructions still use 32-bit addresses, but the extended memory management unit can map the address within a 40-bit physical memory range.

It is for example implemented on Cortex-A7 and Cortex-A15.

Consequence for Debugging

We have extended only the physical address, because the virtual address is still 32-bit.

Example: Memory dump starting at physical address 0x0280004000.
"A:" = absolute address = physical address.

```
Data.dump A:02:80004000
```

Unfortunately the above command will result in a bus error ('????????') on a real chip because the debug interface does not support physical accesses beyond the 4GByte. It will work on the TRACE32 Instruction Set Simulator and on virtual platforms.

In case the Debug Access Port (DAP) of the chip provides an AXI MEM-AP then the debugger can act as a bus master on the AXI, and you can access the physical memory independent of TLB entries.

```
Data.dump AXI:02:80004000
```

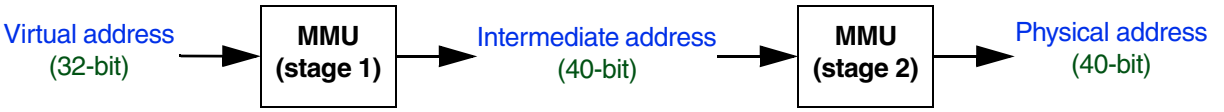
However this does not show you the cache contents in case of a write-back cache. For a cache coherent access you need to set:

```
SYStem.Option.AXIACEEnable ON
```

This requires that the CPU debug logic supports this setting. If the debug logic does not support coherent AXI accesses, this option is will be without effect.

Virtualization Extension, Hypervisor

The 'Virtualization Extension' is an optional extension in Armv7-A. It can for example be found on Cortex-A7 and Cortex-A15. It adds a 'Hypervisor' processor mode used to switch between different guest operating systems. The extension assumes **LPAE** and **TrustZone**. It adds a second stage address translation.



Consequence for Debugging

The debugger shows you the memory view of the mode the core is currently in. The address translation and therefore the view can/will be different for secure mode, non-secure mode, and hypervisor mode.

You can force a certain view/translation by switching to another mode or by using the access classes "Z:" (secure), "N:" (non-secure) or "H:" (hypervisor).

If you want to perform an access addressed by an intermediate address, you can use the 'I:' access class.

OS Awareness for multiple operating systems is under development. At the moment you can have only one OS Awareness at a time.

Run-time Measurements

The **RunTime** command group allows run-time measurements based on polling the CPU run status by software. Therefore the result will be about a 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. Please do not disable the option **SYStem.Option.DBGACK**. The run-time of the debugger accesses while the CPU is halted would also be measured, otherwise.

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 group.

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

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** | **FULL** **STOP** | **FULLCTI**]
ETS.view

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

HSM.Base *<address>*
HSM.RESet

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

ICE.Base *<address>*
ICE.RESet

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

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.)

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 parameterss 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.

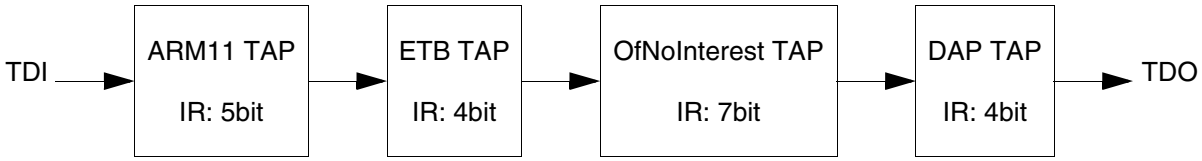
ETB (Embedded Trace Buffer) TAP if the ETB has its own TAP to access its control register (typical with Arm11 cores).
-> ETBDRPOST, ETBDRPRE, ETBIRPOST, ETBIRPRE.

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.

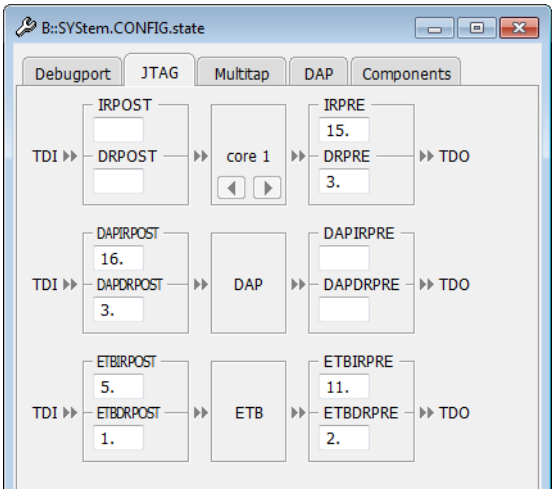
RTP (RAM Trace Port) TAP if the RTP has its own TAP to access its control register.
-> RTPDRPOST, RTPDRPRE, RTPIRPOST, RTPIRPRE.

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:



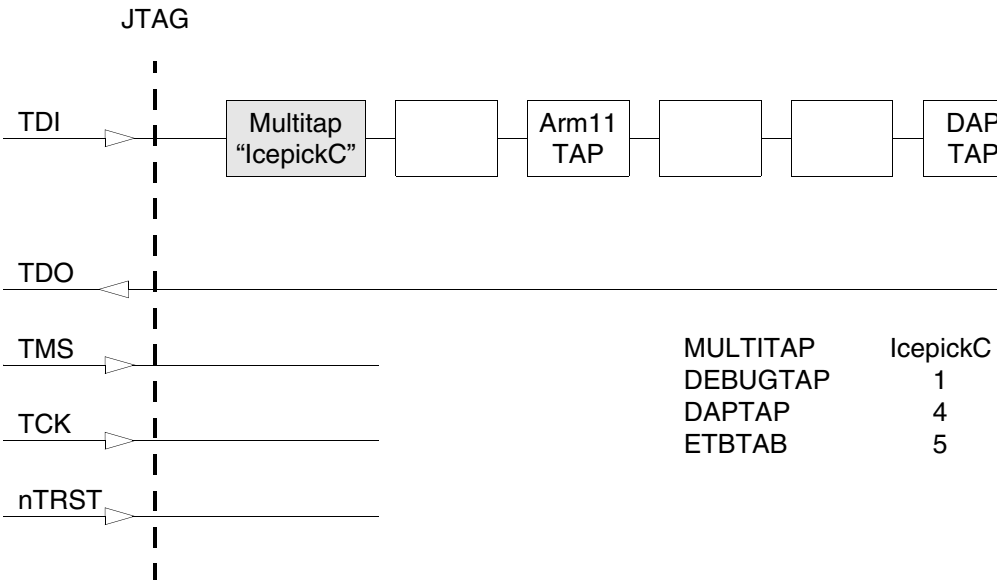

```
SYStem.CONFIG IRPRE 15.
SYStem.CONFIG DRPRE  3.
SYStem.CONFIG DAPIRPOST 16.
SYStem.CONFIG DAPDRPOST  3.
SYStem.CONFIG ETBIRPOST  5.
SYStem.CONFIG ETBDRPOST  1.
SYStem.CONFIG ETBIRPRE  11.
SYStem.CONFIG ETBDRPRE   2.
```



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).

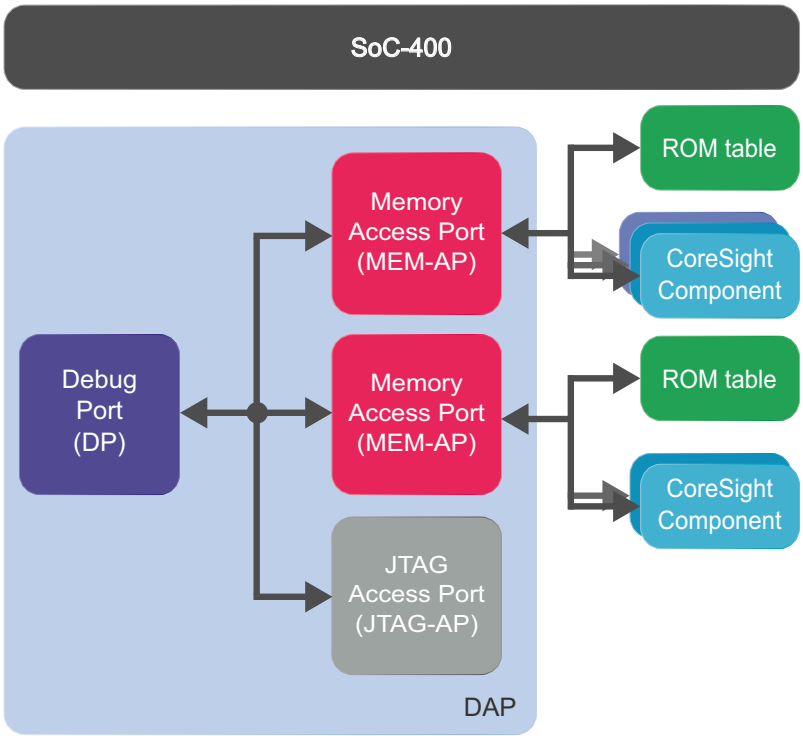
An Access Port (AP) is a CoreSight module from Arm which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

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).

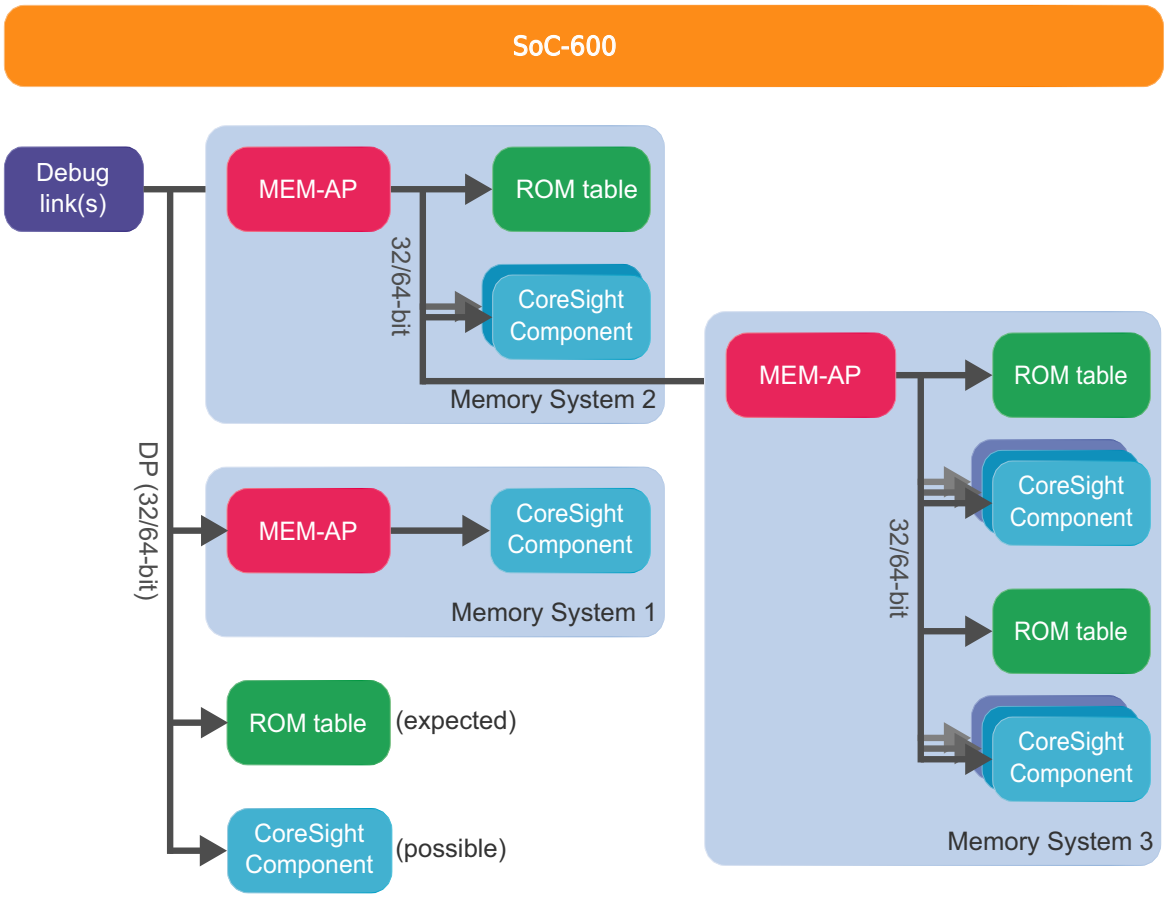
2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number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

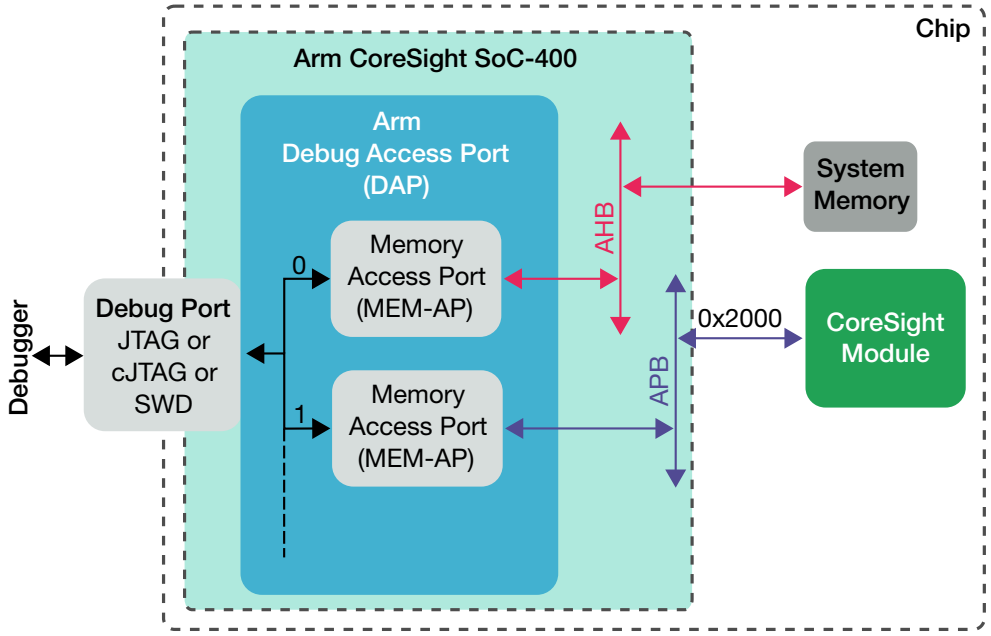
Example 1: SoC-400



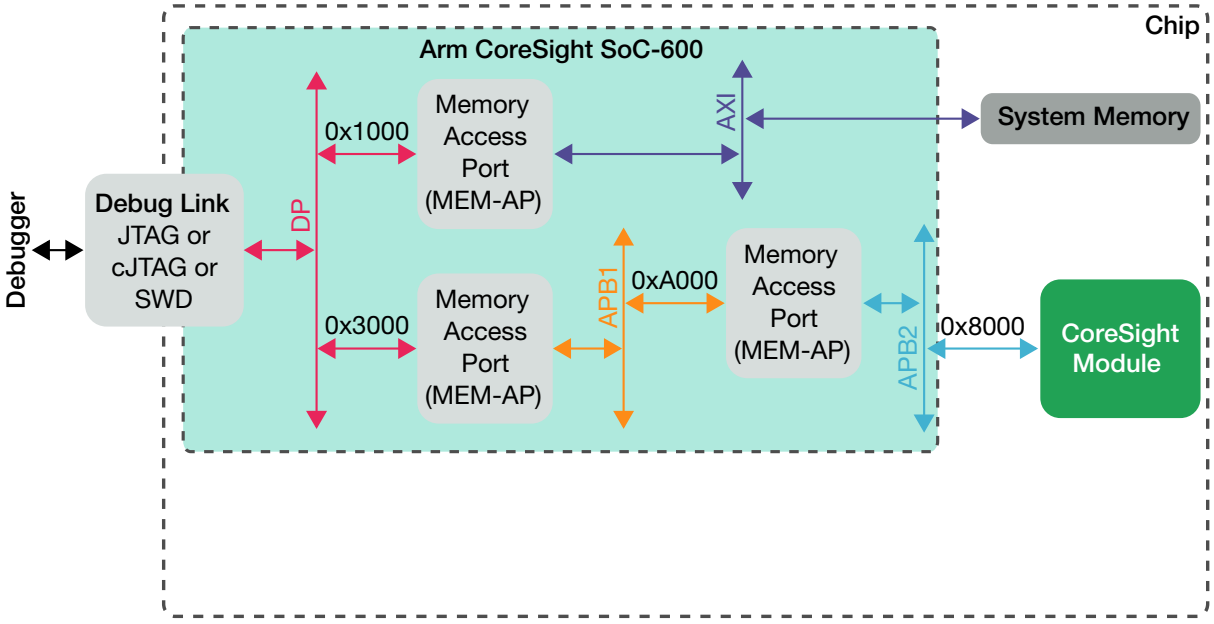
Example 2: SoC-600



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.

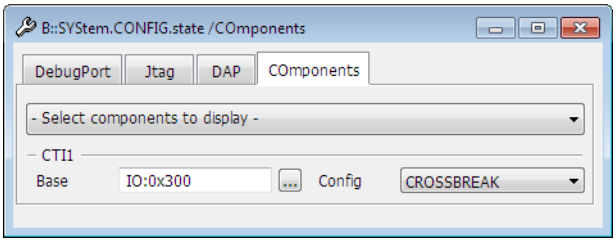
Normally, these values can be set automatically using **SYStem.DETECT.XCPTRI**. For Details, see **"Target Resources"** in XCP Debug Back-End, page 7 (backend_xcp.pdf)

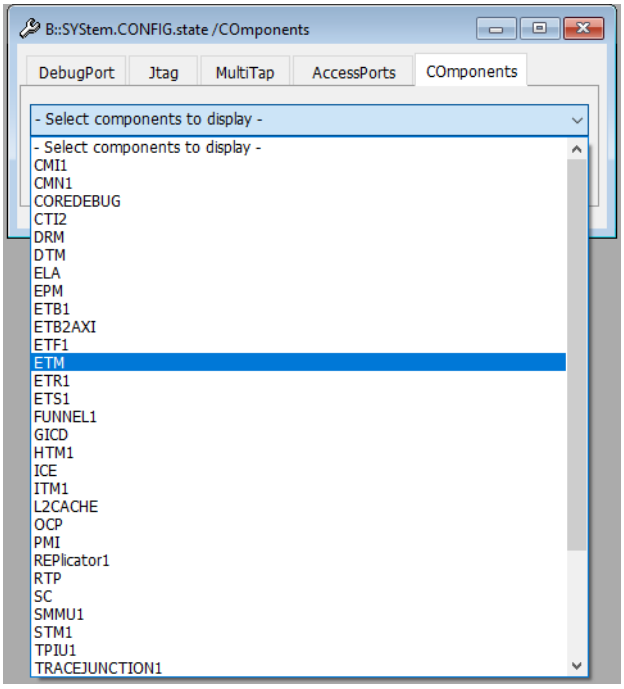
AHBAPn.XCPTRI *<tri>*
Configures the debugger to use the target resource identifier <tri> (0 to 255) for AHB accesses. Default: not set.

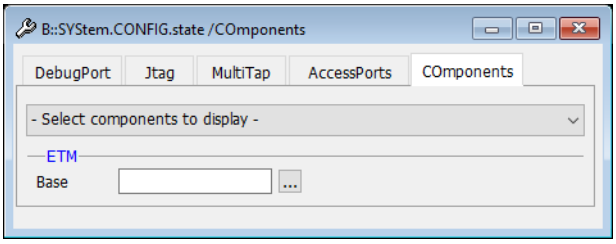
APBAPn.XCPTRI *<tri>*
Configures the debugger to use the target resource identifier <tri> (0 to 255) for APB accesses. Default: not set.

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

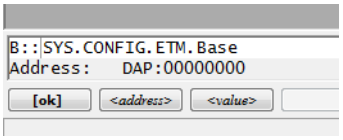
On the **Components** tab in the **SYStem.CONFIG.state** window, you can comfortably add the debug and trace components your chip includes and which you intend to use with the debugger's help.







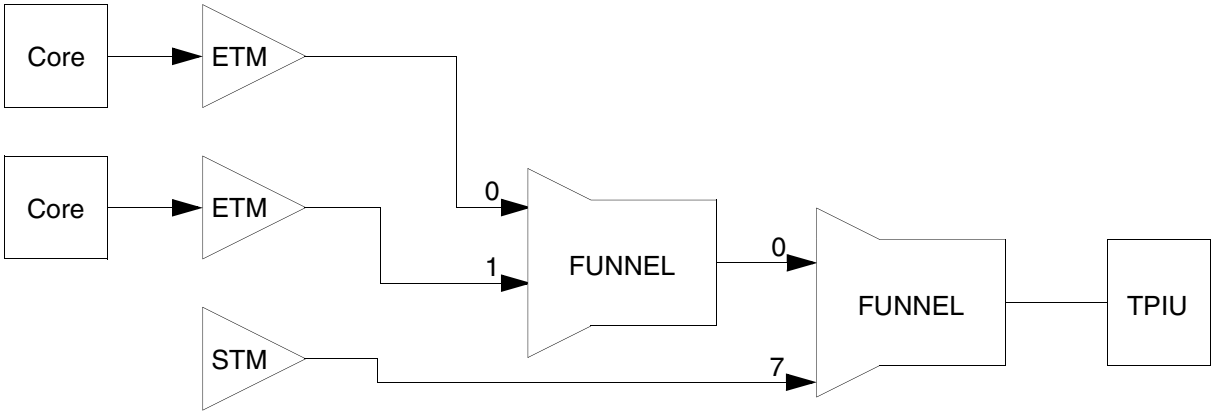
Each configuration can be done by a command in a script file as well. Then you do not need to enter everything again on the next debug session. If you press the button with the three dots you get the corresponding command in the command line where you can view and maybe copy it into a script file.



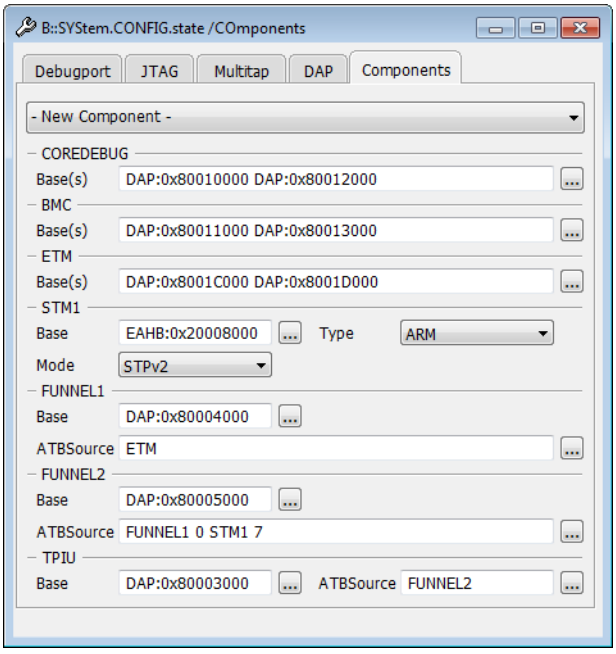
You can have several of the following components: CMI, ETB, ETF, ETR, FUNNEL, STM.
Example: FUNNEL1, FUNNEL2, FUNNEL3,...

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.

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.

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

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).

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

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.

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.

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).

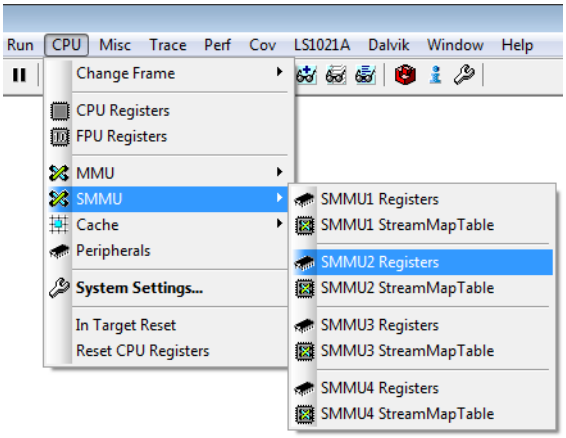
LowwhenStopped The WDTDIS pin is driven low when program is stopped (not XCP).

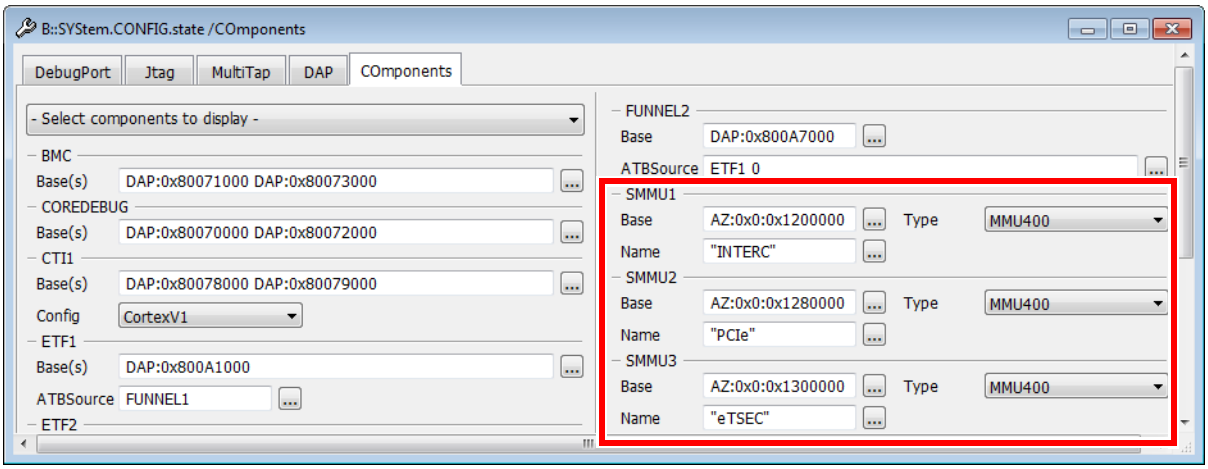
Format:	**SYStem.CONFIG.SMMU***<x> <sub_cmd>*
<x>:	**1** … **20**
<sub_cmd>:	**Base** *<base_address>* **Type MMU400** | **MMU401** | **MMU500** **Name** "*<name>*" **RESet**

For some CPUs with SMMUs, TRACE32 configures the SMMUs parameters *automatically* after you have selected a CPU with the **SYStem.CPU** command.

NOTE: For a *manual* SMMU configuration, use the **SMMU.ADD** command.

You can access the automatically configured SMMUs through the **CPU** menu **> SMMU** submenu in TRACE32. The individual SMMU configurations can be viewed in the **SYStem.CONFIG.state /COmponent** window.





<x>	Serial number of the SMMU.
Base	Logical or physical base address of the memory-mapped SMMU register space.
Type	Defines the type of the Arm system MMU IP block: **MMU400**, **MMU401**, or **MMU500**.
Name	Assigns a user-defined name to an SMMU.
RESet	Resets the configuration of an SMMU specified with *<x>*.

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

<cpu>: **ARM7TDMI** | **ARM740TD** | … (JTAG Debugger Arm7)
ARM9TDMI | **ARM920T** | **ARM940T** |… (JTAG Debugger Arm9)
JANUS2 (JTAG Debugger Janus)
ARM1020E | **ARM1022E** | **ARM1026EJ** |…(JTAG Debugger Arm10)
ARM1136J | **ARM1136JF** |… (JTAG Debugger Arm11)
CORTEXA8 | **SCORPION** |…(JTAG Debugger Cortex-A)
CORTEXM3 |…(JTAG Debugger Cortex-M)

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

Default selection:

- ARM7TDMI if the JTAG Debugger for Arm7 is used.
- ARM9TDMI if the JTAG Debugger for Arm9 is used.
- JANUS2 if the JTAG Debugger for JANUS is used.
- ARM1020E if the JTAG Debugger for Arm10 is used.
- ARM1136J if the JTAG Debugger for Arm11 is used.
- CORTEXA8 if the JTAG Debugger for Cortex-A is used.
- CORTEXM3 if the JTAG Debugger for Cortex-M is used.

SYStem.JtagClock Define the frequency of the debug port

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

SYStem.BdmClock (deprecated)

<frequency>: **4 kHz**…**100 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 affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the 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 (**R**eturned **TCK**). On some processor derivatives (e.g. ArmxxxE-S) there is the need to synchronize the processor clock and the JTAG 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 fix debug clock below 1/6 of the processor clock (Arm7, Arm9), below 1/8 of the processor clock (Arm11), respectively. 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. The **RTCK** mode allows theoretical frequencies up to 1/6 (Arm7, Arm9) or 1/8 (Arm11) of the processor clock. For 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.

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

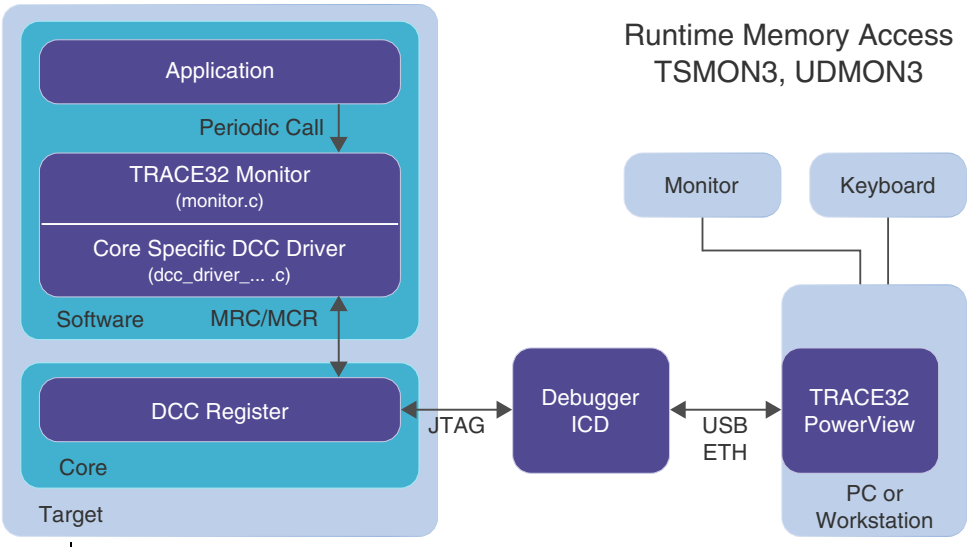
<mode>: **AHB** | **AXI** | **APB** | … (SoC-600)
DAP (SoC-400)

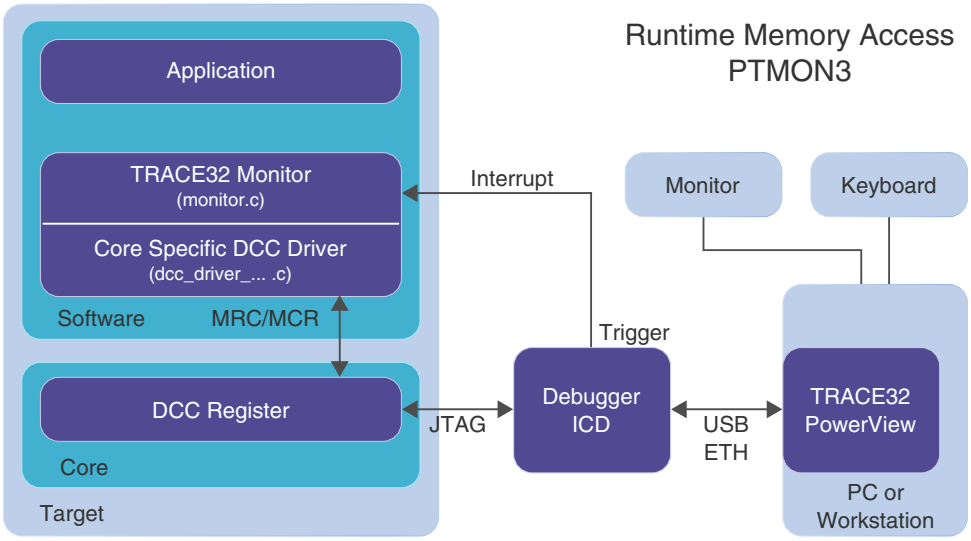
Cerberus
Enable
NEXUS
TSMON3
TSMON
PTMON3
PTMON
QMON (deprecated)
UDMON3
UDMON
RealMON
TrkMON
GdbMON
Denied
StopAndGo

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 CPU 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.
Cerberus	The memory access is done through an Infineon proprietary Cerberus module. This memory access is only available and selectable on a few Infineon processors and only by script or in the command line.
DAP	For SoC-600, **DAP** must not be used anymore. Use AXI or AHB instead, depending on what the chip offers. A run-time memory access is done via the Arm SoC-400 Debug Access Port (DAP). This is only possible if a DAP is available on the chip and if the memory bus is connected to it (Cortex, CoreSight). **NOTE**: The debugger accesses the memory bus and cannot see caches. Run-time memory access via the DAP is not possible on the TRACE32 Instruction Set Simulator.

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.
NEXUS	The memory access is done through the Nexus interface which is only available on MAC7xxx processors.
TSMON3 **TSMON**	TSMON uses a data format which shall not be used anymore. It still works for compatibility reasons. TSMON3 shall be used. A run-time memory access is done via a **Time Sharing Monitor.** The application is responsible for calling the monitor code periodically. The call is typically included in a periodic interrupt or in the idle task of the kernel. See the example in the directory ~~/demo/arm/etc/runtime_memory_access.  Besides run-time memory access TSMON3 would allow run mode debugging. But manual break is not possible with TSMON3 and could only be emulated by polling the DCC port. Therefore better use UDMON3 (or RealMON, TrkMON, GdbMON) for this purpose.

PTMON3 **PTMON**	PTMON uses a data format which shall not be used anymore. It still works for compatibility reasons. PTMON3 shall be used. A run-time memory access is done via a **Pulse Triggered Monitor**. Whenever the debugger wants to perform a memory access while the program is running, the debugger generates a trigger for the trigger bus. If the trigger bus is configured appropriate (**TrBus**), this trigger is output via the TRIGGER connector of the TRACE32 development tool. The TRIGGER output can be connected to an external interrupt in order to call a monitor. See the example in the directory `~~/demo/arm/etc/runtime_memory_access`.  Besides run-time memory access PTMON3 would allow run mode debugging. But manual break is not possible with PTMON3 and could only be emulated by polling the DCC port. Therefore better use UDMON3 (or RealMON, TrkMON, GdbMON) for this purpose.
QMON (deprecated)	Select QNX monitor (pdebug) for Run Mode Debugging of embedded QNX. Ethernet is used as communication interface.

UDMON3 **UDMON**	UDMON uses a data format which shall not be used anymore. It still works for compatibility reasons. UDMON3 shall be used. A run-time memory access is done via a **Usermode Debug Monitor**. The application is responsible for calling the monitor code periodically. The call is typically included in a periodic interrupt or in the idle task of the kernel. For run-time memory access UDMON3 behaves exactly as TSMON3. See the example in the directory `~~/demo/arm/etc/runtime_memory_access` and see the picture at TSMON3. Besides run-time memory access UDMON3 allows run mode debugging. Handling of interrupts when the application is stopped is possible when the background monitor is activated. On-chip breakpoints and manual program break are only possible when the application runs in user (USR) mode. See also the example in the directory `~~/demo/arm/etc/background_monitor`.
RealMON	Run-time memory access and run mode debugging is done via the RealMonitor from Arm. The RealMonitor target software is supplied with Arm Firmware Suite.
TrkMON	Select TRK for Run Mode Debugging of Symbian OS. DCC is used as communication interface.
GdbMON	Select T32server (extended gdbserver) for Run Mode Debugging of embedded Linux. DCC is used as communication interface. For more information refer to **"Run Mode Debugging Manual Linux"** (rtos_linux_run.pdf).
Denied	No memory access is possible while the CPU is executing the program.
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.

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

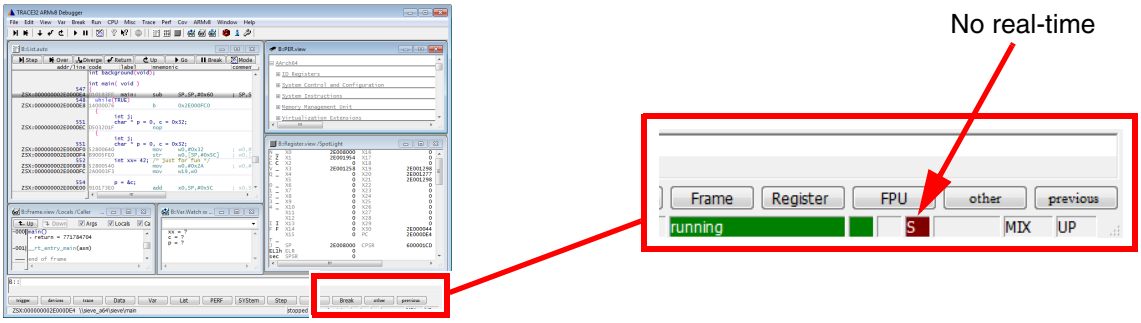
CPU running

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
A	no access	no access	no access	not allowed
AHB or AXI	no access	no access	no access	not allowed
EAHB or EAXI	AHB/AXI	AHB/AXI	AHB/AXI	not allowed

*) 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
```

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** **Prepare** **Go** **Attach** **StandBy** **Up**

Default: Down.

Configures how the debugger connects to the target and how the target is handled.

Down	Disables the debugger. 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.
Prepare	Resets the target. This can be done via the reset line or CPU specific reset registers, see also **SYStem.Option.RESetREGister**. Afterwards direct access to the CoreSight DAP interface is provided. For a reset, the reset line has to be connected to the debug connector. The debugger initializes the debug port (JTAG, SWD, cJTAG) and CoreSight DAP interface, but does not connect to the CPU. This debug mode is used if the CPU shall not be debugged or bypassed, i.e. the debugger can access the memory busses, such as AXI, AHB and APB, directly through the memory access ports of the CoreSight DAP. Typical use cases: • The debugger accesses (physical) memory and bypasses the CPU if a mapping exists. Memory might require initialization before it can be accessed. • The debugger accesses peripherals, e.g. for configuring registers prior to stopping the CPU in debug mode. Peripherals might need to be clocked and powered before they can be accessed. • Third-party software or proprietary debuggers use the TRACE32 API (application programming interface) to access the debug port and DAP via the TRACE32 debugger hardware.

Go	Resets the target via the reset line, initializes the debug port (JTAG, SWD, cJTAG), and starts the program execution. For a reset, the reset line has to be connected to the debug connector. Program execution can, for example, be stopped by the **Break** command.
Attach	No reset happens, the mode of the core (running or halted) does not change. The debug port (JTAG, SWD, cJTAG) will be initialized. After this command has been executed, the user program can, for example, be stopped with the **Break** command.
StandBy	Keeps the target in reset via the reset line and waits until power is detected. For a reset, the reset line has to be connected to the debug connector. Once power has been detected, the debugger restores as many debug registers as possible (e.g. on-chip breakpoints, vector catch events, trace control) and releases the CPU from reset to start the program execution. When a CPU power-down is detected, the debugger switches automatically back to the **StandBy** mode. This allows debugging of a power cycle because debug registers will be restored on power-up. **NOTE**: Usually only on-chip breakpoints and vector catch events can be set while the CPU is running. To set a software breakpoint, the CPU has to be stopped.
Up	Resets the target via the reset line, initializes the debug port (JTAG, SWD, cJTAG), stops the CPU, and enters debug mode. For a reset, the reset line has to be connected to the debug connector. The current state of all registers is read from the CPU.

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.ABORTFIX Do not access memory area from 0x0 to 0x1f

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

Default: OFF.

Workaround for a special customer configuration. It suppresses all debugger accesses to the memory area from 0x0 to 0x1f. This feature is only available on Arm7 family.

SYStem.Option.AMBA Select AMBA bus mode

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

This option is only necessary if a **ARM7 Bus Trace** is used.

Default: OFF.

This option should be set according to the bus mode of the ASIC.

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

This option is required for Cortex-A9, Cortex-A9MPCore r0p0, r0p1, r1p0, r1p1.

Default: OFF.

CPSR.T and CPSR.J bits can be corrupted on an asynchronous break. The fix causes the debugger to replace the asynchronous break by a synchronous break via breakpoint register. Breaks via external DBGRQ signal e.g. from CTI still fail and may not be used.

SYStem.Option.BUGFIX Breakpoint bug fix

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

Default: OFF.

Breakpoint bug fix required on ARM7TDMI-S Rev2:

You need to activate this option when having an ARM7TDMI-S Rev2. The bug is fixed on Rev3 and following. With this option activated and ARM7TDMIS selected as CPU type, we enable the software breakpoint workaround as described in the Arm errata of ARM7TDMI-S Rev2 (“consecutive breakpoint” bug). Software breakpoints are set as undefined opcodes that cause the core to enter the undefined opcode handler. The debugger tries to set a breakpoint at the undef vector (either software or on-chip). When a breakpoint is reached the core will take the undefined exception and stop at the vector. The debugger detects this state and displays the correct registers and CPU state. This workaround is only suitable where undefined instruction trap handling is not being used.

Breakpoint bug fix required on Arm946E-S Rev0, Rev1 and Arm966E-S Rev0, Rev1:
(This is a different bug fix as for the Arm7.) This option will automatically be activated by the TRACE32 software, since the core revision will be read out. On the above revisions the breakpoint code normally used for software breakpoints behave wrong. Having this option active an undefined opcode is used together with an on-chip comparator instead of the breakpoint code.

This option is available on Arm7 and on Arm9, but it has a different meaning.

Format: **SYStem.Option.BUGFIXV4** [**ON** | **OFF**]

Default: OFF.

This option is available on Arm7. You need to activate this option when having an Arm7TDMI-S Rev4.

With this option activated, we replace an asynchronous break, e.g. caused by the “break” command, by a break caused by an on-chip breakpoint range. If the bugfix is not activated when using an Arm7TDMI-S Rev4, the application might be restarted at a wrong address.

There is no known workaround to secure correct behavior of the external DBGRQ input and a program halt caused by an ETM trigger condition. Therefore do not use these features on an Arm7TDMI-S Rev4.

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

Default: OFF.

This option selects the byte ordering mechanism. For correct operation the following three settings must correspond:

- This option
- The compiler setting (-li or -bi compiler option)
- The level of the Arm BIGEND input pin (on Arm7x0T and Arm9x0T and JANUS2 the bit in the CP15 control register)

This option is used for derivatives of the ARM7 and Arm9 family.
The endianness is auto-detected for Arm11.
This option does not apply to Cortex-A/R cores.

SYStem.Option.BOOTMODE Define boot mode

Format: **SYStem.Option.BOOTMODE** *<mode>*

Default: 0.

This option selects a boot mode for the chip.

The command is only available on a few chips providing this feature.

SYStem.Option.CINV Invalidate the cache after memory modification

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

Default: OFF.

If this option is ON the cache is invalidated after memory modifications even when memory is modified by the EPROM Simulator (ESI). This is necessary to maintain software breakpoint consistency.

SYStem.Option.CFLUSH FLUSH the cache before step/go

[SYStem.state window > CFLUSH]

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

Default: ON.

If this option is ON, the cache is invalidated automatically before each **Step** or **Go** command. This is necessary to maintain software breakpoint consistency.

SYStem.Option.CacheParam Define external cache

Only available for: Arm7

Format: **SYStem.Option.CacheParam** *<address_range> <size>*

Define the *<address_range>* and the *<size>* of an external cache.

SYStem.Option.CorePowerDetection Set methods to detect core power

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

<method>: **JtagSEQuence** *<seq_name>* | **none**

Sets and configures methods to detect the power of a core.

The core power is detected when **SYStem.Mode Up** is active or is entered. If a core is not powered, the debugger stays in system mode “Up” but displays the state “running (no power)” in the TRACE32 state line.

At the moment only the method **JtagSEQuence** is available.

JtagSEQuence *<seq_name>*	Enables the detection of the core power via a specified JTAG sequence. The specified JTAG sequence is periodically executed by the debug driver. You can create a JTAG sequence with the command **JTAG.SEQuence.Create**. The debug driver assumes that the core is powered when the JTAG sequence returns zero in the variable **Result0**. In case of an SMP system, use the environment variable **PhysicalCORE** within your JTAG sequence.
JtagSequence none	Disables the detection of the core power via a JTAG sequence.

Example:

```
SYStem.RESet ; resets SYStem settings (unlocks all used JTAG sequences)
SYStem.CPU ARC-HS

; create JTAG sequence for power detection
JTAG.SEQuence.Delete myCorePowerCheck   ; delete old sequence
JTAG.SEQuence.Create myCorePowerCheck   ; create new sequence
JTAG.SEQuence.Add , PrePostRelative  +4.  -4.  +1.  -1.
JTAG.SEQuence.Add , RawShift          4.  0x03  0x00
JTAG.SEQuence.Add , ShiftIrAndExit    4.  0x07
JTAG.SEQuence.Add , RawShift          4.  0x03  0x00
JTAG.SEQuence.Add , ShiftDrAndExit   16.  0x00  Result0
JTAG.SEQuence.Add , RawShift          2.  0x01  0x00
JTAG.SEQuence.Add , ASSIGN   Result0 =  ~ Result0 &  0x0001

; use the new JTAG sequence for detecting the core power
SYStem.Option.CorePowerDetection.JtagSEQuence myCorePowerCheck

; connect to all cores of the chip
SYStem.Mode Attach
```

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

Default: OFF.

Derivatives having a Domain Access Control Registers (DACR) do not allow the debugger to access memory if the location does not have the appropriate access permission. If this option is activated, the debugger temporarily modifies the access permission to get access to any memory location.

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

Default: ON.

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

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

Use case:

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

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

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

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.

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

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

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.DBGACK DBGACK active on debugger memory accesses

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

Default: ON.

If this option is on the DBGACK signal remains active during memory accesses in debug mode. If the DBGACK signal is used to freeze timers or to disable other peripherals it is strictly recommended to enable this option.

Disabling of this option may be useful for triggering on memory accesses from debug mode (only useful for hardware developers).

This option is not available on the Arm10.

SYStem.Option.DBGNOPWRDWN DSCR bit 9 will be set in debug mode

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

Default: OFF.

If this option is on DSCR[9] will be set while the core is in debug mode and cleared while the user application is running. **SYStem.Option.PWRDWN** will be ignored.

This option is normally not useful. It was implemented for a special customer design.

This option is available on the Arm11.

SYStem.Option.DBGUNLOCK Unlock debug register via OSLAR

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

Default: ON.

This option allows the debugger to unlock the debug register by writing to the Operating System Lock Access Register (OSLAR) when a debug session will be started. If it is switched off the operating system is expected to unlock the register access, otherwise debugging is not possible.

This option is only available on the Cortex-R and Cortex-A.

SYStem.Option.DCDIRTY Bugfix for erroneously cleared dirty bits

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

Default: OFF.

This is a workaround for a chip bug which erroneously clears the dirty bits of a data cache line if there is any write-through forced by the debugger in this line. When the option is active the debugger does not use write-through mode in general. It only forces write through on a program memory write.

This option is only available on the Arm1176, Cortex-R, Cortex-A.

SYStem.Option.DCFREEZE Disable data cache linefill in debug mode

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

Default: ON.

This option disables the data cache linefill while the processor is in debug mode. This avoids that the data cache contents is altered on memory read accesses performed by the debugger. This is especially required if you want to inspect the data cache contents. You can disable this option if you want to cause a burst memory access (e.g. on a data.test command) which only occurs on a cache linefill.

This option is available on Arm11, only.

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 **SystemLOG.List** window.

Format: **SYStem.Option.DisMode** *<option>*

<option>: **AUTO**
ACCESS
ARM
THUMB
THUMBEE

Default: AUTO.

This command specifies the selected disassembler.

AUTO 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).

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

This option is only available on XScale.

Default: OFF.

If this option is ON and a trap occurs the trap vector is read from memory and the trap vector is executed out of the memory.

The vector tables have be overloaded by the debugger to place the debug vector instead of the reset vector. If the application changes the vector during run-time the overloaded vector table in the mini instruction cache of the debugger remains active and a trap will jump to unintended position.
With **SYStem.Option.DynVector** trap vector contents are read at run-time and the memory is executed. Executing an application with **SYStem.Option.DynVector ON** has disadvantage on run-time, so that it makes sense to switch off the option after the table has changed and afterwards remains unchanged. We have implemented this by an explicit option to be non intrusive on normal operation.

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.ETBFIXMarvell Read out on-chip trace data

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

Default: OFF

Bugfix for 88FR111 from Marvell. At least the first core revisions have an issue with the ETB read/write pointer. ON activates a different method to read out the on-chip trace data.

SYStem.Option.ETMFIX Shift data of ETM scan chain by one

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

Default: OFF.

Bug fix for ETM7 implementations showing a wrong shift behavior. The ETM register data will be shifted by one bit otherwise. This feature is only available on the Arm7 family.

SYStem.Option.ETMFIXWO Bugfix for write-only ETM register

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

Default: OFF.

Bug fix for a customer device where ETM registers can not be read. This fix is only useful on this certain device.

SYStem.Option.ETMFIX4 Use only every fourth ETM data package

Format: **SYStem.Option.ETMFIX4** [**ON** | **OFF**]

Default: OFF.

Bug fix for a customer device where each ETM data package was sent out four times.

SYStem.Option.EXEC EXEC signal can be used by bustrace

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

Default: OFF.

Defines whether the EXEC line is available to the bustrace or not. The EXEC signal indicates if a fetched command has been executed. The bustrace can work without EXEC signal, but it is not possible to show the condition code pass/fail for conditional instructions. The option has no effect when no bustrace is available. This command has no meaning for the ETM trace.

SYStem.Option.EXTBYPASS Switch off the fake TAP mechanism

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

Default: ON.

Bugfix for DB8500 V1. It allows you to switch off the fake TAP mechanism of the modem.

SYStem.Option.FASTBREAKDETECTION Fast core halt detection

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

Default: OFF.

It advises the debugger to do a permanent polling via JTAG to check if the core has halted. This allows a faster detection and generation of trigger signal for other tools like PowerIntegrator, especially if the hardware signal DBGACK is not available on the JTAG connector. It causes a high payload on the JTAG interface which will be a disadvantage e.g. if other debuggers use the same JTAG interface (multicore debugging).

This option is available on Arm9, only.

SYStem.Option.HRCWOVerRide Enable override mechanism

Format: **SYStem.Option.HRCWOVerRide** [**ON** | **OFF**] [**/NONE** | **/PORESET**]

Default: OFF.

Enables the Hardcoded Reset Configuration Word override mechanism for NXP/Freescale Layerscape/QorIQ devices. The feature is required e.g. to program the flash in cases where the flash content is empty or corrupted.

In order to use this functionality, please contact Lauterbach for more details.

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

Default: OFF.

Bugfix for 88FR111 from Marvell. ON locks the usage of read-only/write-only on-chip breakpoints. They do not work on the 88FR111, at least not on the first core revisions.

SYStem.Option.ICEPICK Enable/disable assertions and wait-in-reset

Format: **SYStem.Option.ICEPICK** *<option>*

<option>: **SystemReset.**[**ON** | **OFF**]
WaitInReset.[**ON** | **OFF**]

Default: SystemReset.ON WaitInReset.ON may be preset with the correct parameters for known SoCs in TRACE32.

SystemReset	Enables/disables the assertions of **SystemReset** using the TI-ICEPick. • **ON**: Enables the assertion of **SystemReset**. • **OFF**: Disables the assertion of **SystemReset**.
WaitInReset	Enables/disables the TI-ICEPick Wait-In-Reset functionality. This flag allows depending on the SoC implementation to hold a core on the reset vector. • **ON**: Enables the Wait-In-Reset. • **OFF**: Disables the Wait-In-Reset.

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.IRQBREAKFIX Break bugfix by using IRQ

Format: **SYStem.Option.IRQBREAKFIX** *<address>*

The bug shows up on Cortex-A9, Cortex-A9MPCore r0p0, r0p1, r1p0, r1p1.

Default: 0 = OFF.

CPSR.T and CPSR.J bits can be corrupted on an asynchronous break. The bug fix is intended for an SMP multicore debug session where hardware based synchronous break is required. Instead causing an asynchronous break via CTI an IRQ is requested via CTI. There needs to be a breakpoint at the end of the

IRQ routine handling this case. The fix causes the debugger to replace the program counter value by the IRQ link register R14_irq - 4 and the CPSR register by SPSR_irq if the core halts at *<address>*. Everything else like initializing the IRQ and CTI needs to be done by a user script.

SYStem.Option.KEYCODE Define key code to unsecure processor

Format: **SYStem.Option.KEYCODE** *<key>*

Default: 0, means no key required.

Some processors have a security feature and require a key to un-secure the processor in order to allow debugging. The processor will use the specified key on the next debugger start-up (e.g. SYStem.Up) and forgets it immediately. For the next start-up the key code must be specified again.

This option is for example used on TMS570 derivatives to send a 128-bit key code (<key>: two 64-bit words, LSB will be sent first) to the Advanced JTAG Security Module (AJSM) to unlock JTAG if the device was secured.

The same option is also used on older Arm9 based derivatives having a different security mechanism.

SYStem.Option.L2Cache L2 cache used

Format: **SYStem.Option.L2Cache** [**ON** | **OFF**] (deprecated)
Use SYStem.CONFIG.L2CACHE.Type instead.

Default: OFF, means no L2 cache is used.

On certain Marvell derivatives the debugger can not detect if an (optional) level 2 cache is available and used. The information is needed to activate L2 cache coherency operations.

This option is available on Marvell Arm9, Cortex-A.

SYStem.Option.L2CacheBase Define base address of L2 cache register

Format: **SYStem.Option.L2CacheBase** *<base_address>* (deprecated)
Use SYStem.CONFIG.L2CACHE.Base instead.

Default: 0, means no L2 cache implemented.

In case the L2 cache from Arm (L210, L220 and PL310) is available and active on the chip, then the debugger needs to flush and invalidate the L2 cache when patching the program e.g. when setting a software breakpoint. Therefore it needs to know the (physical) base address of the L2 register block.

This option is available on Arm9, Arm11, Cortex-R, Cortex-A.

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.

Format: **SYStem.Option.MACHINESPACES** [**ON** | **OFF** | **HOSTREMAP**]

Default: OFF

Enables the TRACE32 support for debugging virtualized systems. Virtualized systems are systems running under the control of a hypervisor.

After loading a Hypervisor Awareness, TRACE32 is able to access the context of each guest machine. Both currently active and currently inactive guest machines can be debugged.

ON	Addresses are extended with an identifier called machine ID. The machine ID clearly specifies to which host or guest machine the address belongs. The host machine always uses machine ID 0. Guests have a machine ID larger than 0. TRACE32 currently supports machine IDs up to 30. The debugger address translation (**MMU** and **TRANSlation** command groups) can be individually configured for each virtual machine. Individual symbol sets can be loaded for each virtual machine.
OFF	The machine ID support is disabled.
HOSTREMAP Hypervisor FIASCO	**HOSTREMAP** is only relevant for a hypervisor where: • The hypervisor itself uses tasks and • The tasks behave like virtual machines. If **SYStem.Option.MACHINESPACES** is set to **HOSTREMAP**, then such hypervisor tasks are assigned space IDs instead of machine IDs, whereas the real guest machines are assigned machine IDs. **NOTE**: This option requires a suitable Hypervisor Awareness which supports **HOSTREMAP**. You must also set **SYStem.Option.MMUSPACES** to **ON**.

Machine IDs (0 and > 0)

- On Arm CPUs with hardware virtualization, guest machines are running in the non-secure zone (N:) and use machine IDs > 0.
- The hypervisor functionality is usually running in the hypervisor zone (H:) and uses machine ID 0.
- Software running in the secure monitor mode (Z: for Arm32) or EL3 mode (M: for Arm64) is also using machine ID 0.

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.MemStatusCheck Check status bits during memory access

Format:	**SYStem.Option.MemStatusCheck** [**ON**	**OFF**]

Default: OFF.

Enables status flags check during a memory access. The debugger checks if the CPU is ready to receive/provide new data. Usually this is not needed. Only slow targets (like emulations systems) may need a status check.

SYStem.Option.MMUPhysLogMemaccess Memory access preferences

Format:	**SYStem.Option.MMUPhysLogMemaccess** [**ON**	**OFF**]

Controls whether TRACE32 prefers a cached logical memory access over a (potentially uncached) physical memory access to keep caches updated and coherent.

NOTE:	This option should usually not be changed.

ON	A cached logical memory access is used. This option is enabled by default for Armv7 and older cores.
OFF	A (potentially uncached) physical memory access is used. This option is disabled by default for Armv8 because the physical memory can usually be accessed while the caches are still kept coherent.

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.MonitorHoldoffTime Delay between monitor accesses

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

Default: 0.

It specifies the minimum delay between two access to the target debug client in case of run-mode debugging.

SYStem.Option.MPUBYPASS Ignore MPU access permission settings

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

Default: OFF.

Derivatives having a memory protection unit do not allow the debugger to access memory if the location does not have the appropriate access permission. If this option is activated, the debugger temporarily modifies the access permission to get access to the memory location.

SYStem.Option.MultiplesFIX No multiple loads/stores

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

Default: OFF.

Bug fix for derivatives (e.g. Arm946 V1.1) which do not handle multiple loads (LDM) and multiple store (STM) commands properly in debug mode. When activated only single loads/stores are used by the debugger.

SYStem.Option.NODATA No data connected to the trace

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

This option is only necessary if a **Bus Trace** is used.

Default: OFF.

It should be ON, if a trace is connected and data information can not be recorded. Otherwise undefined data will be displayed in the trace records.

SYStem.Option.NOIRCHECK No JTAG instruction register check

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

Default: OFF.

Bug fix for derivatives which do not return the correct pattern on a JTAG instruction register (IR) scan. When activated the returned pattern will not be checked by the debugger. On Arm7 also the check of the return pattern on a scan chain selection is disabled.

This option is only available on Arm7 and Arm9.

The option is automatically activated when using **SYStem.Option.TURBO**.

SYStem.Option.NoPRCRReset Do not cause reset by PRCR

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

Default: OFF.

It causes the debugger not to (additionally) use the soft reset via DBGPRCR register on functions like **SYStem.Up**, **SYStem.Mode Go**, **SYStem.RESetOut**.

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.NoSecureFix — Do not switch to secure mode

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

Default: OFF.

This is a bugfix for customer specific devices which do not allow the debugger to temporarily switch to secure mode while the application is in non-secure mode.

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**.

Format: **SYStem.Option.PC** *<address>*

Default address: 0

After each load or store operation from debug mode the Arm core makes some instruction fetches from memory. These fetches are not necessary for the debugger, but it is not possible to suppress them.

This option allows to specify the base address of these fetches. The fetch address is anywhere within a 64 KByte block that begins at the specified base address. It is necessary to modify this option if these fetches go to aborted memory locations.

This option is not available/required on the Arm10 and Arm11. There are no dummy-fetches on Arm10 and Arm11.

SYStem.Option.ProgramAccessFix Program memory access bug fix

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

Default: OFF.

Program memory bug fix implemented for a certain core.

SYStem.Option.PROTECTION Sends an unsecure sequence to the core

Format: **SYStem.Option.PROTECTION** *<file>*

This option was made for certain Arm9 derivatives having a protected access to the debug features. It sends the key pattern in the file in a certain way to the core in order to gain the right to debug the core.

This option is available on Arm9.

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

Default: ON.

In case of a chip level TAP (SYStem.CONFIG MULTITAP) this option decides if power, clock and secure state will be checked or not.

This option is only available on Arm11, Cortex-R, Cortex-A.

SYStem.Option.PWRCHECKFIX Check power and clock

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

Default: OFF.

Fix for a certain chip bug: It uses the OSLK bit instead of the SPD bit of the PRSR register to detect power down.

This option is only available on Cortex-R, Cortex-A.

SYStem.Option.PWRDWN Allow power-down mode

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

Default: OFF.

Arm11: If this option is OFF, the debugger sets the external signal **DBGNOPWRDWN** high in order to force the system power controller in emulate mode. Otherwise the communication to the debugger gets lost when entering power down state.

Some OMAPxxxx derivatives: If this option is OFF, the debugger forces the OMAP to keep clock and keep power.

Cortex-R, Cortex-A: Controls the PWRDWN bit in device power-down and reset control register (PRCR).

This option is only available on Arm11, Cortex-R, Cortex-A.

SYStem.Option.PWRDWNRecover Mode to handle special power recovery

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

Default: OFF.

Assumes **SYStem.JtagClock RTCK** is selected.

When the target core is running and RTCK stops working for longer than specified by **SYStem.Option.PWRDWNRecoverTimeout** it is assumed power is gone. In this case “running (power down)” will be shown. On power recovery the target logic ensures the core immediately enters debug mode by asserting DBGRQ signal. The debugger detects the recovery, restores all debug register and restarts the program execution.

This option is only available on Arm9.

SYStem.Option.PWRDWNRecoverTimeOut Timeout for power recovery

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

Specifies a timeout period as a limit to decide if just a sleep mode was entered (stopped RTCK) or a real power down happened which requires the debug registers to be restored on a power recovery. See command **SYStem.Option.PWRDWNRecover**.

This option is only available on Arm9.

SYStem.Option.PWROVR Specifies power override bit

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

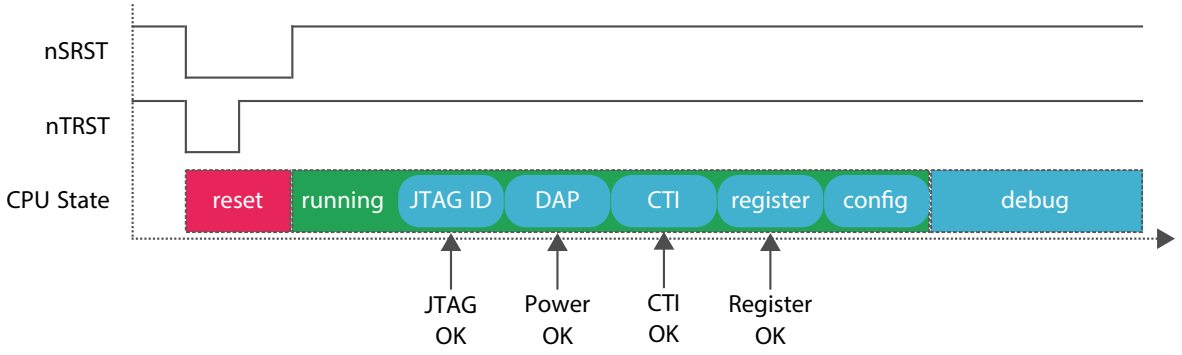
Specifies the power override bit when a certain derivative providing this function is selected.

This option is only available on certain Arm9 and Arm11 derivatives.

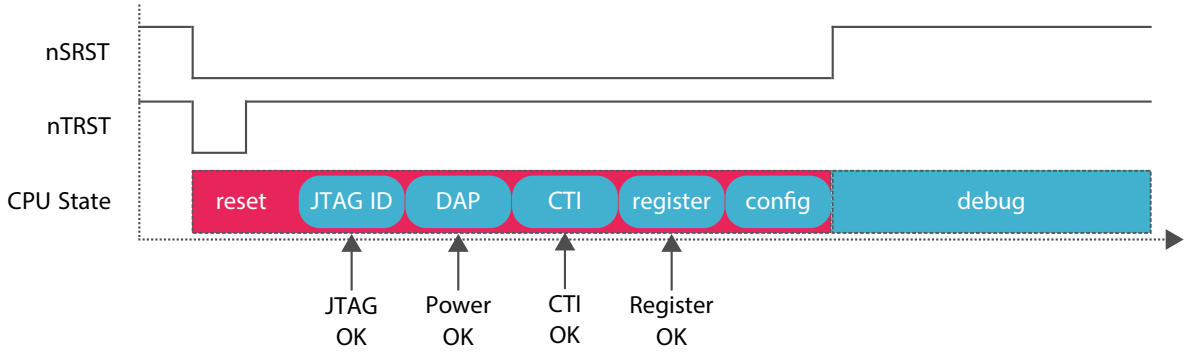
Format: **SYStem.Option.ResBreak** [**ON** | **OFF**]

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.



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

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

Default: nSRST

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

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

None Detection of external resets is disabled.

SYStem.Option.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.RESTARTFIX Wait after core restart

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

Default: OFF.

Bug fix for a certain customer derivative. When activated the debugger keeps the JTAG state machine on every restart for 10 µs in Run-Test/Idle state before the JTAG communication will be continued. This option is available on Arm7 and will be ignored on other debuggers.

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.

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

Default: ON.

If the ABORT (if AMBA: BERROR) line becomes active during a system speed access the Arm core can change to ABORT mode. When this option is on this change of mode is indicated by the warning '**emulator berr error**'.

This option is not available on the Arm10 and Arm11 (always shown).

SYStem.Option.SLaVeSOFTRESet Allow soft reset of slave cores

[build 167397 - DVD 09/2024]

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

Default: OFF.

Allow the debugger to do a soft reset of a slave core during **SYStem.Up**. the availability of this soft reset mechanism depends on the target core type and is also implementation defined. Only set to **ON** when the reset event on a slave core is not distributed to other cores, e.g. by a reset controller.

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.

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

Default: OFF.

Activate this option if software breakpoints should be written by 64-bit accesses. This was implemented in order not to corrupt ECC.

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.SPLIT Access memory depending on CPSR

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

Default: OFF.

If this option is ON, the debugger does privileged or non-privileged memory access depending on the current CPU mode (CPSR register). If this option is OFF, the debugger accesses the memory in privileged mode except another access mode is requested. This feature is only available if a DEBUG INTERFACE (LA-7701) is used for the Arm7.

Format: **SYStem.Option.StandByTraceDelaytime** *<delay_in_us>*

Default: 0.

Only when standby mode is active you can specify a time delay where the debugger waits after reset is deasserted before it activates the trace. This option is available on Arm9 only.

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).

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

Default: OFF.

If this option is active the debugger sends a special initialization sequence, which is required for some derivatives from Texas Instruments (TI) to enable the on-chip debug support. When a TI CPU type (e.g. “OMAP1510”) is selected, this option is automatically set.

This option is only available on Arm9.

SYStem.Option.TIETMFIX Bug fix for customer specific ASIC

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

SYStem.Option.TIDEMUXFIX Bug fix for customer specific ASIC

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

SYStem.Option.TraceStrobe Deprecated command

Format: **SYStem.Option.TraceStrobe** [**CE** | **OE** | **CE+OE** | **STR** | **STR-**] (deprecated)

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.TURBO Speed up memory access

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

Default: OFF.

If TURBO is disabled the CPU checks after each system speed memory access in debug mode if the CPU has finished the corresponding cycle. This check will significantly reduce the down- and upload speed (30-40%).

If TURBO is enabled the CPU will make no checks. This may result in unpredictable errors if the memory interface is slow. Therefore it is recommended to use this option only for a program download and in case you know that the memory interface is fast enough to take the data with the speed they are provided by the debugger.

This option is not available on the Arm10.

SYStem.Option.WaitIDCODE IDCODE polling after deasserting reset

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

Default: OFF = disabled.

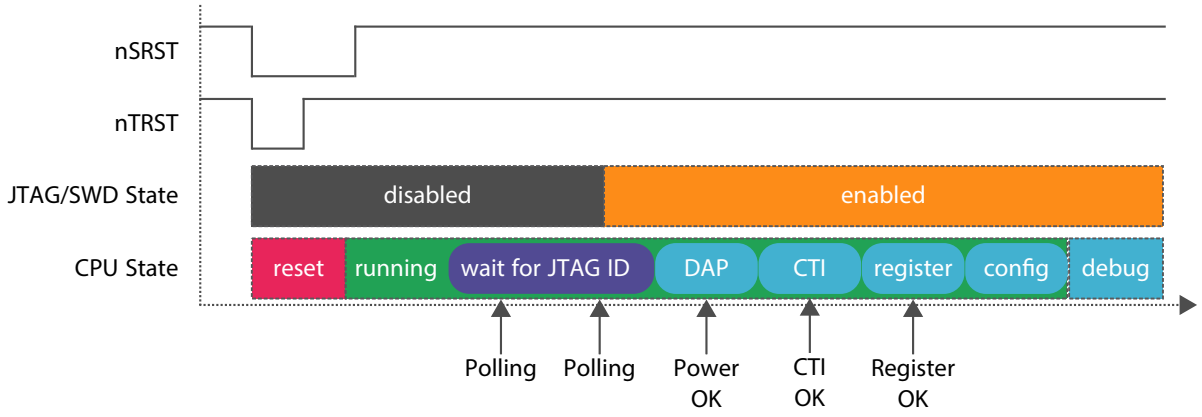
Allows to add additional busy time after reset. The command is limited to systems that use an Arm DAP.

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**.

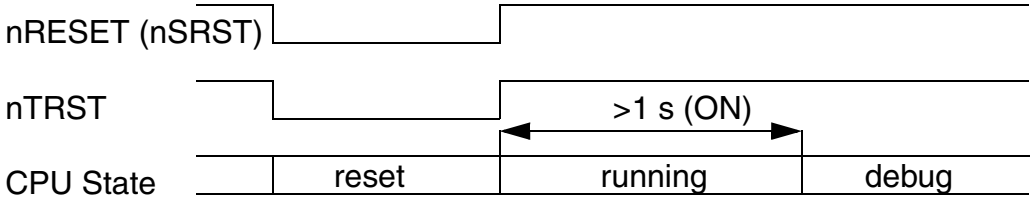
If **SYStem.Option.ResBreak** is disabled, **SYStem.Option.WaitReset** can be used to specify a waiting time between the deassertion of nSRST and nTRST and the first JTAG activity. During this time the core may execute some code, e.g to enable the JTAG port.

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.WATCHDOG Disable watchdog while debugging

Format:	**SYStem.Option.WATCHDOG** [**DEFault** | **OFF**]

Default: DEFault

Enables/disables the internal watchdog on some devices on connection time, e.g. during **SYStem.Up**. The option is available on some Spansion/Cypress S6J devices. Please refer to the example scripts if the option is available.

DEFault	Does not modify the watchdog configuration.
OFF	Disables the watchdog when connecting.

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

Default: OFF.

The **SYStem.Option.ZoneSPACES** command must be set to **ON** if an Arm CPU with TrustZone or VirtualizationExtension is debugged. In these Arm CPUs, the processor has two or more CPU operation modes called:

- Non-secure mode
- Secure mode
- Hypervisor mode
- 64-bit EL3/Monitor mode (Armv8-A only)

Within TRACE32, these CPU operation modes are referred to as zones.

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

In each CPU operation mode (zone), the CPU uses separate MMU translation tables for memory accesses and separate register sets. Consequently, in each zone, different code and data can be visible on the same logical addresses.

To ease debug-scenarios where the CPU operation mode switches between non-secure, secure or hypervisor mode, it is helpful to load symbol sets for each used zone.

OFF	TRACE32 does not separate symbols by access class. Loading two or more symbol sets with overlapping address ranges will result in unpredictable behavior. Loaded symbols are independent of Arm zones.
ON	Separate symbol sets can be loaded for each zone, even with overlapping address ranges. Loaded symbols are specific to one of the Arm zones - each symbol carries one of the access classes N:, Z:, H: or M: For details and examples, see below.

If **SYStem.Option.ZoneSPACES** is enabled (**ON**), TRACE32 enforces any memory address specified in a TRACE32 command to have an access class which clearly indicates to which zone the memory address belongs. The following access classes are supported:

N	Non-secure mode Example: Linux user application
Z	Secure mode Example: Secure crypto routine
H	Hypervisor mode Example: XEN hypervisor
M Armv8-A only	64-bit EL3/Monitor mode Example: Trusted boot stage / monitor

If an address specified in a command is not clearly attributed to N: Z:, H: or M:, the access class of the current PC context is used to complete the addresses' access class.

Every loaded symbol is attributed to either non-secure (N:), secure (Z:), hypervisor (H:) or EL3/monitor (M:) zone. If a symbol is referenced by name, the associated access class (N:, Z:, H: or M:) will be used automatically, so that the memory access is done within the correct CPU mode context. As a result, the symbol's logical address will be translated to the physical address with the correct MMU translation table.

NOTE:	The loaded symbols and their associated access class can be examined with command **sYmbol.List** or **sYmbol.Browse** or **sYmbol.INFO**.

Example: Symbols Loading

```
SYStem.Option.ZoneSPACES ON

; 1. Load the vmlinux symbols for non-secure mode (access classes N:, NP:
; and ND: are used for the symbols) with offset 0x0:
Data.LOAD.Elf vmlinux N:0x0 /NoCODE

; 2. Load the sysmon symbols for secure mode (access classes Z:, ZP: and
; ZD: are used for the symbols) with offset 0x0:
Data.LOAD.Elf sysmon Z:0x0 /NoCODE

; 3. Load the xen-syms symbols for hypervisor mode (access classes H:,
; HP: and HD: are used for the symbols) but without offset:
Data.LOAD.Elf xen-syms H: /NoCODE

; 4. Load the sieve symbols without specification of a target access
; class and address:
Data.LOAD.Elf sieve /NoCODE
; Assuming that the current CPU mode is non-secure in this example, the
; symbols of sieve will be assigned the access classes N:, NP: and ND:
; during loading.
```

Example: Symbolic Memory Access

```
; dump the address on symbol swapper_pg_dir which belongs
; to the non-secure symbol set "vmlinux" we have loaded above:

Data.dump swapper_pg_dir

; This will automatically use access class N: for the memory access,
; even if the CPU is currently not in non-secure mode.
```

Example: Deleting Zone-specific Symbols

To delete a complete symbol set belonging to a specific zone, e.g. the non-secure zone, use the following command to delete all symbols in the specified address range.

```
sYmbol.Delete N:0x0--0xffffffff   ; non-secure mode (access classes N:)
```

Example: Zone-specific Debugger Address Translation Setup

If the option **ZoneSPACES** is enabled and the debugger address translation is used (**TRANSlation** commands), a strict zone separation of the address translations is enforced. Also, common address ranges created with **TRANSlation.COMMON** will always be specific for a certain zone.

This script shows how to define separate translations for the zones **N:** and **H:**

```
SYStem.Option.ZoneSPACES ON

Data.LOAD.Elf sysmon   Z:0 /NoCODE
Data.LOAD.Elf usermode N:0 /NoCODE /NoClear

; set up address translation for secure mode
TRANSlation.Create Z:0xC0000000++0x0fffffff A:0x10000000

; set up address translation for non-secure mode
TRANSlation.Create N:0xC0000000++0x1fffffff A:0x40000000

; enable address translation and table walk
TRANSlation.ON

; check the complete translation setup
TRANSlation.List
```

If the CPU's virtualization extension is used to virtualize one or more guest systems, the hypervisor always runs in the CPU's hypervisor mode (zone H:), and the current guest system (if a ready-to-run guest is configured at all by the hypervisor) will run in the CPU's non-secure mode (zone N:).

Often, an operation system (such as a Linux kernel) runs in the context of the guest system.

In such a setup with hypervisor and guest OS, it is possible to load both the hypervisor symbols to H: and all OS-related symbols to N:

A TRACE32 OS Awareness can be loaded in TRACE32 to support the work with the OS in the guest system. This is done as follows:

1. Configure the OS Awareness as for a non-virtualized system. See:
   - **"Training Linux Debugging"** (training_rtos_linux.pdf)
   - **TASK.CONFIG** command
2. Additionally set the default access class of the OS Awareness to the non-secure zone:

```
TASK.ACCESS N:
```

   The TRACE32 OS Awareness is now configured to find guest OS kernel symbols in the **non-secure** zone.

NOTE:	This debugger setup, which is based on the option **ZoneSPACES**, allows work with only one guest system simultaneously. If the hypervisor has configured more than one guest, only the guest that is active in the non-secure CPU mode is visible. To work with another guest, the system must continue running until an inactive guest becomes the active guest. With **SYStem.Option.MACHINESPACES** enabled, TRACE32 also supports concurrent debugging of a virtualized system with hypervisor and multiple guests. the CPU specific zones N: Z: H: and M: will be extended by machine specific zones. Each of these zones is identified by a machine ID. Each guest has its own zone because it uses a separate translation table and a separate register set.

Example: Setup for a Guest OS and a Hypervisor

In this script, the hypervisor is configured to run in zone **H:** and a Linux kernel with OS Awareness as current guest OS in zone **N:**

```
SYStem.Option.ZoneSPACES ON

; within the OS Awareness we need the space ID to separate address spaces
; of different processes / tasks
SYStem.Option.MMUSPACES ON

; here we let the target system boot the hypervisor. The hypervisor will
; set up the guest and boot Linux on the guest system.
...

; load the hypervisor symbols
Data.LOAD.Elf xen-syms H:0 /NoCODE
Data.LOAD.Elf usermode N:0 /NoCODE /NoClear

; set up the Linux OS Awareness
TASK.CONFIG    ~~/demo/arm/kernel/linux/linux-3.x/linux3.t32
MENU.ReProgram ~~/demo/arm/kernel/linux/linux-3.x/linux.men

; instruct the OS Awareness to access all OS-related symbols with
; access class N:
TASK.ACCESS N:

; set up the debugger address translation for the guest OS

; Note that the default address translation in the following command
; defines a translation of the logical kernel addresses range
; N:0xC0000000++0xFFFFFFF to the intermediate address range
; starting at I:0x40000000
MMU.FORMAT linux swapper_pg_dir N:0xC0000000++0xFFFFFFF I:0x40000000

; define the common address range for the guest kernel symbols
TRANSlation.COMMON N:0xC0000000--0xFFFFFFFF

; enable the address translation and the table walk
TRANSlation.TableWalk ON
TRANSlation.ON
```

NOTE: If **SYStem.Option.MMUSPACES ON** is used, all addresses for all zones will show a space ID (such as N:**0x024A**:0x00320100), even if the OS Awareness runs only in one zone (as defined with command **TASK.ACCESS**).

Any task-related command, such as **MMU.List.TaskPageTable** *<task_name>*, will automatically refer to tasks running in the same zone as the OS Awareness.

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

Default: OFF

This option is for a Zynq Ultrascale+ device using JTAG Boot mode. There are two cases:

1. Device operates in cascaded mode. The Arm DAP and TAP controllers both use the PL JTAG interface, i.e. forming a JTAG daisy chain.
2. Device operates in independent mode. The TAP controller is accessed via the PL JTAG interface. The Arm DAP is connected to the MIO or EMIO JTAG interface.

This command controls whether the debugger connects to the device in independent or cascaded mode. This depends on the used JTAG interface.

ON	The Arm DAP is accessed through the MIO or EMIO JTAG interface. No JTAG chain configuration is required by the debugger. **NOTE**: Please set this option to **ON** if JTAG is connected via the independent JTAG (e.g. via MIO or EMIO via FPGA) lines.
OFF	The Arm DAP is accessed through the PL JTAG interface and has to be chained with the TAP controller by the debugger.

SYStem.RESetOut Assert nRESET/nSRST on JTAG connector

[SYStem.state window > RESetOut]

Format:	**SYStem.RESetOut**

If possible (nRESET/nSRST is open collector), this command asserts the nRESET/nSRST line on the JTAG connector. While the CPU is in debug mode, this function will be ignored. Use the **SYStem.Up** command if you want to reset the CPU in debug mode.

Format: **SYStem.state**

Displays the **SYStem.state** window for Arm.

SYStem.Option.HRCWOVerRide()

[build 144077 - DVD 02/2022]

Syntax: **SYStem.Option.HRCWOVerRide()**

Returns current setting of **SYStem.Option.HRCWOVerRide**.

Return Value Type: Boolean.

Return Value and Description:

TRUE	**SYStem.Option.HRCWOVerRide ON** has been set.
FALSE	**SYStem.Option.HRCWOVerRide OFF** has been set.

Arm Specific Benchmarking Commands

The **BMC** (**B**ench**M**ark **C**ounter) commands provide control of the on-chip performance monitor unit (PMU). The PMU consists of a group of counters that can be configured to count certain events in order to get statistics on the operation of the processor and the memory system.

The counters of Cortex-A/R cores can be read at run-time. The counters of Arm11 cores can only be read while the target application is halted. This group of counters is not available for Arm7 to Arm10 cores.

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.

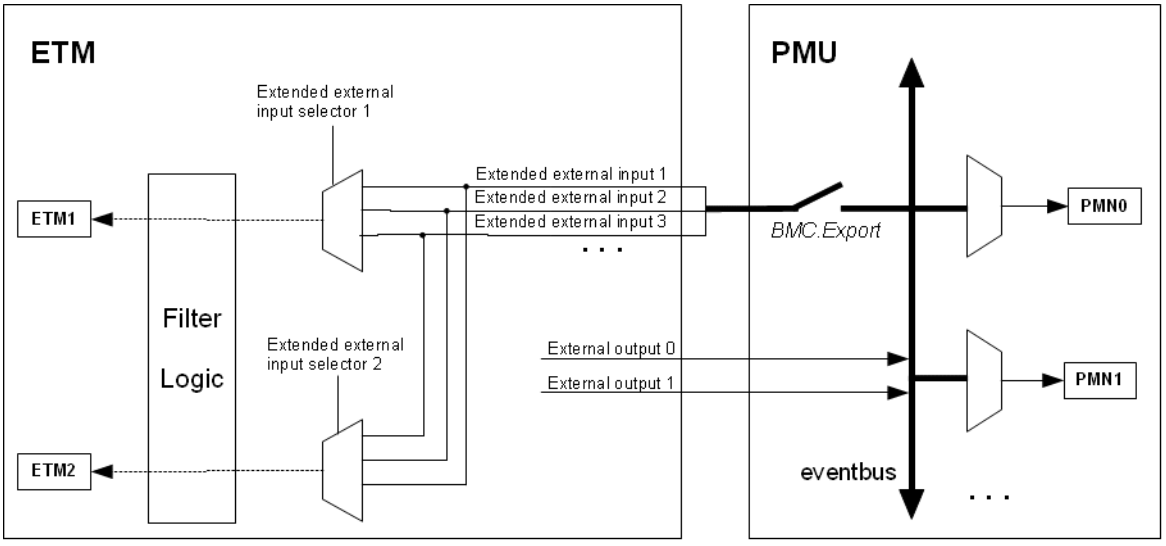
BMC.EXPORT Export benchmarking events from event bus

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

Enable / disable the export of the benchmarking events from the event bus. If enabled, it allows an external monitoring tool, such as an ETM to trace the events. For further information please refer to the target processor manual under the topic performance monitoring.

Default: OFF

The figure below depicts an example configuration comprising the PMU and ETM:

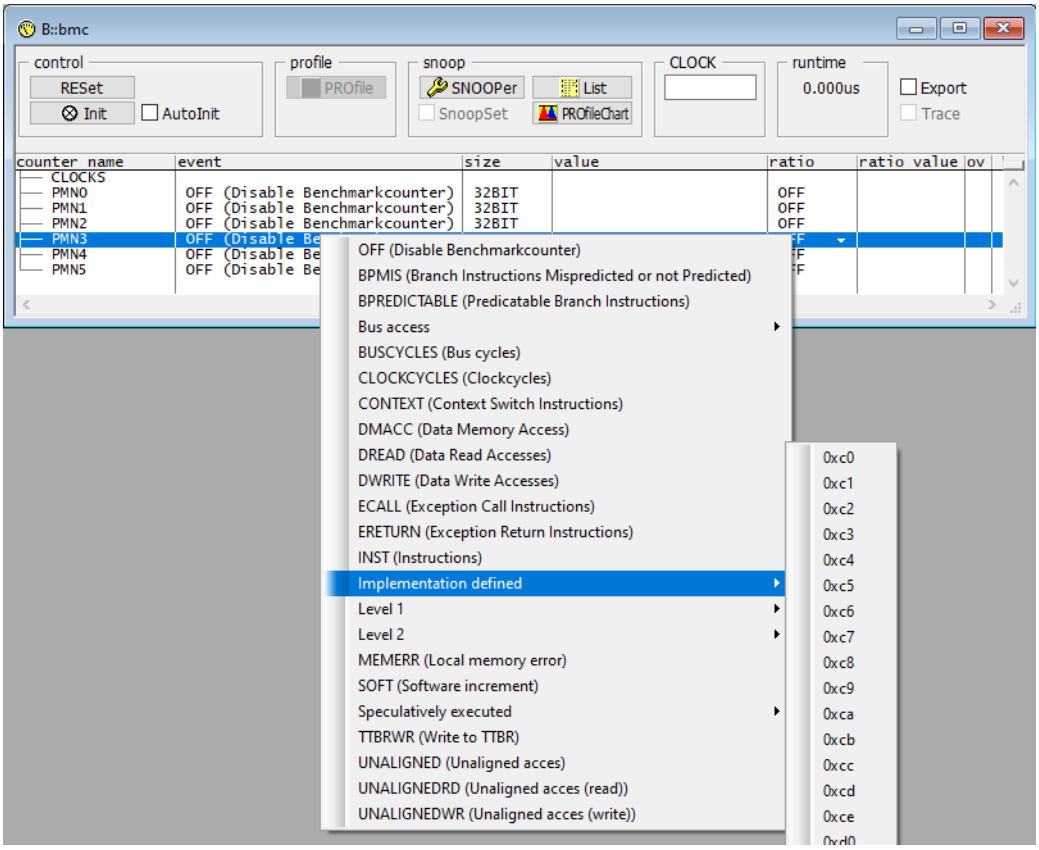


In case ETM1 or ETM2 are selected for event counting, **BMC.EXPORT** will automatically be switched on. Furthermore the according extended external input selectors of the ETM will be set accordingly.

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

CortexA15 only. SoC manufacturers can define their own events to be counted on CortexA15 devices. These custom events can be placed within ID range 0xC0 - 0xFF.

Event names may differ between manufacturers (or even between devices from the same manufacturer), so these IDs will appear as event names in the pulldown list and command path.



Format:	**BMC.MODE** *<mode>*
<mode>:	**OFF** **ICACHE** **DCACHE** **SYSIF** **CLOCK** **TIME**

This command only applies to some Arm9 based derivatives from Texas Instruments.

The Benchmark Counter - short BMC - is a hardware counter. It collects information about the throughput of the target processor, like instruction or data cache misses. This information may be helpful in finding bottlenecks and tuning the application.

CLOCK	Incremented for each CPU clock.
ICACHE	Counts Instructions CACHE misses, in relation to total instruction access.
DCACHE	Counts Data CACHE misses, in relation to total data access.
OFF	Switch off the benchmark counter.
SYSIF	Counts if SYStem bus InterFace is busy, in relation to total system bus access.
TIME	TIME is measured by counting CLOCK. The translation to TIME is done by using the CPU frequency. For this reason, the CPU frequency has to be entered with the command **BMC.CLOCK**.

Format:	**BMC.***<counter>***.EVENT** *<event>*
<counter>:	**PMN0** **PMN1**
<event>:	**OFF** | **INST** | **BINST** | **BMIS** | **PC** | **ICMISS** | **ITLBMISS** | **ISTALL** | **DACCESS** | **DCACHE** | **DCMISS** | **DTBLMISS** | **DSTALL** | **DFULL** | **DCWB** | **WBDRAIN** | **TLBMISS** | **EMEM** | **ETMEXTOUT0** | **ETMEXTOUT1** | **Delta** | **Echo** | **CLOCK** | **TIME** | **NONE** | …

The command is available on Arm1136, Arm1176 and Cortex cores. This description applies to Arm1136. All available events are described in detail in the technical reference guide of the Arm cores.

Performance Monitors - short PMN - are implemented as 32-bit hardware counter. They collect information about the throughput of the target processor and its pipeline stages. They count certain events, like cache misses or CPU cycles. Further, they deliver information about the efficiency of the instruction or data cache, the TLBs (translation look aside buffers) and some other performance values. This information may be helpful in finding bottlenecks and tuning the application.

<event>	For a description of the *<events>*, refer to the *Technical Reference Manual* (TRM) of the respective core, chapter "*Performance Monitor Unit*" (PMU). For a description of some selected *<events>*, see below.
OFF	Switch off the performance monitor.
INST	The selected counter counts executed instructions.
BINST	Counts executed branch instructions.
BMIS	Counts branches which were mispredicted by the core (for static) or prefetch unit (for dynamic) branch prediction. A branch misprediction causes the pipeline to be flushed, and the correct instruction to be fetched.
PC	Counts changes of the PC by the program e.g. as in a MOV or LDR instruction with PC as destination.
ICMISS	Counts instruction cache misses which requires a instruction fetch from the external memory.
ITLBMISS	Counts misses of the instruction MicroTLB.

ISTALL	ISTALL increments the counter by 1 for every cycle the condition is valid. The CPU is stalled when the instruction buffer cannot deliver an instruction. This happens as a result of an instruction cache miss or an instruction MicroTLB miss.
DACCESS	DACCESS is incremented by 1 for every nonsequential data access, regardless of whether or not the item is cached or not.
DCACHE	DCACHE is incremented for each access to the data cache.
DCMISS	DCMISS counts for missing data in the data cache.
DTBLMISS	Counts misses in the data MicroTLB.
DSTALL	In a data dependency conflict the CPU is stalled. DSTALL increments the counter by one for every cycle the stall persists.
DFULL	If the pipeline of load store unit is full, the counter will be incremented by one for each clock the condition is met.
DCWB	Data cache write back occurs for each half line of four words that are written back from cache to memory.
WBDRAIN	Write buffer drains force all buffered data writes to be written to external memory. WBDRAIN will count all that drains which are done because of a data synchronization barrier or strongly ordered operations.
TBLMISS	Counts main TLB misses.
EMEM	Incremented for each explicit external data access. That includes cache refills, non-cashable and write-through access. It does not include instruction cache fills or data write backs.
ETMEXTOUT0	The counter is incremented, if the ETMEXTOUT0-signal is asserted for a cycle. The ETM can be programmed to rise that signal on behalf / as result of certain events, like a counter overflow or an address compare.
EMTEXTOUT1	The counter is incremented, if the ETMEXTOUT1-signal is asserted for a cycle. The ETM can be programmed to rise that signal on behalf of certain events, like a counter overflow or an address compare.
Delta	Counts hits of the Delta-Marker, if specified.
Echo	Counts hits of the Echo-Marker, if specified.
CLOCK	The counter is incremented for every cpu clock.

TIME	TIME is measured by counting CLOCK. The transaction to TIME is done by using the cpu frequency. For this reason, the CPU frequency has to be entered with the command **BMC.CLOCK**.
INIT	Reset the benchmark counter to zero.

Example 1: To count for branches taken, in relation to mispredicted branches, use the following commands:

```
BMC.RESet                    ; Reset the BMC settings

BMC.state                    ; Display the BMC window

BMC.PMN0.EVENT BINST         ; Set the first (PMN0) performance counter
                             ; to count all taken branches

BMC.PMN1.EVENT BMIS          ; Set the second (PMN1) performance counter
                             ; to mispredicted branches

BMC.PMN0.RATIO PMN1/PMN0     ; Calculate the ratio between branches
                             ; taken and branches mispredicted

Go sieve                     ; Go to the function sieve

BMC.Init                     ; Initialize the benchmark counter to start
                             ; the measurement of function sieve

Go.Return                    ; Go to the last instruction of the function
                             ; sieve
```

Example 2: To count for data access in relation to data cache misses:

```
BMC.RESet                    ; Reset the BMC settings

BMC.state                    ; Display the BMC window

BMC.PMN0.EVENT DCACCESS      ; Set the first (PMN0) performance counter
                             ; to count all data accesses

BMC.PMN1.EVENT DCMISS        ; Set the second (PMN1) performance counter
                             ; to count data cache misses

BMC.PMN0.RATIO PMN1/PMN0     ; Calculate the ratio between data access
                             ; and cache misses

Go sieve                     ; Go to the function sieve

BMC.Init                     ; Initialize the benchmark counter

Go.Return                    ; Go to the last instruction of the function
                             ; sieve
```

Benchmark counter values can be returned with the function **BMC.COUNTER()**.

BMC.PRESCALER — Prescale the measured cycles

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

If ON, the cycle counter register, which counts for the cpu cycles which is used to measure the elapsed time, will be divided (prescaled) by 64. The display of the time will be corrected accordingly.

BMC.TARA — Calibrate the benchmark counter

Format: **BMC.TARA**

Due to restricted technical feasibility, the benchmark counter will start counting before the application runs. To improve the exactness of the result you can perform **BMC.Init**, single step an assembler command and execute **BMC.TARA**. On following measurements the obtained result will be subtracted from the benchmark counter.

Arm Specific TrOnchip Commands

The **TrOnchip** command group provides low-level access to the on-chip debug register.

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.MatchMachine (deprecated)	**Break.CONFIG.MatchMachine**
TrOnchip.MatchZone (deprecated)	**Break.CONFIG.MatchZone**
TrOnchip.ContextID (deprecated)	**Break.CONFIG.UseContextID**
TrOnchip.MachineID (deprecated)	**Break.CONFIG.UseMachineID**
TrOnchip.VarCONVert (deprecated)	**Break.CONFIG.VarConvert**

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

TrOnchip.A Programming the ICE breaker module

Available for Arm7 and Arm9 family.

Example: Assume there is a byte variable called 'flag' and you want to trigger if the value 59 is written to the variable.

```
Break.Set flag /Alpha          ; set an alpha breakpoint to the address
                               ; of the variable flag

TrOnchip.A Address Alpha       ; enable alpha break for on-chip trigger

TrOnchip.A Value 0xxxxxx59     ; specify data pattern; this example
                               ; assumes that the address of flags is on
                               ; an address dividable by 4 and you have
                               ; little endian byte ordering (lowest byte
                               ; on data bus)

TrOnchip.A Cycle Write         ; specify that you want to trigger only on
                               ; a write access

TrOnchip.A Size Byte           ; specify that you want to trigger only on
                               ; byte access
```

TrOnchip.A.Value — Define data selector

Format: **TrOnchip.A.Value** *<hexmask>* | *<bitmask>*
TrOnchip.B.Value *<hexmask>* | *<bitmask>*

Defines the two data selectors of ICE breaker as hex or binary mask (x means don't care). If you want to trigger on a certain byte or word access you must specify the mask according to the address of the access. E.g. you make a byte access on address 2 and you want to trigger on the value 33, then the necessary mask is 0xx33xxxx.

Available for Arm7 and Arm9 family.

TrOnchip.A.Size — Define access size for data selector

Format: **TrOnchip.A.Size** *<size>*
TrOnchip.B.Size *<size>*

<size>: **OFF**
Byte
Word
Long

Defines on which access size when ICE breaker stops the program execution.

Available for Arm7 and Arm9 family.

TrOnchip.A.CYcle — Define access type

Format: **TrOnchip.A.CYcle** *<cycle>*
TrOnchip.B.CYcle *<cycle>*

<cycle>: **OFF**
Read
Write
Access
Execute

Defines on which cycle the ICE breaker stops the program execution.

OFF	Cycle type does not matter.
Read	Stop the program execution on a read access.
Write	Stop the program execution on a write access.
Access	Stop the program execution on a read or write access.
Execute	Stop the program execution on an instruction is executed.

Available for Arm7 and Arm9 family.

Format: **TrOnchip.A.Address** *<selector>*
TrOnchip.B.Address *<selector>*

<selector>: **OFF**
Alpha
Beta
Charly

The address/range for an address selector can not be defined directly. Set an breakpoint of the type Alpha, Beta or Charly to the address/range.

Example 1:

```
Break.Set 1000 /Alpha            ; set an Alpha breakpoint to 1000
TrOnchip.A.Address Alpha         ; use Alpha breakpoint as address
                                 ; selector for the unit A
```

Example 2:

```
Var.Break.Set flags[3] /Beta     ; set a Beta breakpoint to flags[3]
TrOnchip.B.Address Beta          ; use Beta breakpoint as address
                                 ; selector for the unit B
```

Available for Arm7 and Arm9 family.

Format: **TrOnchip.A.Trans** *<mode>*
TrOnchip.B.Trans *<mode>*

<mode>: **OFF**
User
Svc

Defines in which mode ICE breaker should stop the program execution.

OFF	Mode doesn't matter.
Svc	Stop the program execution only in supervisor mode.
User	Stop the program execution only in user mode.

Available for Arm7 and Arm9 family.

TrOnchip.A.Extern Define the use of EXTERN lines

Format: **TrOnchip.A.Extern** *<mode>*
TrOnchip.B.Extern *<mode>*

<mode>: **OFF**
Low
High

Defines if the EXTERN lines are considered by unit A or unit B.

Available for Arm7 and Arm9 family.

Format: **TrOnchip.AddressMask** *<value>* | *<bitmask>*

TrOnchip.ContextID Enable context ID comparison

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

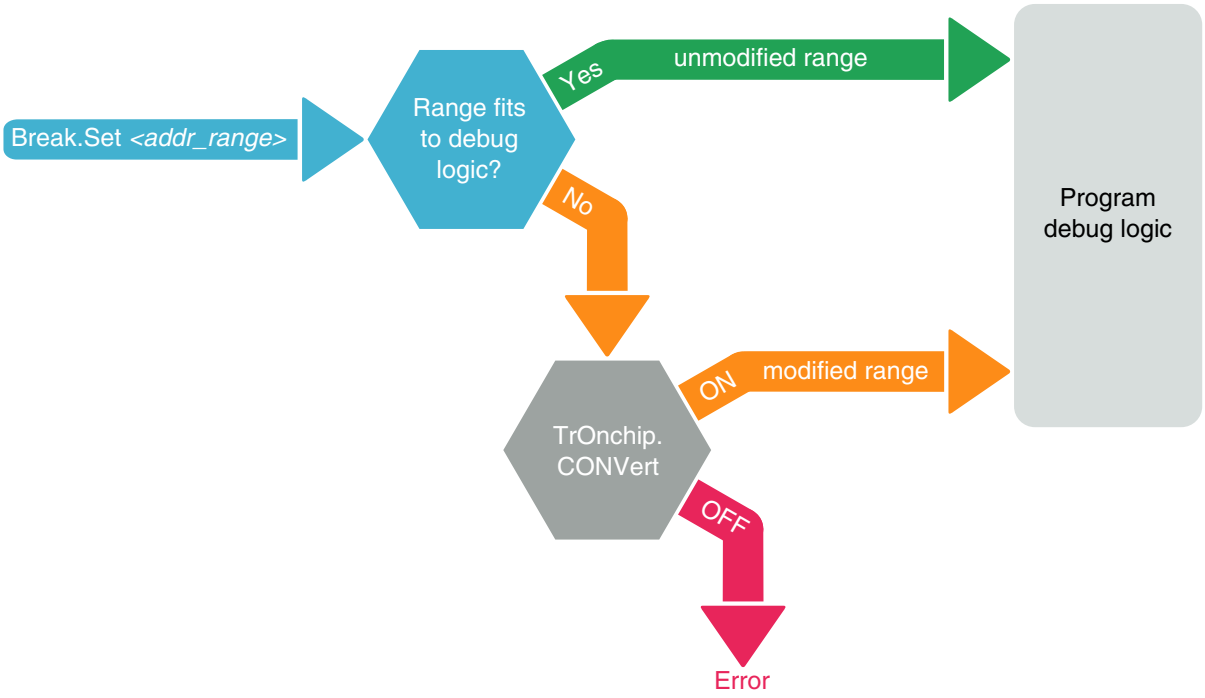
If the debug unit provides breakpoint registers with ContextID comparison capability, **TrOnchip.ContextID** has to be set to **ON** in order to set task/process specific breakpoints that work in real-time.

Example:

```
TrOnchip.ContextID ON

Break.Set VectorSwi /Program /Onchip /TASK EKern.exe:Thread1
```

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.MachineID Extend on-chip breakpoint/trace filter by machine ID

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

If the debug unit provides breakpoint registers with Machine ID comparison capability, **TrOnchip.MachineID** has to be set to **ON** in order to set machine specific breakpoints that work in real-time.

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.

TrOnchip.MatchMachine Extend on-chip breakpoint/trace filter by machine

Format:	**TrOnchip.MatchMachine** [**ON** | **OFF**] (deprecated) **Use Break.CONFIG.MatchMachine 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 machine match. Trace filters and triggers become active if both the address and the machine match.

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

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

NOTE:	**SYStem.Option.ZoneSPACES** must be set to **ON** for **TrOnchip.MatchZone ON** to take effect. However, the setting **TrOnchip.MatchZone ON** is *not* supported by all Arm cores nor by all ETMs.

Example: In these two demo code snippets, let's compare the setting **TrOnchip.MatchZone ON** and **OFF** for an on-chip breakpoint at address 0x100 in zone Z (= secure memory).

```
SYStem.Option.ZoneSPACES ON

;create an on-chip breakpoint in secure memory
Break.Set ZSR:0x100 /Onchip

TrOnchip.MatchZone  ON      ;observe the zones for on-chip breakpoints

;--> application execution will stop at the on-chip breakpoint
;    only if both conditions are fulfilled:
;    a) the address is 0x100 and
;    b) the zone is Z (= secure memory)
```

```
SYStem.Option.ZoneSPACES ON

;create an on-chip breakpoint in secure memory
Break.Set ZSR:0x100 /Onchip

TrOnchip.MatchZone  OFF     ;ignore the zones for on-chip breakpoints

;--> now application execution will stop at address 0x100
;    irrespective of the zone
```

TrOnchip.Mode — Configure unit A and B

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

<mode>:
AORB
AANDB
BAFTERA
WATCH

Defines the way in which unit A and B are used together.

AANDB	Stop the program execution if both units match.
AORB	Stop the program execution if unit A or unit B match.
BAFTERA	Stop the program execution if first unit A and then unit B match.
WATCH	Cause assertion of the internal watchpoint signal on a match.

Available for Arm7 and Arm9 family.

TrOnchip.RESet — Reset on-chip trigger settings

Format: **TrOnchip.RESet**

Resets all TrOnchip settings.

Format:	**TrOnchip. Set** *<item>* [**ON** | **OFF**]
<item> :	ARM9, ARM11, Cortex-A/-R: [**FIQ** | **IRQ** | **DABORT** | **PABORT** | **SWI** | **UNDEF** | **RESET**] Devices having TrustZone (ARM1176, Cortex-A) additionally: [**NFIQ** | **NIRQ** | **NDABORT** | **NPABORT** | **NSWI** | **NUNDEF** | **SFIQ** | **SIRQ** | **SDABORT** | **SPABORT** | **SSWI** | **SUNDEF** | **SRESET** | **MAFIC** | **MIRQ** | **MDABORT** | **MPABORT** | **MSWI**] Devices having a Hypervisor mode (e.g. Cortex-A7, -A15) additionally: [**HFIQ** | **HIRQ** | **HDABORT** | **HPABORT** | **HSWI** | **HUNDEF** | **HENTRY**]

Default: DABORT, PABORT, UNDEF, RESET ON, others OFF.

On devices having TrustZone you can specify for most exceptions if the vector catch shall take effect only in non-secure (N...), secure (S...) or monitor mode (M...), on devices having a Hypervisor mode also in hypervisor mode (H...).

FIQ, ... **HENTRY**	Sets/resets the corresponding bits in the vector catch register of the core. If the bit of a vector is set and the corresponding exception occurs, the processor enters debug state as if there had been a breakpoint set on an instruction fetch from that exception vector.
StepVector (deprecated)	Please see **TrOnchip.StepVector**.

TrOnchip.StepVector Step into exception handler

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

Default: OFF.

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

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

Default: OFF.

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

Format:	**TrOnchip.TEnable** *<mode>*
<mode>:	**ALL** **Alpha** **Beta** **Charly** **Delta** **Echo**

Defines a filter for the trace. The Preprocessor for the Arm7 family (bus trace) provides 1 address comparator that is implemented as a comparator (bit mask). Since this comparator is provided by the TRACE32 development tools, it is listed as a Hardware Breakpoint.

Example 1: Sample only entries to the function `sieve`.

```
Break.Set sieve /Charly
TrOnchip.TEnable Charly
TrOnchip.TCYcle Fetch
```

Example 2: Sample all read and write accesses to the variable `flags[3]`.

```
Var.Break.Set flags[3] /Alpha
TrOnchip.TEnable Alpha
TrOnchip.TCYcle Access
```

Format: **TrOnchip.TCYcle** *<cycle>*

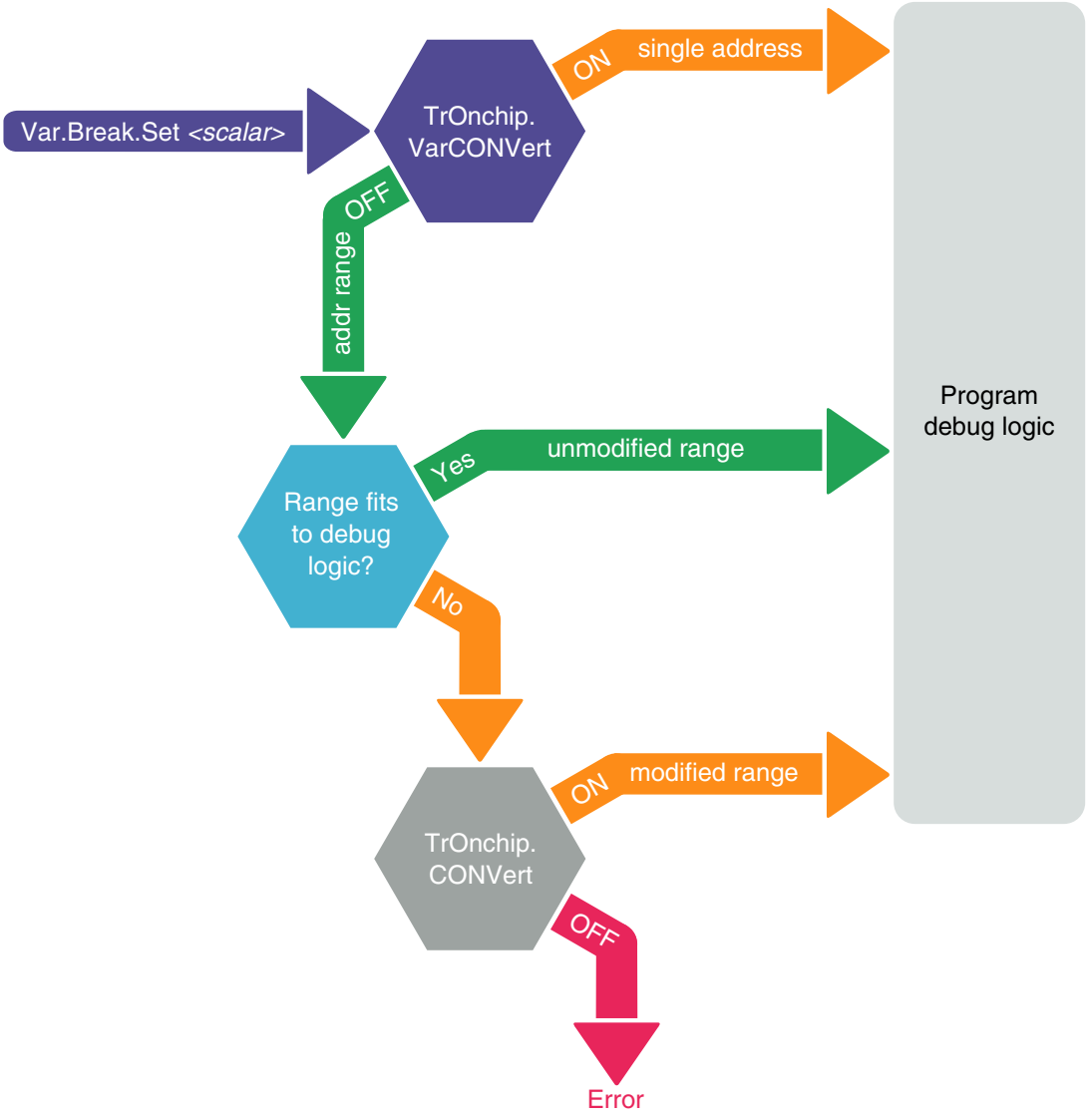
<cycle>: **ANY**
Read
Write
Access
Fetch
Soft

Defines the cycle type for the bus trace address selector.

ANY	Cycle type doesn't matter.
Read	Record only read accesses.
Write	Record only write accesses.
Access	Record only data accesses.
Fetch	Record only instruction fetches.
Soft	Not used now.

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.

TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

MMU.DUMP Page wise display of MMU translation table

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

Displays the contents of the CPU specific MMU translation table.

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

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

TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Displays the MMU translation table entries of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*	The following options are only available if **SYStem.Option.MACHINESPACES** is set to **ON**. Dumps a page table of a virtual machine. The **MACHINE** option applies to **PageTable** and **KernelPageTable** and some *<cpu_specific_tables>.* The parameters *<machine_magic>*, *<machine_id>* and *<machine_name>* are displayed in the **TASK.List.MACHINES** window.
Fulltranslation	For page tables of guest machines both the intermediate address and the physical address is displayed in the **MMU.DUMP** window. The physical address is derived from a table walk using the guest's intermediate page table.

ITLB	Displays the contents of the Instruction Translation Lookaside Buffer. For column descriptions, click here.
DTLB	Displays the contents of the Data Translation Lookaside sBuffer. For column descriptions, click here.
TLB0	Displays the contents of the Translation Lookaside Buffer 0. For column descriptions, click here.
TLB1	Displays the contents of the Translation Lookaside Buffer 1. For column descriptions, click here.
NonSecPageTable	Displays the translation table used if the CPU is in non-secure mode and in privilege level PL0 or PL1. This is the table pointed to by MMU registers TTBR0 and TTBR1 in non-secure mode. This option is only visible if the CPU has the TrustZone and/or Virtualization Extension.
SecPageTable	Displays the translation table used if the CPU is in secure mode. This is the table pointed to by MMU registers TTBR0 and TTBR1 in secure mode. This option is only visible if the CPU has the TrustZone Extension.
HypPageTable	Displays the translation table used by the MMU when the CPU is in HYP mode. This is the table pointed to by MMU register HTTBR. This table is only available in CPUs with Virtualization Extension.
IntermedPageTable	Displays the translation table used by the MMU for the second stage translation of a guest machine (intermediate address to physical address). This is the table pointed to by MMU register VTTBR. This table is only available in CPUs with Virtualization Extension.

Description of Columns in the ITLB, DTLB, andTLB<x> Dump Window

[Back]

Logical	Logical address.
Physical	Physical address.
Vmid	Virtual machine ID.
Asid	Address space ID.

Glb	Global flag.
Sec	Non-secure identifier for physical address.
idx	Index of the TLB entry.
pagesize	Page size.
Hyp	Hypervisor entry flag.
V	Valid flag.
L	Locked flag.
I	Inner shareability flag.
O	Outer shareability flag.
M	Indicates if the line was brought in when MMU was enabled.
D	Domain ID.
Attributes	Memory Attributes (check design manual of respective architecture for the format).
Tablewalk	Table walk information.

Examples for Page Tables in Virtualized Systems

Example 1:

```
SYStem.Option.MACHINESPACES ON

; your code to load Hypervisor Awareness and define guest machine setup.

;                              <machine_id>
MMU.DUMP.PageTable /MACHINE     2.


;                              <machine_name>
MMU.DUMP.PageTable /MACHINE     "Dom0"
```

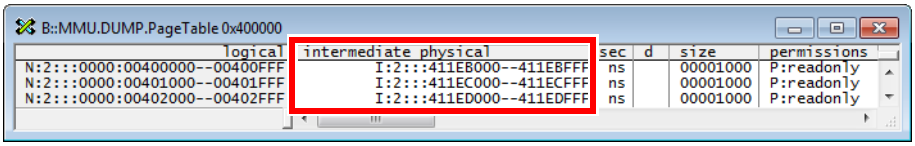
Example 2:

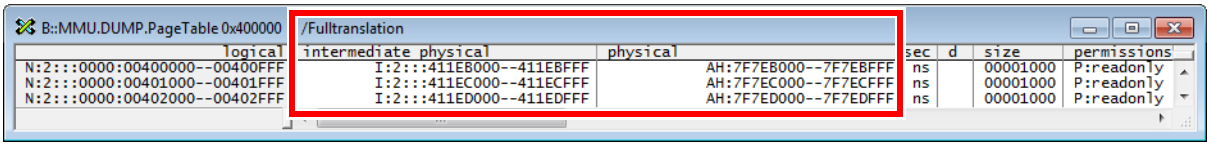
```
SYStem.Option.MACHINESPACES ON

; your code to load Hypervisor Awareness and define guest machine setup.

;                          <machine_name>:::<task_name>
MMU.DUMP.TaskPageTable              "Dom0:::swapper"
```

Example 3:

```
SYStem.Option.MACHINESPACES ON

;your code to load Hypervisor Awareness and define guest machine setup.

;a) dumps the current guest page table of the current machine, showing
;   the intermediate addresses.
;   Without the option /Fulltranslation the column "physical" is hidden.
MMU.DUMP.PageTable 0x400000

;b) With the option /Fulltranslation the intermediate addresses
;   are translated to physical addresses and shown in column "physical"
MMU.DUMP.PageTable 0x400000 /Fulltranslation

;c) dumps the current page table of machine 2
;                              <machine_id>
MMU.DUMP.PageTable  /MACHINE     2.         /Fulltranslation
```

Results for 3 a) and 3 b)





Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] [**/***<option>*]
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** *<cpu_specific_tables>*
<option>:	**MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **Fulltranslation**

Lists the address translation of the CPU-specific MMU table.
In contrast to **MMU.DUMP,** multiple consecutive page table entries with identical page attributes are listed as a single line, showing the total mapped address range.

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

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

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

CPU-specific Tables in MMU.List <table>

NonSecPageTable	Displays the translation table used if the CPU is in non-secure mode and in privilege level PL0 or PL1. This is the table pointed to by MMU registers TTBR0 and TTBR1 in non-secure mode. This option is only visible if the CPU has the TrustZone and/or Virtualization Extension. This option is only enabled if Exception levels EL0 or EL1 use AArch32 mode.
SecPageTable	Displays the translation table used if the CPU is in secure mode. This is the table pointed to by MMU registers TTBR0 and TTBR1 in secure mode. This option is only visible if the CPU has the TrustZone Extension. This option is only enabled if the Exception level EL1 uses AArch32 mode.
HypPageTable	Displays the translation table used by the MMU when the CPU is in HYP mode. This is the table pointed to by MMU register HTTBR. This table is only available in CPUs with Virtualization Extension.
IntermedPageTable	Displays the translation table used by the MMU for the second stage translation of a guest machine (intermediate address to physical address). This is the table pointed to by MMU register VTTBR. This table is only available in CPUs with Virtualization Extension.

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] [*/<option>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** *<cpu_specific_tables>*
<option>:	**MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **Fulltranslation**

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

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

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

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID and/or machine ID: loads the translation table of the specified process and/or machine • else, this command loads the table the CPU currently uses for MMU translation.
KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.

ALL	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**.
<option>	For description of the options, see **MMU.DUMP**.

CPU-specific Tables in MMU.SCAN <table>

OEMAddressTable	Loads the OEM Address Table from the CPU to the debugger-internal translation table.
NonSecPageTable	Displays the translation table used if the CPU is in non-secure mode and in privilege level PL0 or PL1. This is the table pointed to by MMU registers TTBR0 and TTBR1 in non-secure mode. This option is only visible if the CPU has the TrustZone and/or Virtualization Extension. This option is only enabled if Exception levels EL0 or EL1 use AArch32 mode.
SecPageTable	Displays the translation table used if the CPU is in secure mode. This is the table pointed to by MMU registers TTBR0 and TTBR1 in secure mode. This option is only visible if the CPU has the TrustZone Extension. This option is only enabled if the Exception level EL1 uses AArch32 mode.
HypPageTable	Loads the translation table used by the MMU when the CPU is in HYP mode. This is the table pointed to by MMU register HTTBR. This table is only available in CPUs with Virtualization Extension.
IntermedPageTable	Loads the translation table used by the MMU for the second stage translation of a guest machine (intermediate address to physical address). This is the table pointed to by MMU register VTTBR. This table is only available in CPUs with Virtualization Extension.

SMMU — Hardware system MMU (SMMU)

Using the **SMMU** command group, you can analyze the current setup of up to 20 system MMU instances. Selecting a CPU with a built-in SMMU activates the **SMMU** command group.

```
SYStem.CPU CortexA53  ;for example, the 'CortexA53' CPU is SMMU-capable

SMMU.ADD ...      ;you can now define an SMMU, e.g. an SMMU for a
                  ;graphics processing unit (GPU)
```

Some SoC CPU types have already SMMUs predefined as component, visible in the **SYStem.CONFIG** component dialog window.

TRACE32 supports the SMMU types MMU-400, MMU-401 and MMU-500 (based on the Arm SMMU architecture specification v2, short SMMU-v2) and MMU-600 (based on the Arm SMMU architecture specification v3, short SMMU-v3).

The TRACE32 SMMU support visualizes most of the configuration settings of an SMMU. These visualizations include:

- The Stream Table with all Stream Map Register Groups (SMRG, for SMMU-v2) or all Stream Table Entries (STE, for SMMU-v3)
- Access to both the non-secure and the secure SMMU view
- Tabular overview over principal data of each SMRG or STE listed in the Stream Table such as
  - Stream matching register settings (for SMMU-v2)
  - Translation context type (stage 1 / stage 2 enabled / bypass / fault)
  - The context's stream world of a SMRG (HYPC and MONC flags) or STE (EL1/EL2/EL3)
  - Stage 1 / stage 2 context bank indices (for SMMU-v2)
  - The availability of stage1 and stage 2 page tables, their format and the MMU-enable/disableT state for the stage 1 and/or stage 2 address translation
  - VMID and the number of stage 1 Context Descriptors for a STE (for SMMU-v3)
- The stage 1 Context Descriptor Table for a given STE (for SMMU-v3)
- Page table lists or dumps for stage 1 and/or stage 2 address translation contexts
- A quick indication of contexts where a fault has occurred or contexts that are stalled (SMMU-v2)
- A quick indication of the global SMMU fault status
- CMD Queue and Event Queue dumps with filtering options (for SMMU-v3)

- Peripheral register view:
  - Global Configuration Registers of the SMMU
  - The SMRG / STE Registers
  - The Context Bank Registers (SMMU-v2) / Context Descriptor Registers (SMMU-v3)
  - MMU-specific Registers such as Performance Measurement Unit Registers, Translation Control Unit Registers or Translation Buffer Unit Registers (for SMMU-v3)

A good way to familiarize yourself with the **SMMU** command group is to start with:

- The **SMMU.ADD** command
- The **SMMU.StreamTable** command which offers GUI-based access to almost all SMMU data
- The guide **Overview - How To**
- **Glossary - SMMU**
- **Arguments in SMMU Commands**

The **SMMU.StreamTable** command and the window of the same name serve as your SMMU command and control center in TRACE32. The right-click popup menu in the **SMMU.StreamTable** window allows you to execute all frequently-used SMMU commands through the user interface TRACE32 PowerView.

The other SMMU commands are designed primarily for use in PRACTICE scripts (*.cmm) and for users accustomed to working with the command line.

NOTE:	The primary table of streams is called *Stream Map Table* in the SMMU-v2 architecture specification, whereas it is called *Stream Table* in the SMMU-v3 architecture specification. To keep the TRACE32 user interface simple, a single unified command, **SMMU.StreamTable,** is used to access the table of streams for all supported SMMU architecture versions. **SMMU.StreamTable** replaces the deprecated command **SMMU.StreamMapTable** which was used for SMMU-v2 *Stream Map Table* access in older TRACE32 versions. However, **SMMU.StreamMapTable** remains an accepted command in scripts to preserve backward compatibility.

Overview - How To

This chapter is a brief guide which commands can be used to perform common tasks. The guide is split into two parts: one for MMU-400, MMU-401 and MMU-500 which follow the SMMU-v2 specification and one for MMU-600 and newer which follow the SMMU-v3 specification.

MMU-400, MMU-401 and MMU-500:

How To...	GUI action or commands
Define a new SMMU	**SMMU.Add** To get the non-secure/secure SMMU view, specify a non-secure/secure base address.
View the Stream Table with all SMRGs View the stream configurations and see the context bank indices of stage 1 and stage 2	**SMMU.StreamTable**
List or dump stage 1 or stage 2 page tables of a stream	In **SMMU.StreamTable** window: use popup menu or double click on column **stage 1 pagetbl. fmt** or **stage 2 pagetbl. fmt** **SMMU.StreamMapRegGrp.list** **SMMU.StreamMapRegGrp.Dump**
View a stream's SMRG registers	In **SMMU.StreamTable** window: use popup menu or double click on any column of **stream matching** or **context type** **SMMU.StreamMapRegGrp.Register** **SMMU.Register.StreamMapRegGrp**
View stage 1 or stage 2 context bank registers	In **SMMU.StreamTable** window: use popup menu or double click on column **stage 1 cbndx** or **stage 2 cbndx** **SMMU.StreamMapRegGrp.ContextReg** **SMMU.Register.ContextBank**
View global SMMU registers	In **SMMU.StreamTable** window: use popup menu or double click status line **SMMU.Register.Global**
View global SMMU fault flags	Fault flags are displayed in the status line at the bottom of the **SMMU.StreamTable** window. Alternatively, open the global SMMU registers with **SMMU.Register.Global** and view register SMMU_GFSR / SMMU_sGFSR (non-sec/sec)
Check if an SMMU stream is in a fault state	Open the **SMMU.StreamTable** window: Streams in fault/stall/multi state have red **F**/**S**/**M** marks in column **stage 1 state** or **stage 2 state**
View Security State Determination Table (SSD)	In **SMMU.StreamTable** window: use popup menu **SMMU.SSDtable**

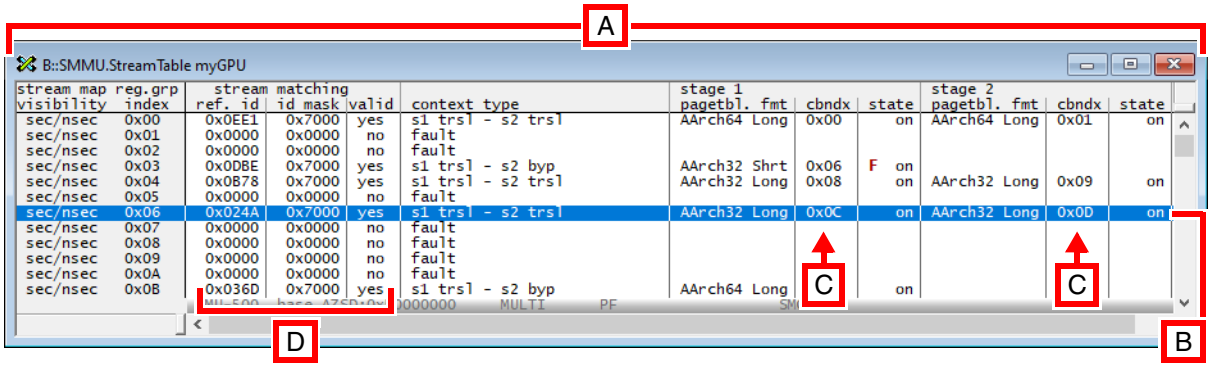
MMU-600 and newer:

How To...	GUI action or commands
Define a new SMMU	**SMMU.Add** Use a secure base address. Default SMMU view is non-secure. Switch to secure view with option **/SECure** in most commands or use check box **Show secure entries** in the header of most SMMU windows.
View the Stream Table with all valid STEs View the stream configuration, VMID, stream world, stage 2 page table format, number of CDs	**SMMU.StreamTable**
View the Context Descriptor Table of a STE with a list of all valid substreams (CDs) View the ASID, stage 1 page table format and TT0/TT1 translation enable state of substreams	In **SMMU.StreamTable** window: use popup menu or click on the STE's **list CDT** button in the **S1 PT fmt** column to open the Context Descriptor Table window. **SMMU.CtxtDescTable**
List or dump stage 2 page tables of a STE	In **SMMU.StreamTable** window: use popup menu or double click on column **S2 PT fmt** or **stage 2 pagetbl. fmt** **SMMU.StreamTblEntry.list** **SMMU.StreamTblEntry.Dump**
List or dump stage 1 page tables of a STE/CD	If STE has only one CD: use popup menu in **SMMU.StreamTable** window or double click on column **S1 PT fmt** to view the CD's page table. If STE has more than one CD: click on the STE's **list CDT** button in the **S1 PT fmt** column to open the Context Descriptor Table window. Here, use popup menu or double click on column **S1 PT fmt**. **SMMU.StreamTblEntry.list** **SMMU.StreamTblEntry.Dump**
View a stream's STE registers	In **SMMU.StreamTable** window: use popup menu or double click on column **configuration** **SMMU.StreamTblEntry.Register** **SMMU.Register.StreamTblEntry**

How To...	GUI action or commands
View the stage 1 CD registers for a substream	If STE has only one CD: use popup menu in **SMMU.StreamTable** window or double click on column **ASID** to view the CD registers. If STE has more than one CD: click on the STE's **list CDT** button in the **S1 PT fmt** column to open the Context Descriptor Table window. Here, use popup menu or double click on column **ASID**. **SMMU.Register.S1Context**
View global SMMU registers	In **SMMU.StreamTable** window: use popup menu or double click status line **SMMU.Register.Global**
View global SMMU fault flags	Fault flags are displayed in the status line at the bottom of the **SMMU.StreamTable** window. Alternatively, open the global SMMU registers with **SMMU.Register.Global** and view register SMMU_GERROR / SMMU_S_GERROR
Check if an SMMU stream or substream is in a fault state Dump Event Queue entries	In the **SMMU.StreamTable** or the **SMMU.CtxtDescTable** window: • either use popup menu **Dump Queue Entries** - **Event Queue** to dump all Event Queue entries • or, with mouse over STE or CD of interest, use popup menu **Dump associated Queue Entries** - **Event Queue** to dump Event Queue entries filtered by Stream ID and Substream ID **SMMU.DumpQueue.Event**
Dump CMD Queue entries	In the **SMMU.StreamTable** or the **SMMU.CtxtDescTable** window: • either use popup menu **Dump Queue Entries** - **CMD Queue** to dump all CMD Queue entries • or, with mouse over STE or CD of interest, use popup menu **Dump associated Queue Entries** - **CMD Queue** to dump CMD Queue entries filtered by Stream ID and Substream ID **SMMU.DumpQueue.CMD**

The following two figures illustrate a few SMMU terms. For explanations of the illustrated SMMU terms and other important SMMU terms not shown here, see below.

MMU-400, MMU-401 and MMU-500:



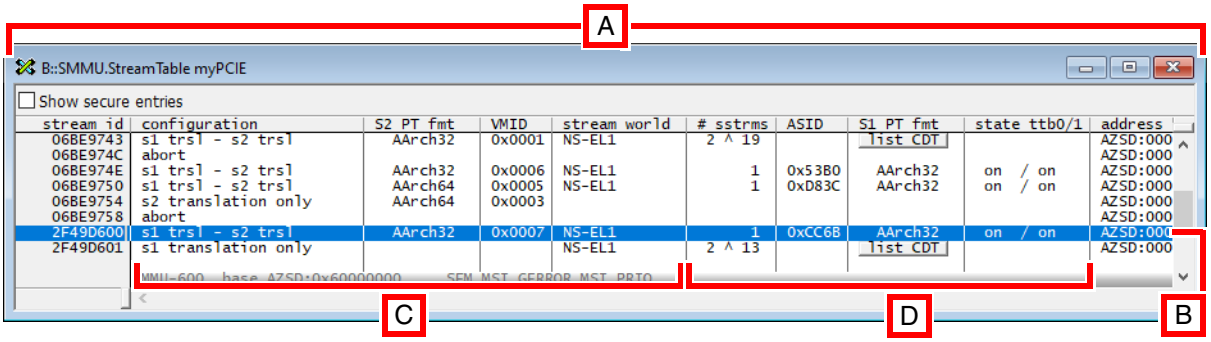
A See stream table.

B Each row stands for a stream map register group (SMRG).

C Index of a translation context bank.

D Data from stream matching registers, see stream matching.

MMU-600 and newer:



A See stream table.

B Each row stands for a stream table entry (STE).

C Stream configuration and stage 2 context.

D Substream data and either stage 1 context or button to view the STE's Context Descriptor Table.

Context Descriptor (CD)

MMU-600 and newer only
A data structure in memory containing register fields which describe a stage 1 translation context, including a pointer to the stage 1 translation table. A CD is identified by its substream ID and by the stream ID of the it belongs to.

Context Descriptor Table (CDT)

MMU-600 and newer only
A table in memory with one or two levels which holds a number of Context Descriptors. Each Context Descriptor Table belongs to one Stream Table Entry.

A CDT can be displayed using command **SMMU.CtxtDescTable**.

Memory Transaction Stream

A stream of memory access transactions sent from a device through the SMMU to the system memory bus. The stream consists of the address to be accessed and a number of design specific memory attributes such as the privilege, cacheability, security attributes or other attributes.

The streams carry a stream ID which the SMMU uses to determine a translation context for the memory transaction stream. As a result, the SMMU may or may not translate the address and/or the memory attributes of the stream before it is forwarded to the system memory bus.

Queue

MMU-600 and newer only
Data structure consisting of a circular buffer in memory which holds queue entries. Queue entries may hold commands for the SMMU (in the CMD Queue) or events generated by the SMMU (in the Event Queue). Queues can be viewed using command **SMMU.DumpQueue**.

Security State Determination Table (SSD Table)

MMU-400, MMU-401 and MMU-500 only
If the SMMU supports two security states (secure and non-secure) an SSD index qualifies memory transactions sent to the SMMU. The SSD index is a hardware signal which is used by the SMMU to decide whether the incoming memory transaction belongs to the secure or the non-secure domain.

The information whether a SSD index belongs to the secure or to the non-secure domain is contained in the SMMU's SSD table.

Stream ID

Peripheral devices connected to an SMMU issue memory transaction streams. Every incoming memory transaction stream carries a Stream Identifier which is used by the SMMU to associate a translation context to the transaction stream. The streams are stored in the Stream Table of the SMMU.

Stream Map Register Group (SMRG)

MMU-400, MMU-401 and MMU-500 only
A group of SMMU registers determining the translation context for a memory transaction stream. The Stream Table holds the SMRGs.

Stream Table (ST) / Stream Mapping Table (SMT)

An SMMU table which describes what to do with an incoming memory transaction stream from a peripheral device. In particular, this table associates an incoming memory transaction stream with a translation context, using the stream ID of the stream as selector of a translation context.

In MMU-400, MMU-401 and MMU-500 (Arm SMMU-v2 specification based), this table of streams is referred to as *Stream Mapping Table*. In MMU-600 and newer (Arm SMMU-v3 specification based), this table of streams is referred to as *Stream Table*. The Stream (Mapping) Table is the central table of the SMMU.

- MMU-400, MMU-401 and MMU-500): each Stream Mapping Table entry consists of a group of registers, called Stream Map Register Group, which describe the translation context. In case an SMMU supports *stream matching*, TRACE32 also displays the *stream matching registers* associated with an entry's stream map register group.
- MMU-600 and newer: the stream table is a data structure in memory and consists of Stream Table Entries which describe the translation context type, the stage 2 translation tables and points to a Context Descriptor Table which holds stage 1 translation contexts.

A Stream Table can be displayed using command **SMMU.StreamTable**.

Stream Matching

MMU-400, MMU-401 and MMU-500 only
In an SMMU which supports stream matching, the stream ID of an incoming memory transaction stream undergoes a matching process to determine which entry of the Stream Table will used to specify the translation context for the stream.
TRACE32 displays the reference ID and the bit mask used by the SMMU to perform the Stream ID matching process in the **SMMU.StreamTable** window.

Stream Table Entry (STE)

MMU-600 and newer only
A data structure in memory describing the translation context for each stream. This data structure register contains fields which describe the type of context, the stage 2 translation context, including a pointer to the stage 2 translation table and a pointer to a Context Descriptor Table holding stage 1 contexts. Each STE is identified by its Stream ID.

Note: for MMU-400, MMU-401 and MMU-500 the entries of the Stream Table are called Stream Map Register Group.

Substream ID

Peripheral devices connected to an SMMU issue memory transaction streams. Every incoming memory transaction stream carries a Stream Identifier which is used by the SMMU to associate a translation context to the transaction stream. The streams are stored in the Stream Table of the SMMU.

Translation Context

A translation context describes the translation process of a incoming memory transaction stream. An incoming memory transaction stream may undergo a stage 1 address translation and/or a stage 2 address translation. Further, the memory attributes of the incoming memory transaction stream may be changed. It is also possible that an incoming memory transaction stream is rendered as fault.

The Stream Table determines which translation context is applied to an incoming memory transaction stream.

Translation Context Bank (short: Context Bank)

MMU-400, MMU-401 and MMU-500 only
A group of SMMU registers specifying the translation context for an incoming memory transaction stream. The registers carry largely the same names and contain the same information as the core's MMU registers describing the address translation process.

The registers of a translation context bank describe the translation table base address, the memory attributes to be used during the translation table walk and translation attribute remapping.

This table provides an overview of frequently-used arguments in SMMU commands. Arguments that are only used in one **SMMU** command are described together with that **SMMU** command.

<name>	User-defined name of an SMMU. Use the **SMMU.ADD** command to define an SMMU and its name. This name will be used to identify an SMMU in all other **SMMU** commands.
<smrg_index>	Index of a stream map register group, e.g. 0x04. The indices are listed in the **index** column of the **SMMU.StreamTable**. The *<smrg_index>* is equivalent to the *<stream_id>* used in MMU-600 and newer. *Only applicable for MMU-400, MMU-401 and MMU-500.*
<cbndx>	Index of a translation context bank. *Only applicable for MMU-400, MMU-401 and MMU-500.*
<stream_id> | *<range>*	Index of a StreamTable Entry or a range of Stream Table Entries. The indices are listed in the **index** column of the **SMMU.StreamTable**. The *<stream_id>* is equivalent to the *<smrg_index>* used in MMU-400, MMU-401 and MMU-500. *Only applicable for MMU-600 and newer.*
<substream_id> | *<range>*	Index of a Context Descriptor Table Entry or a range of Context Descriptor Table Entries. *Only applicable for MMU-600 and newer.*
<address> | *<range>*	Logical address or logical address range describing the start address or the address range to be displayed in the SMMU page table list or dump windows.
IntermediatePT	Used to switch between stage 1 and stage 2 page table or register view: • Omit this option to view the translation table entries or registers of stage 1. • Include this option to view the translation table entries or registers of stage 2.
SECure	Used to switch between the non-secure and the secure SMMU content. • Omit this option to view the non-secure table entries or registers • Include this option to view the secure table entries or registers *Only applicable for MMU-600 and newer.*

Format:	**SMMU.ADD "***<name>***"** *<smmu_type> <base_address>*
<smmu_type>:	**MMU400** | **MMU401** | **MMU500** | **MMU600**

Defines a new SMMU (a hardware system MMU). A maximum of 20 SMMUs can be defined.

NOTE: For some CPUs with SMMUs, TRACE32 will automatically configure the SMMU parameters, so that you can immediately work with the SMMUs and do not need to manually configure them.
After selecting the CPU type, check one of the following locations in TRACE32 to see if there are any pre-configured SMMUs:
- The **CPU** menu > **SMMU** popup menu
- The **SYStem.CONFIG.state /COmponents** window

Arguments:

<name>	User-defined name of an SMMU. The name must be unique and can be max. 9 characters long. **NOTE**: • For the **SMMU.ADD** command, the name must be quoted. • For *all other* **SMMU** commands, omit the quotation marks from the name identifying an SMMU. See also PRACTICE script example below.
<smmu_type>	Defines the type of the Arm system MMU IP block: • SMMUv2 based: **MMU400**, **MMU401** or **MMU500** • SMMUv3 based: **MMU600**

<base_address>	Logical or physical base address of the memory-mapped SMMU register space. **NOTE** for **MMU400, MMU401, MMU500**: If the SMMU supports two security states (secure and non-secure), not all SMMU registers are visible from the non-secure domain. • If you specify a **secure** address as the SMMU base address, you will see the secure view of the SMMU. • If you specify a **non-secure** address as the SMMU base address, you will only see the non-secure SMMU view. Secure SMMU registers will not be visible. To specify a **secure** address, precede the base address with an access class such as AZSD: or ZSD: Always specify either a secure or a non-secure base address so that the SMMU security view is clearly determined. When executing command **SMMU.ADD**, an access class with ambiguous security status will be augmented to either secure or non-secure, according to the current CPU security status and a warning message will be printed. Access classes with a distinct security status will be left unchanged, e.g. the access classes NSD:, NUD:, HD: etc. **NOTE** for **MMU600 and newer**: if CPU supports two security states, always specify the SMMU base address as a secure address (e.g. ZSD: or AZSD:) so that TRACE32 can access both the secure and non-secure SMMU registers.

Example:

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU600 AZSD:0x50000000

;display the stream table of the SMMU named “myGPU”
SMMU.StreamTable myGPU
```

Format:	**SMMU.Clear** *<name>*

Deletes an SMMU definition, which was created with the **SMMU.ADD** command of TRACE32. The **SMMU.Clear** command does not affect your target SMMU.

To delete all SMMU definitions created with the **SMMU.ADD** command of TRACE32, use **SMMU.RESet**.

Argument:

<name>	For a description of *<name>*, click here.

Example:

```
SMMU.Clear myGPU      ;deletes the SMMU named myGPU
```

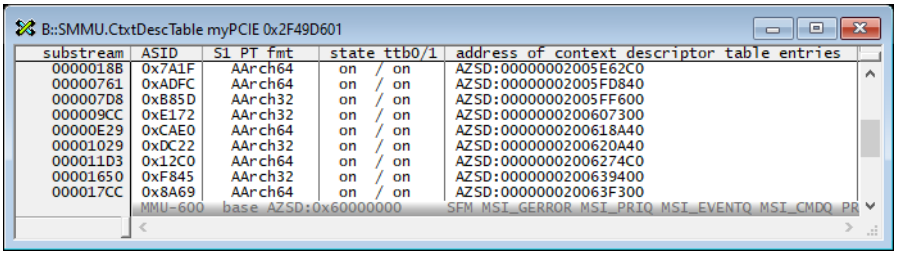
SMMU.CtxtDescTable List a context descriptor table

MMU-600 and newer only

Format:	**SMMU.CtxtDescTable** *<args>*
<args> :	*<name> <stream_id>* [*<substream_id>* | *<range>*] [**/SECure**]

Opens a window and lists all valid stage 1 Context Descriptors in the Context Descriptor Table of the Stream Table Entry specified by *<stream_id>*. Specify option **/SECure** to select the secure SMMU view. A description of the columns is given in this table. The status line of the window shows the global error flags which are currently set for the SMMU.

If you want to limit the Substream IDs displayed in the window, you can specify a numeric *<substream_id>* as lower limit for the displayed SubstreamIDs. Alternatively, you can specify a range as *<substream_id>* to set a lower and an upper limit to the displayed Substream IDs.



Examples:

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU600 AZSD:0x50000000

;list the context descriptors of the stream table with Stream ID 0x6B9743
of the SMMU named "myGPU"
SMMU.CtxtDescTable myGPU 0x6B9743

;same as above, but limit the listing to Substream IDs >= 0x1000
SMMU.CtxtDescTable myGPU 0x6B9743 0x1000

;list the context descriptors of the stream table with secure Stream ID
0x1D73D281 of the SMMU named "myGPU". List only Substream ID in the range
0x1000--0x1FFF
SMMU.CtxtDescTable myGPU 0x1D73D281 0x1000--0x1FFF /SECure
```

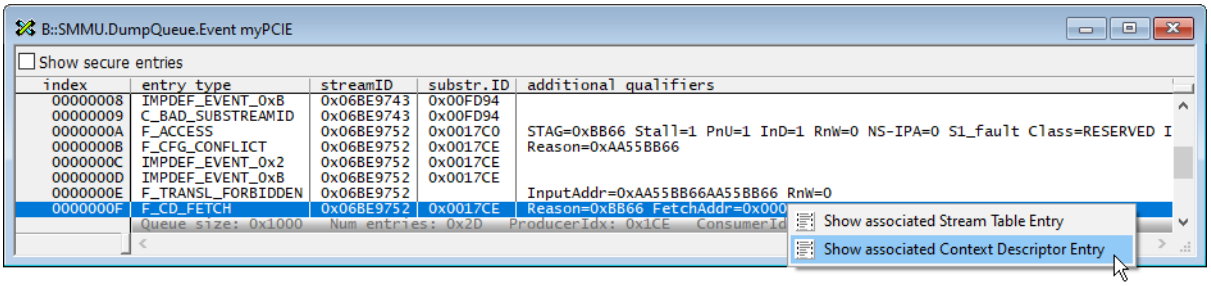
SMMU.DumpQueue.<queue> Dump entries of a queue

MMU-600 and newer only

Using the **SMMU.DumpQueue** command group, you can dump entries of SMMU Queues. Analyzing entries of the Event Queue is important to find error conditions of SMMU streams - in addition to global error flags of the SMMU.

SMMU.DumpQueue.CMD	Dump entries of the Cmd Queue
SMMU.DumpQueue.Event	Dump entries of the Event Queue

The commands **SMMU.DumpQueue.CMD** and **SMMU.DumpQueue.Event** open a window which shows all valid entries of the queue in the sequence of their creation.



The dump queue windows displays the following columns:

Column	Description
index	Index of the entry. Entries are dumped in the sequence of their creation. The oldest entry always carries index 0 in the dump window. This is the entry pointed to by the queue's Consumer Index register. The newest entry has the largest index in the dump window. This is the entry pointed to by the queue's Producer Index register.
entry type	Decoded type of the queue entry.
secure (CMD queue only)	Indicates the state of the SSec bit in the queue entry. If secure is 1, the entry targets the secure SMMU view, otherwise the non-secure view.
streamID	Shows the content of the entry's Stream ID field. Blank if the entry has no Stream ID field.
substr.ID	Shows the content of the entry's Substream ID field.Blank if the entry has no Substream ID field. For the CMD queue, UNKNOWN is displayed if the entry has a Substream ID field but the entry's SSV (SubStream Valid) bit is 0.
additional qualifiers	Depending on the event type, additional event record fields such as addresses and flags are decoded and printed in this column. Note: it is not supported to filter entries by additional qualifier fields.
address of entry	Displays the physical address of the queue table entry record.

The status line of the window shows the following information:

- the number of entries the queue can hold, i.e. its size
- the number of valid entries it holds currently
- the current producer index
- the current consumer index
- if the queue is full, a message "Queue is FULL" is displayed.

NOTE: Use the popup menu to quickly open **SMMU.StreamTable** or **SMMU.CtxtDescrTable** window. This conveniently allows to view the Stream Table Entry or Context Descriptor associated with the queue entry underneath the mouse pointer.

Filter options

As queues can hold a very large number of entries, command **SMMU.DumpQueue.<queue>** offers filter options allowing dump only entries satisfying certain criteria. The following filter options are available:

Filter option	Description
/QETYPE <qe_type>	Dump only queue entries with entry type *<qe_type>* The values allowed for *<qe_type>* are specific to the queue type and the SMMU type.
/StreamID <stream_id> | <range>	Dump only entries with a certain Stream ID. *<stream_id>* can either be a single numeric value or a numeric range. If it is a range, only those queue entries will be dumped if their Stream ID field falls into the specified range.
/SubStreamID <substream_id> | <range>	Dump only entries with a certain Substream ID. *<substream_id>* can either be a single numeric value or a numeric range. If it is a range, only those queue entries will be dumped if their Substream ID field falls into the specified range. In event queue, entries where the SSV (SubStream Valid) bit is 0 are not dumped at all if the **/SubStreamID** filter is active.

Note that for sake of Stream ID and/or Substream ID filtering, TRACE32 evaluates the event record fields StreamID, SubStreamID and SSV regardless of the queue entry type.

SMMU.DumpQueue.CMD Dump cmd queue entries

MMU-600 and newer only

Format:	**SMMU.DumpQueue.CMD** *<name>* [*<entry_idx>* | *<range>*] [**/SECure**] [*<filter_opts>*]
<entry_idx> | *<range>*	Starts the dump with *<entry_index>* or dumps only entries with index in *<range>*
<filter_opts>:	[**/QETYPE** *<qe_type>*] [**/StreamID** *<stream_id>*] [**/SubstreamID** *<substream_id>*]

Opens the **SMMU.DumpQueue** window and dumps all valid entries of the non-secure or the secure Cmd Queue. See **SMMU.DumpQueue** for a description of the dump queue window.

MMU-600 and newer only

Format:	**SMMU.DumpQueue.Event** *<name>* [*<entry_idx>* | *<range>*] [**/SECure**] [*<filter_opts>*]
<entry_idx> | *<range>*	Starts the dump with *<entry_index>* or dumps only entries with index in *<range>*
<filter_opts>:	[**/QETYPE** *<qe_type>*] [**/StreamID** *<stream_id>*] [**/SubstreamID** *<substream_id>*]

Opens the **SMMU.DumpQueue** window and dumps all valid entries of the non-secure or the secure Event Queue. See **SMMU.DumpQueue** for a description of the dump queue window.

Examples:

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU600 AZSD:0x50000000

;open the event queue dump window for the non-secure SMMU view and dump
all entries
SMMU.DumpQueue.Event myGPU

;open the queue dump window for the secure SMMU view and dump all entries
starting with index 0x200
SMMU.DumpQueue.Event myGPU 0x200 /SECure

;dump only entries of type F_TRANSLATION
SMMU.DumpQueue.Event myGPU /QETYPE F_TRANSLATION

;dump only entries where the Stream ID field is in the range 0x5000--
0x5FFF
SMMU.DumpQueue.Event myGPU /StreamID 0x5000--0x5FFF

;dump only entries where the Stream ID field is 0x6BE900 and the
SubStream ID field is in the range 0x140--0x17F
SMMU.DumpQueue.Event myGPU /StreamID 0x6BE900 /SubStreamID 0x140--0x17F
```

Using the **SMMU.Register** command group, you can view and modify the peripheral registers of an SMMU. The command group provides the following commands:

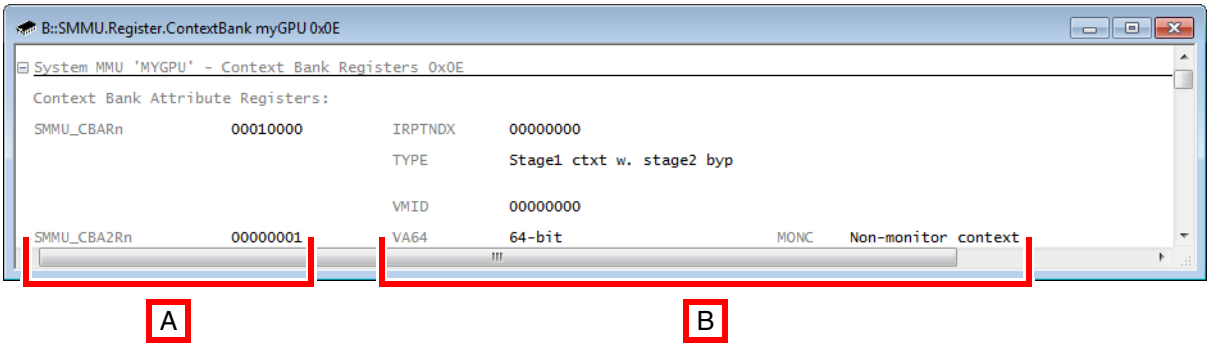
SMMU.Register.Global	Display the global registers of an SMMU
SMMU.Register.ContextBank	Display the registers of a context bank *MMU-400, MMU-401 and MMU-500 only.*
SMMU.Register.StreamMapRegGrp	Display the registers of an SMRG *MMU-400, MMU-401 and MMU-500 only.*
SMMU.Register.StreamTableEntry	Display the registers of a Stream Table Entry. *MMU-600 and newer only.*
SMMU.Register.Stage1Context	Display the registers of a Context Descriptor Table Entry (the stage 1 context of a substream). *MMU-600 and newer only.*

Example:

```
;open the SMMU.Register.StreamMapRegGrp window of SMMU "myGPU" and show
the registers of Stream Table Entry with Stream ID 0x02010A
SMMU.Register.StreamTableEntry   myGPU   0x02010A

;highlight changes in orange in any SMMU.Register.* window
SETUP.Var %SpotLight.on
```

Format:	**SMMU.Register.ContextBank** *<name> <cbndx>*

Opens the peripheral register window **SMMU.Register.ContextBank**. This window displays the registers of the specified context bank. These are listed under the section heading **Context Bank Registers**.



A Register name and content.

B Names of the register bit fields and bit field values.

NOTE: The commands **SMMU.Register.ContextBank** and **SMMU.StreamMapRegGrp.ContextReg** are similar.

The difference between the two commands is:

- The first command expects a *<cbndx>* as an argument and allows to view an arbitrary context bank.
- The second command expects an *<smrg_index>* with an optional **IntermediatePT** as arguments and displays either a stage 1 or stage 2 context bank associated with the *<smrg_index>*.

Argument:

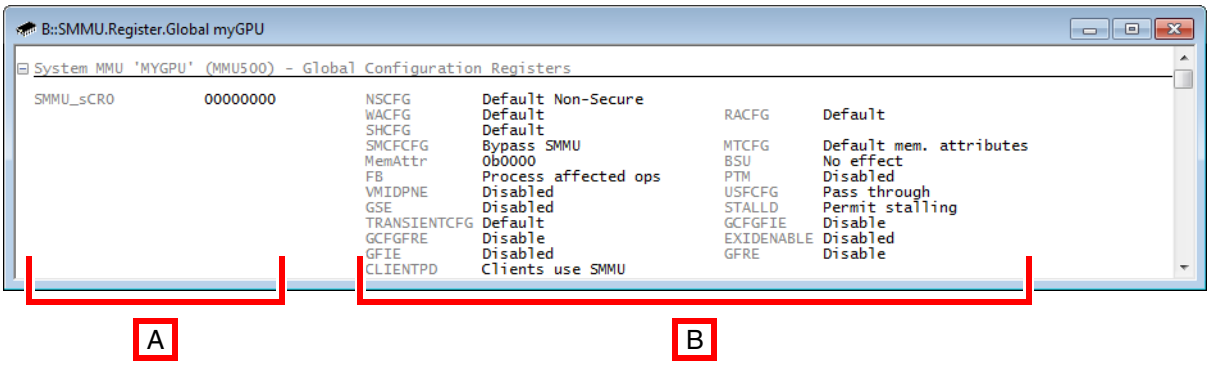
<name>	For a description of *<name>*, etc., click here.

Example:

```
SMMU.Register.ContextBank myGPU 0x16
```

Format: **SMMU.Register.Global** *<name>*

Opens the peripheral register window **SMMU.Register.Global**. This window displays the global registers of the specified SMMU. These are listed under the section heading **Global Configuration Registers**.



A Register name and content.

B Names of the register bit fields and bit field values.

Argument:

<name>	For a description of *<name>*, click here.

Example:

```
SMMU.Register.Global myGPU
```

To display the global registers of an SMMU via the user interface TRACE32 PowerView:

- In the **SMMU.StreamTable** window, right-click an SMRG, and then select **Peripherals** > **Global Configuration Registers** from the popup menu.

SMMU.Register.MMUregs Display MMU specific registers

MMU-600 and newer only

Format: **SMMU.Register.MMUregs** *<name>*

Opens the peripheral register window and shows the MMU specific register blocks which are not part of the SMMU architectural registers. Examples for MMU specific registers are registers for the SMMU Translation Control Unit (TCU), Translation Buffer Unit (TBU) and Performance Measurement Unit (PMU) described in the Arm MMU-600 specification.

SMMU.Register.S1Context — Display stage 1 context descriptor registers

MMU-600 and newer only

Format:	**SMMU.Register.S1Context** *<args>*
<args>:	*<name> <stream_id>* **[/SubstreamID *<substream_id>*]** [**/SECure**]

Opens the peripheral register window for the SMMU named *<name>* and displays the registers of a stage 1 Context Descriptor specified by *<stream_id>* and *<substream_id>*.

If the Stream Table Entry specified by *<stream_id>* has only one Context Descriptor, you can omit option **/SubstreamID *<substream_id>***. In this case, the Context Descriptor with Substream ID 0 will be displayed.

Specify option /**SECure** to select the secure SMMU view.

SMMU.Register.StreamTblEntry — Display stream table entry registers

MMU-600 and newer only

Format:	**SMMU.Register.StreamTblEntry** *<args>*
<args> :	*<name> <stream_id>* [**/SECure**]

Opens the peripheral register window for the SMMU named *<name>* and displays the registers of the Stream Table Entry which is specified by *<stream_id>*.

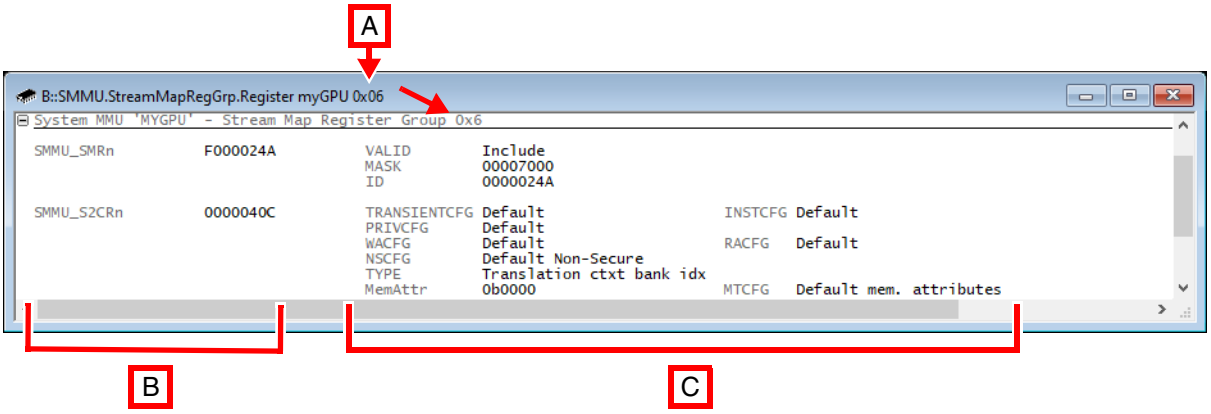
Specify option /**SECure** to select the secure SMMU view.

Example:

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU600 AZSD:0x50000000

;list the Stream Table Entry with Stream ID 0x6B9743 from the secure
Stream Table of SMMU "myGPU"
SMMU.StreamTable myGPU 0x6B9743 /SECure
```

Format:	**SMMU.Register.StreamMapRegGrp** *<args>* **SMMU.StreamMapRegGrp.Register** *<args>* (as an alias)
<args>:	*<name> <smrg_index>*

Opens the peripheral register window **SMMU.Register.StreamMapRegGrp**. This window displays the registers of the specified SMRG. These are listed under the gray section heading **Stream Map Register Group**.



A 0x0D is the *<smrg_index>* of the selected SMRG.

B Register name and content.

C Names of the register bit fields and bit field values.

Compare also to **SMMU.StreamMapRegGrp.ContextReg**.

Arguments:

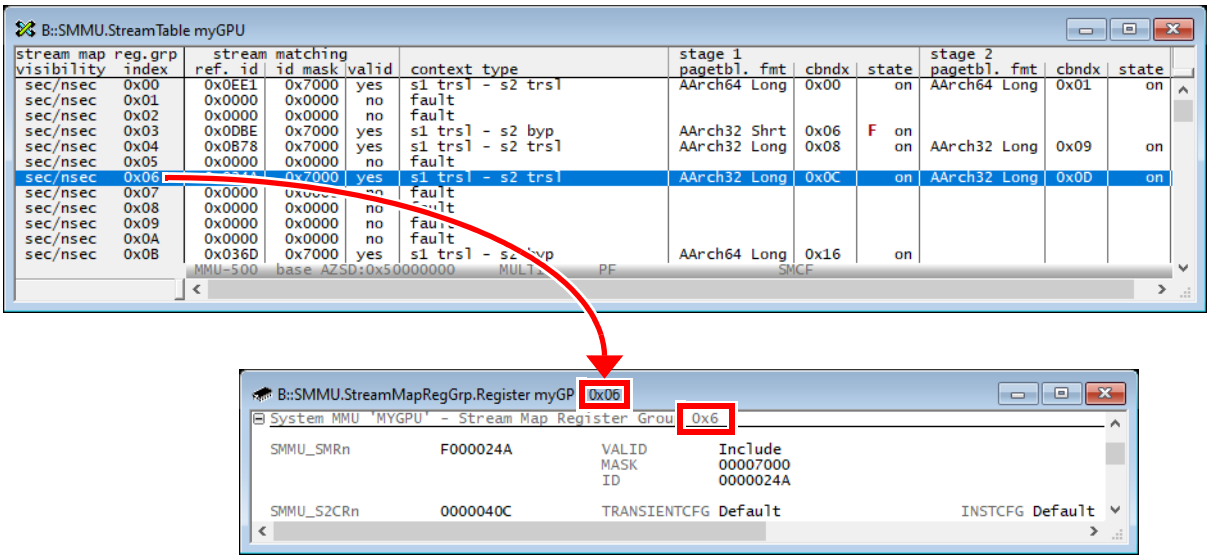
<name>	For a description of *<name>*, etc., click here.

Example:

```
SMMU.StreamMapRegGrp.Register    myGPU    0x06
```

To view the registers of an SMRG via the user interface TRACE32 PowerView:

- In the **SMMU.StreamTable** window, right-click an SMRG, and then select **Peripherals** > **Stream Mapping Registers** from the popup menu.



SMMU.RESet

Delete all SMMU definitions

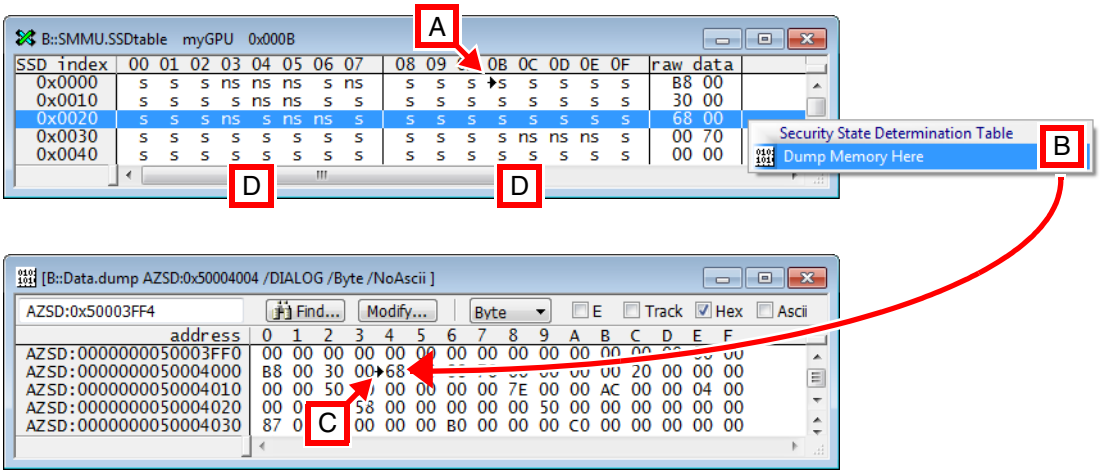
Format: **SMMU.RESet**

Deletes all SMMU definitions created with **SMMU.ADD** from TRACE32. The **SMMU.RESet** command does not affect your target SMMU.

To delete an individual SMMU created with **SMMU.ADD**, use **SMMU.Clear**.

Format: **SMMU.SSDtable** *<name>* [*<start_index>*]

Displays the security state determination table (SSD table) as a bit field consisting of **s** (secure) or **ns** (non-secure) entries. If the SMMU has no SSD table defined, you receive an error message in the **AREA** window.

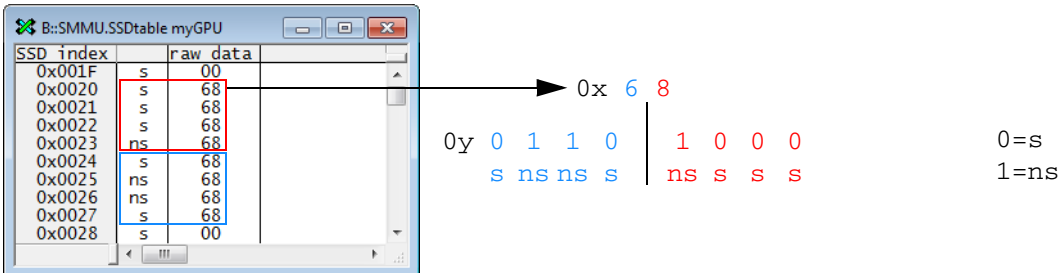


A In the SSD table, the black arrow indicates the *<start_index>*, here 0x00B

B Right-click to dump the SSD table raw data in memory.

For each SSD index of an incoming memory transaction stream, the SSD table indicates whether the outgoing memory transaction stream accesses the secure (**s**) or non-secure (**ns**) memory domain.

You may find the SSD table easier to interpret by reducing the width of the **SMMU.SSDtable** window. Example for the raw data `0x68` in the SSD table:



C In the **Data.dump** window, the black arrow indicates the dumped raw data from the SSD table.

D The 1st white column (00 to 07) relates to the 1st **raw data** column.
The 2nd white column (08 to 0F) relates to the 2nd **raw data** column, etc.

Arguments:

<name>	For a description of *<name>*, click here.
<start_index>	Starts the display of the SSD table at the specified SSD index. See **SSD index** column in the **SMMU.SSDtable** window.

Example:

```
;display the SSD table starting at the SSD index 0x000B
SMMU.SSDtable    myGPU    0x000B
```

To view the SSD table via the user interface TRACE32 PowerView:

- In the **SMMU.StreamTable** window, right-click any SMRG, and then select **Security State Determination Table (SSD)** from the popup menu.

NOTE:	The menu item is grayed out if the SMMU does not support the two security states **s** (secure) or **ns** (non-secure).

SMMU.StreamMapRegGrp — Access to stream map table entries

MMU-400, MMU-401 and MMU-500 only

The **SMMU.StreamMapRegGrp** command group allows to view the details of the translation context associated with stage 1 and/or stage 2 of an SMRG. Every SMRG is identified by its *<smrg_index>*.

The **SMMU.StreamMapRegGrp** command group provides the following commands:

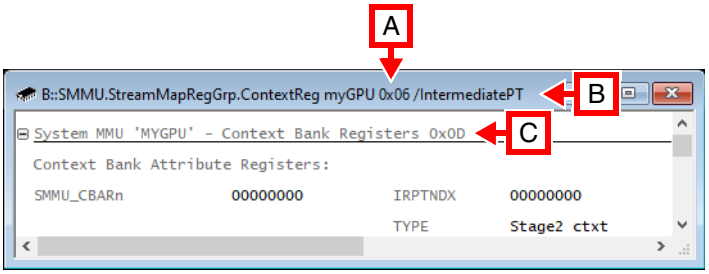
SMMU.StreamMapRegGrp.ContextReg	Shows the registers of the context bank associated with the stage 1 and/or stage 2 translation.
SMMU.StreamMapRegGrp.Dump	Dumps the page table associated with the stage 1 and/or stage 2 translation page wise.
SMMU.StreamMapRegGrp.list	Lists the page table entries associated with the stage 1 and/or stage 2 translation in a compact format.

MMU-400, MMU-401 and MMU-500 only

Format:	**SMMU.StreamMapRegGrp.ContextReg** *<args>*
<args>:	*<name> <smrg_index>* [**/IntermediatePT**]

Opens the peripheral register window **SMMU.StreamMapRegGrp.ContextReg**, displaying the context bank registers of stage 1 or stage 2 of the specified *<smrg_index>* [**A**]. The context bank index (cbndx) of the shown context bank registers is printed in the gray section heading **Context Bank Registers** [**C**].

The **cbndx** columns in the **SMMU.StreamTable** window tell you which context bank is associated with stage 1 or stage 2: If there is no context bank defined for stage 1 or stage 2, then the respective **cbndx** cell is empty. In this case, the peripheral register window **SMMU.StreamMapRegGrp.ContextReg** does not open.



A 0x0A is the *<smrg_index>* of the selected SMRG.

B The option **IntermediatePT** is used to display the context bank registers of stage 2.

C 0x15 is the index from the **cbndx** column of a stage 2 context bank. See example below.

Compare also to **SMMU.StreamMapRegGrp.Register**.

NOTE: The commands **SMMU.Register.ContextBank** and **SMMU.StreamMapRegGrp.ContextReg** are similar.

The difference between the two commands is:

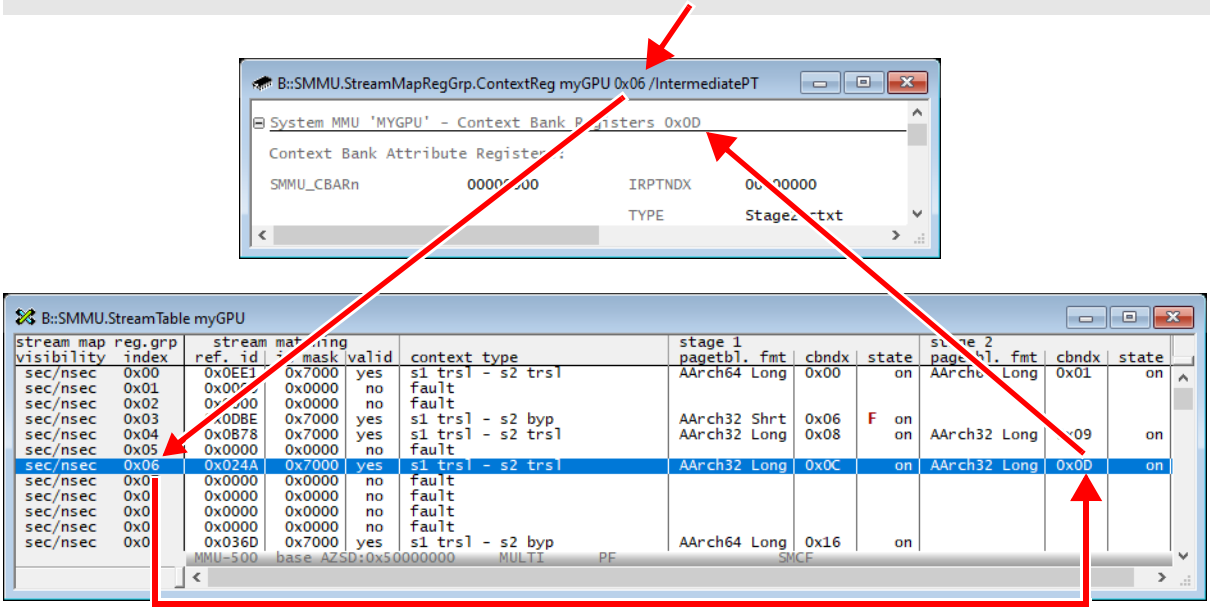
- The first command expects a *<cbndx>* as an argument and allows to view an arbitrary context bank.
- The second command expects an *<smrg_index>* with an optional **IntermediatePT** as arguments and displays either a stage 1 or stage 2 context bank associated with the *<smrg_index>*.

Arguments:

<name>	For a description of *<name>*, etc., click here.

PRACTICE Script Example and Illustration of the Context Bank Look-up:

```
SMMU.StreamMapRegGrp.ContextReg myGPU 0x06 /IntermediatePT
```



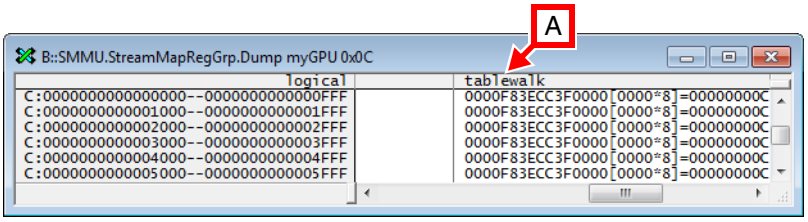
To display the context bank registers via the user interface TRACE32 PowerView:

- In the **SMMU.StreamTable** window, right-click an SMRG, and then select **Peripherals** > **Context Bank Registers of Stage 1** or **2** from the popup menu.

MMU-400, MMU-401 and MMU-500 only

Format:	**SMMU.StreamMapRegGrp.Dump** *<args>*
<args>:	*<name> <smrg_index>* [*<address>* | *<range>* [*<ttb_address>*]] [*/<option>*]

Opens the **SMMU.StreamMapRegGrp.Dump** window for the specified SMRG, displaying the page table entries of the SMRG page wise. If no valid translation context is defined, the window displays the error message “registerset undefined”.



A To view the details of the page table walk, scroll to the right-most column of the window.
For a description of the columns in the **SMMU.StreamMapRegGrp.Dump** window, click here.

Arguments:

<name>	For a description of *<name>*, etc., click here.
<address> | *<range>*	If specified, start the dump with *<address>* or, alternatively, limit the dumped address range to address to *<range>.*
<ttb_address>	If specified, *<ttb_address>* will be used as page table base address. The other page table parameters are still extracted from the SMRG context.
IntermediatePT	Omit this option to view translation table entries of stage 1. Include this option to view translation table entries of stage 2. In SMMUs that support only stage 2 page tables, this option can be omitted.

Example:

```
SMMU.StreamMapRegGrp.Dump myGPU 0x0C
```

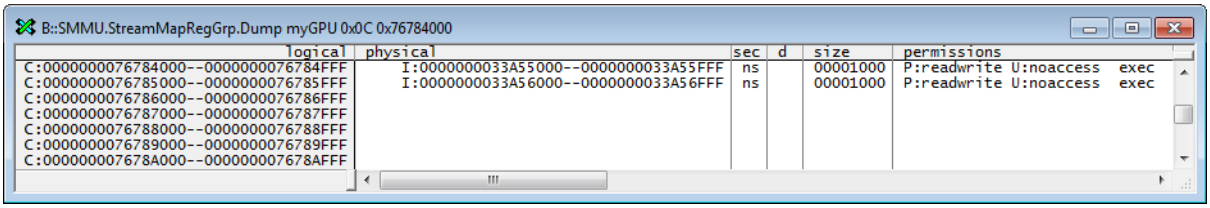
To display an SMMU page table page-wise via the user interface TRACE32 PowerView:

- In the **SMMU.StreamTable** window, right-click an SMRG, and then select from the popup menu:
  - **Stage 1 Page Table** > **Dump** or
  - **Stage 2 Page Table** > **Dump**

Description of Columns

This table describes the columns of the following windows:

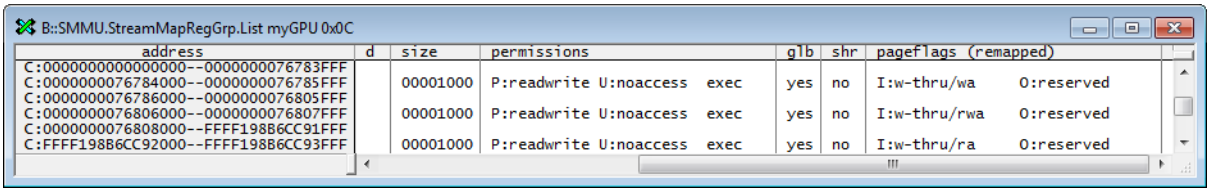
- **SMMU.StreamMapRegGrp.list** / **SMMU.StreamTblEntry.list**
- **SMMU.StreamMapRegGrp.Dump** / **SMMU.StreamTblEntry.Dump**



Column	Description
logical	Logical page address range
physical	Physical page address range
sec	Security state of entry (s=secure, ns=non-secure, sns=non-secure entry in secure page table)
d	Domain
size	Size of mapped page in bytes
permissions	Access permissions (P=privileged, U=unprivileged, exec=execution allowed)
glb	Global page
shr	Shareability (no=non-shareable, yes=shareable, inn=inner shareable, out=outer shareable)
pageflags	Memory attributes (see **Description of the memory attributes**.)
tablewalk	Only for **SMMU.StreamMapRegGrp.Dump**: • Details of table walk for logical page address (one sub column for each table level, showing the table base address, entry index, entry width in bytes and value of table entry)

Format:	**SMMU.StreamMapRegGrp.list** *<args>*
<args>:	*<name> <smrg_index>* [*<address>* | *<range>* [*<ttb_address>*]] [**/IntermediatePT**]

Opens the **SMMU.StreamMapRegGrp.list** window for the specified SMMU, listing the page table entries of a stream map group. If no valid translation context is defined, the window displays an error message.



For a description of the columns in the **SMMU.StreamMapRegGrp.list** window, click here.

Arguments:

<name>	For a description of *<name>*, etc., click here.
<address> / *<range>*	If specified, start the page table list with *<address>* or, alternatively, limit the listed address range to address to *<range>*.
<ttb_address>	If specified, *<ttb_address>* will be used as page table base address. The other page table parameters are still extracted from the SMRG context.
IntermediatePT	Omit this option to view translation table entries of stage 1. Include this option to view translation table entries of stage 2. In SMMUs that support only stage 2 page tables, this option can be omitted.

Example:

```
SMMU.StreamMapRegGrp.list myGPU 0x0C
```

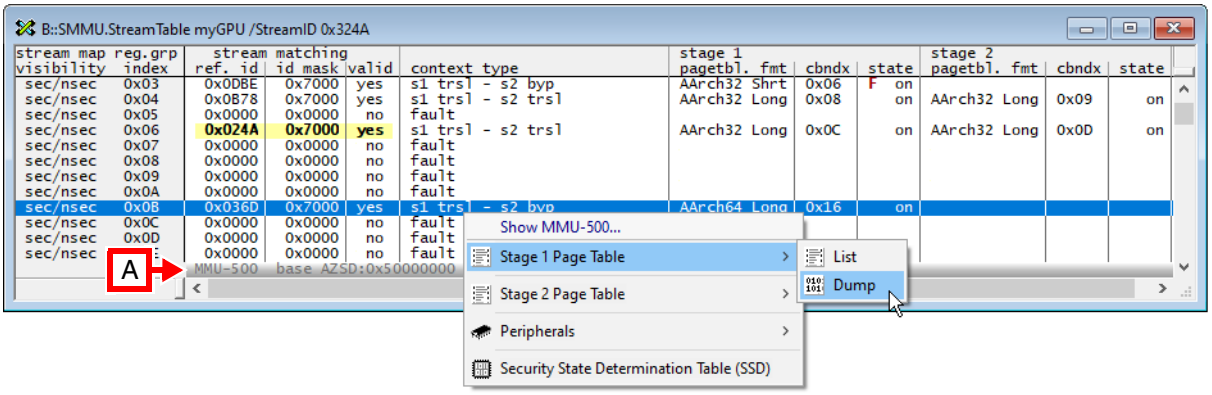
To list the page table entries via the user interface TRACE32 PowerView:

- In the **SMMU.StreamTable** window, right-click an SMRG, and then select from the popup menu:
  - **Stage 1 Page Table** > **List** or
  - **Stage 2 Page Table** > **List**

Format:	**SMMU.StreamTable** *<args>*
	SMMU.StreamMapTable *<args>* **(as an alias)**
<args>:	*<name>* [**/StreamID** *<value>*] (for MMU-400, MMU-401 and MMU-500)
	<name> [*<stream_id>*] [**/SECure**] (for MMU-600 and newer)

Opens the **SMMU.StreamTable** window for the SMMU that has the specified *<name>*. The content and popup menu depends on the SMMU type for which the **SMMU.StreamTable** window is opened. The two variants of the window are described as follows:

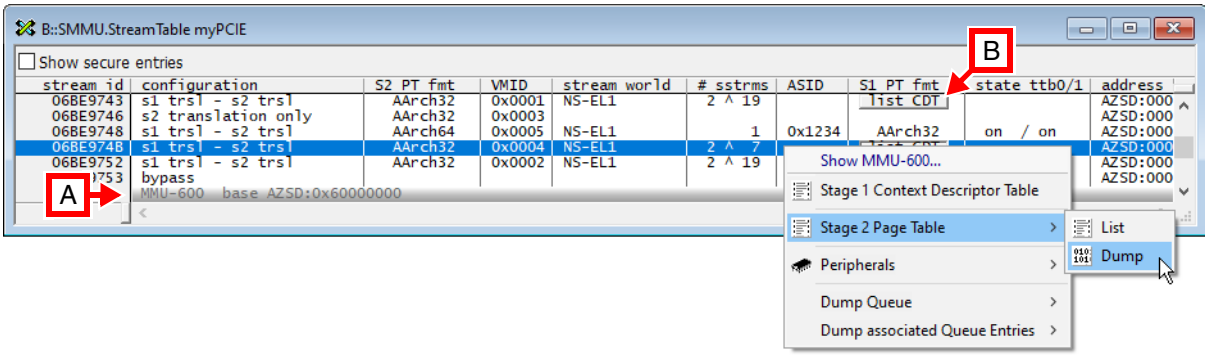
MMU-400, MMU-401, MMU-500:
The window lists all Stream Map Register Groups of the secure or non-secure view of the SMMU. The window provides an overview of the secure or non-secure SMMU configuration.



A The gray window status bar displays the *<smmu_type>* and the SMMU *<base_address>*.
In addition, the window status bar informs you of global faults in the SMMU, if there are any faults.

MMU-600 and newer:
The window lists all valid Stream Table Entries of either the secure or the non-secure view of the SMMU. The security status of the view can be changed using option **/SECure** or, alternatively, using the **Show secure entries** checkbox in the window header.

The Stream ID range displayed can be limited if argument *<stream_id>* is used. You can either specify a number as start value or a range.



A The gray window status bar displays the *<smmu_type>* and the SMMU *<base_address>*.
In addition, the window status bar informs you of global faults in the SMMU, if there are any faults.

B For STEs with more than one substream, click the button **list CDT** to view the substreams.

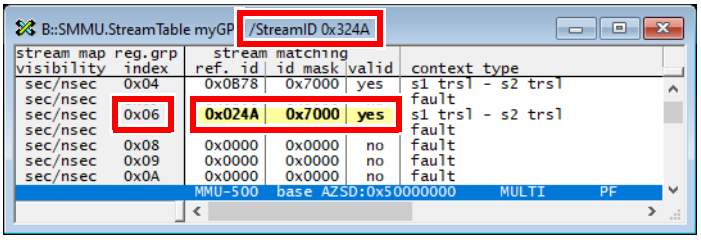
Arguments

<name>	For a description of *<name>*, click here.
StreamID *<value>* (MMU-400, MMU-401 and MMU-500 only)	Only available for SMMUs that support stream ID matching. The **StreamID** option highlights all SMRGs in yellow that match the specified stream ID *<value>*. SMRGs highlighted in yellow help you identify incorrect settings of the stream matching registers. For *<value>*, specify the stream ID of an incoming memory transaction stream. • The highlighted SMRG indicates which stream map table entry will be used to translate the incoming memory transaction stream. • More than one highlighted row indicates a potential, global SMMU fault called stream match conflict fault. The stream ID matching algorithm of TRACE32 mimics the SMMU stream matching on the real hardware. The reference ID, mask and validity fields of the stream match register are listed in the **ref. id**, **id mask** and **valid** columns.
<stream_id> (MMU-600 and newer only)	Either the start point (if a single number is given) or numeric range (if a numeric range is given) of Stream IDs that are displayed in the window.

[Back to Top]

MMU-400, MMU-401, MMU-500:
This PRACTICE script example shows how to define an SMMU with the **SMMU.ADD** command. Then the script opens the SMMU in the **SMMU.StreamTable** window, searches for the *<stream_id>* 0x**3**24A and highlights the matching SMRG 0x**0**24A in yellow.

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU500 AZSD:0x50000000

;open the window and highlight the matching SMRG in yellow
SMMU.StreamTable myGPU /StreamID 0x324A
```



NOTE:	At first glance, the **Stream ID** 0x**3**24A does not seem to match the SMRG 0x**0**24A. However, if you take the ID mask 0x7000 (= 0y0111_0000_0000_0000) into account, the match is correct.

The row highlighted in yellow in the **SMMU.StreamTable** window is a correct match for the **Stream ID** 0x324A we searched for.

See also function **SMMU.StreamID2SMRG()** in **"General Function Reference"** (general_func.pdf).

MMU-600 and newer:
This PRACTICE script example shows how to define an SMMU with the **SMMU.ADD** command. Then the script opens the SMMU in the **SMMU.StreamTable** window starting with Stream ID 0x10000

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU600 AZSD:0x50000000

;open the Stream Table window, showing entries starting with
Stream ID 0x10000
SMMU.StreamTable myGPU 0x10000
```

[Back to Top]

By right-clicking an entry or double-clicking certain cells of an entry, you can open additional windows to receive more information about the selected entry.

- Right-clicking opens the **Popup Menu**.

MMU-400, MMU-401, MMU-500:

- Double-clicking an entry in the columns **ref. id**, **id mask**, **valid**, or **context type** opens the **SMMU.StreamMapRegGrp.Register** window.
- Double-clicking an SMRG in the two columns **pagetbl. fmt** opens the **SMMU.StreamMapRegGrp.list** window, displaying the page table for stage 1 or stage 2.
- Double-clicking an SMRG in the two **cbndx** columns or the two **state** columns opens the **SMMU.StreamMapRegGrp.ContextReg** window, displaying the context bank registers for stage 1 or stage 2.

MMU-600 and newer:

- Double-clicking an entry in the columns **configuration**, **VMID**, **stream world**, or **# sstrms** opens the **SMMU.StreamTblEntry.Register** window showing the stream entry registers.
- Double-clicking an entry in the column **S2 PT fmt** opens the **SMMU.StreamTblEntry.list** window, displaying the stage 2 page table.

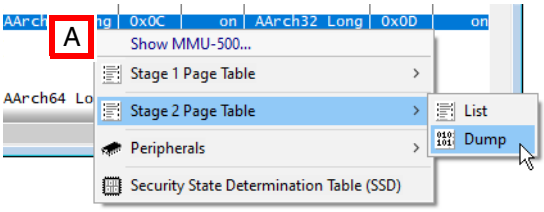
If an entry has only one stage 1 context descriptor:

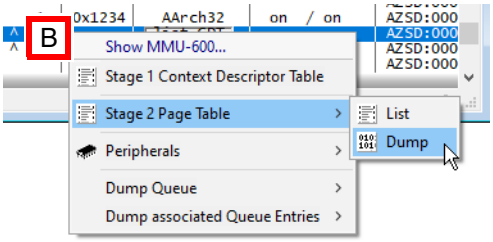
- Double-clicking valid data in columns **ASID** or **state ttb0/1** opens the **SMMU.Register.S1Context** window, displaying the stage 1 context registers.
- Double-clicking valid data in column **S1 PT fmt** opens the **SMMU.StreamTblEntry.list** window, displaying the stage 1 page table.

If an entry has more than one stage 1 context descriptor:

- Click on the list CDT button in column **S1 PT fmt** to open the **SMMU.CtxtDescTable** window, listing all valid Context Descriptors for the stream entry. The **SMMU.CtxtDescTable** window allows to view the registers and stage 1 page tables associated with each Context Descriptor.

[Back to Top]





A Example popup menu for MMU-400, MMU-401 and MMU-500

B Example popup menu for MMU-600 and newer

The entries visible in the popup menus depend on the capabilities of the SMMU such as the capability to support stage 1 or stage 2 and if the SMMU supports two security states.

The popup menu in the **SMMU.StreamTable** window provides convenient shortcuts to the following commands:

MMU-400, MMU-401 and MMU-500:

Popup Menu	Command
Stage 1 Page Table > **Stage 2 Page Table >**	(--)
• **List** • **Dump**	• **SMMU.StreamMapRegGrp.list** • **SMMU.StreamMapRegGrp.Dump**
Peripherals >	(--)
• **Global Configuration Registers** • **Stream Mapping Registers** • **Context Bank Registers of Stage 1** and **Context Bank Registers of Stage 2**	• **SMMU.Register.Global** • **SMMU.Register.StreamMapRegGrp** • **SMMU.Register.ContextBank**
Security State Determination Table (SSD)	**SMMU.SSDtable**

MMU-600 and newer:

Popup Menu	Command
Stage 1 Context Descriptor Table	**SMMU.CtxtDescTable**
Stage 1 Page Table > **Stage 2 Page Table >**	(--)
• **List** • **Dump**	• **SMMU.StreamTblEntry.list** • **SMMU.StreamTblEntry.Dump**
Peripherals >	(--)
• **Global Configuration Registers** • **MMU specific Registers** • **Stream Table Entry Registers** • **Stage 1 Context Descriptor Registers**	• **SMMU.Register.Global** • **SMMU.Register.MMU** • **SMMU.Register.StreamTblEntry** • **SMMU.Register.S1Context**
Dump Queue > **Dump associated Queue Entries >**	(--)
• **Event Queue** • **Cmd Queue**	• **SMMU.DumpQueue.Event** • **SMMU.DumpQueue.CMD**

[Back to Top]

MMU-400, MMU-401 and MMU-500:

Column Name	Description
stream map reg. grp	• **visibility**: The column is only visible if the SMMU supports the two security states *secure* and *non-secure*. The label **sec/nsec** indicates that the SMRG is visible to secure and non-secure accesses. The label **sec only** indicates that the SMRG is visible to secure accesses only. • **index**: The index numbers start at **0x00** and are incremented by 1 per SMRG.
stream matching	See description of the columns **ref. id**, **id mask**, and **valid** below.
ref. id, **id mask**, and **valid**	If the SMMU supports *stream matching*, then the following columns are visible: **ref. id**, **id mask**, and **valid**. Otherwise, these columns are hidden.
context type	Depending on the translation context of a stream mapping register group, the following values are displayed [Description of Values]: • s2 translation only • s1 trsl - s2 trsl • s1 trsl - s2 fault • s1 trsl - s2 byp • fault (s1 trsl-s2 trsl) • fault (s1 trsl-s2 flt) • fault (s1 trsl-s2 byp) • fault • bypass mode • reserved • HYPC or MONC
stage 1 pagetbl. fmt or **stage 2 pagetbl. fmt**	Displays the page table format of **stage 1** or **stage 2** [Description of Values]: • Short descr. (32-bit Arm architecture only) • Long descr. (32-bit Arm architecture only) • AArch32 Short (64-bit Arm architecture only) • AArch32 Long (64-bit Arm architecture only) • AArch64 Long (64-bit Arm architecture only)
cbndx	Displays the context bank index (cbndx) associated with the translation context of **stage 1** or **stage 2**.

Column Name	Description
state	Displays whether the MMU of **stage 1** or **stage 2** is enabled (ON) or disabled (OFF) and whether a fault has occurred in a translation context bank: • **F**: any single fault • **M**: multiple faults • **S**: the SMMU is stalled The letters F, M, and S are highlighted in red in the **SMMU.StreamTable** window (example). The information about the faults is derived from the register SMMU_CBn_FSR (fault status register of the context bank). Double-click the respective **state** cell to open the **SMMU.StreamMapRegGrp.ContextReg** window. The register SMMU_CBn_FSR provides details about the fault.

MMU-600 and newer:

Column Name	Description
configuration	Depending on the translation context of a stream entry, the following values are displayed [Description of Values]: • s1 translation only • s2 translation only • s1 trsl - s2 trsl • bypass • abort A misconfiguration of the stream entry is indicated by a display of ILLEGAL.
S2 PT fmt or **S1 PT fmt**	Displays the page table format of **stage 2** or **stage 1**: • AArch32 • AArch64
VMID	Displays the VMID of the stream table entry stage 2 registers
stream world	Depending of the stream world of a stream entry, the following values are displayed: • NS-EL1 • EL2 • EL2-E2H • EL3 • Secure • Reserved
# sstrms	Displays the max. number of stage 1 context descriptors for the stream table entry, as configured in the S1CDMAX field
ASID	Displays the ASID of a stage 1 context descriptor

Column Name	Description
S1 PT fmt	If only a single context descriptor entry exists in the CDT associated with the stream table entry, it's stage 1page table format is displayed (AArch32 or AArch64). If the CDT contains more than one entry, a button labelled *list CDT* is displayed which directly opens the CDT.
state ttb0/1	Displays the state of the stage 1 context **tt0** / **tt1** translation table disable bits, where **tt0** refers to the address translation of the lower address range. **tt1** refers to the address translation of the upper address range. Possible values for: **tt0** / **tt1** • **on** means the translation for the tt0 / tt1 address range is enabled • **off** means the translation for the tt0 / tt1 address range is disabled
address of stream table entries ***or*** **address of context desciptor table entries**	Displays table walk details, i.e. the physical addresses of the level 1 and/or level 2 table entries. If the table has only one level, one address is displayed, for a 2-level table two addresses are displayed.

[Back to Top]

MMU-400, MMU-401 and MMU-500:

Values in the Column "context type"	Description
s2 translation only	Context defines a stage 2 translation only
s1 trsl - s2 trsl	Context defines a stage 1 translation, followed by a stage 2 translation (nested translation)
s1 trsl - s2 fault	Context defines a stage 1 translation followed by a stage 2 fault
s1 trsl - s2 byp	Context defines a stage 1 translation followed by a stage 2 bypass
fault (s1 trsl-s2 trsl)	Context defines a stage 1 translation followed by a stage 2 translation, but SMMU has no stage 1 (SMMU configuration fault)
fault (s1 trsl-s2 flt)	Context defines a stage 1 translation followed by a stage 2 fault, but SMMU has no stage 1 (SMMU configuration fault)
fault (s1 trsl-s2 byp)	Context defines a stage 1 translation followed by a stage 2 bypassn, but SMMU has no stage 1 (SMMU configuration fault)
fault	Context defines a fault
bypass mode	Context defines bypass mode
reserved	Context type is improperly defined
HYPC	Is displayed on the right-hand side of the column if the context is a hypervisor context.
MONC	Is displayed on the right-hand side of the column if the context is a monitor context.

Values in the Columns "stage 1 pagetbl. fmt" "stage 2 pagetbl. fmt"	Description
Short descr.	Page table uses the 32-bit short descriptor format (32-bit targets only)
Long descr.	Page table uses the 32-bit long descriptor (LPAE) format (32-bit targets only)
AArch32 Short	Page table uses the 32-bit short descriptor format (64-bit targets only)
AArch32 Long	Page table uses the 32-bit long descriptor (LPAE) format (64-bit targets only)
AArch64 Long	Page table uses the 64-bit long descriptor (LPAE) format (64-bit targets only)

MMU-600 and newer:

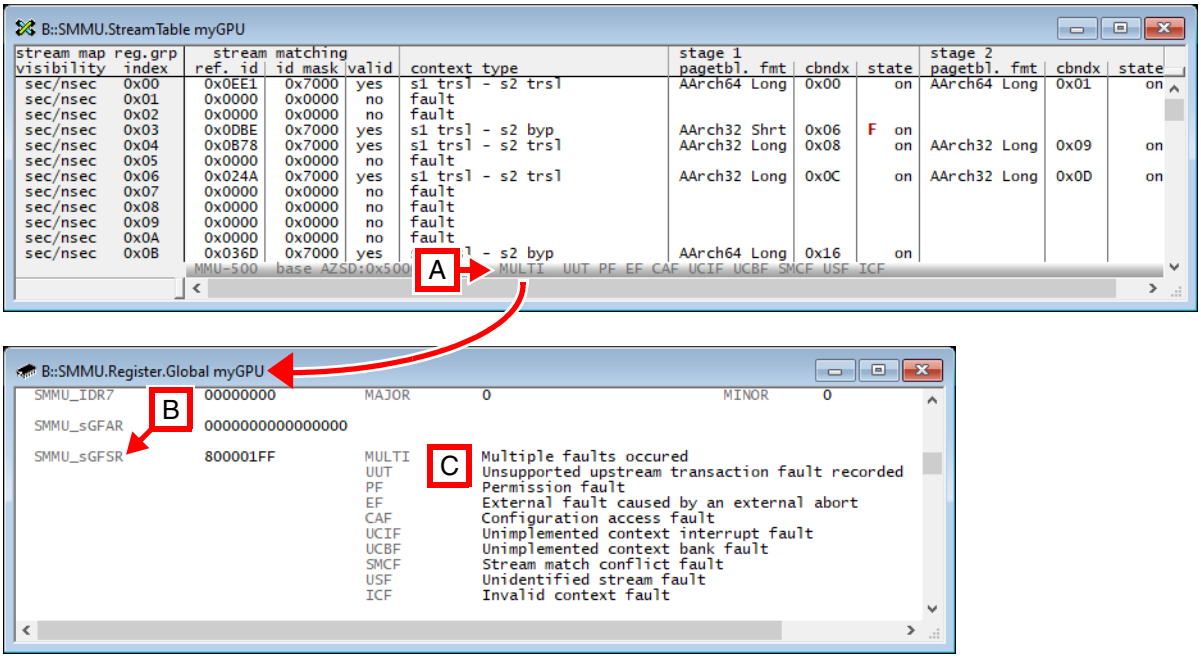
Values in the Column "configuration"	Description
s1 translation only	Context defines a stage 1 translation only
s2 translation only	Context defines a stage 2 translation only
s1 trsl - s2 trsl	Context defines a stage 1 translation, followed by a stage 2 translation
bypass	Context defines bypass mode, no translation is performed.
abort	Context defines an abort condition.
ILLEGAL (s1 trsl only)	Misconfiguration of the stream table entry: stage 1 translation is configured but not supported
ILLEGAL (s2 trsl only)	Misconfiguration of the stream table entry: stage 2 translation is configured but not supported
ILLEGAL (s1 + s2 trsl)	Misconfiguration of the stream table entry: stage 1+2 translations are configured but not supported
ILLEGAL (secure+s2 trsl)	Misconfiguration of the stream table entry: stage 2 translation is configured in a secure stream table entry

[Back to Top]

Codes in the gray window status bar at the bottom of the **SMMU.StreamTable** window indicate the current global fault / global error status of the SMMU:

MMU-400, MMU-401, MMU-500:
These codes for the global faults are MULTI, UUT, PF, EF, CAF, UCIF, UCBF, SMCF, USF, ICF [**A**]. These flags correspond to the flags of the SMMU_sGFSR register.

To view the descriptions of the global faults, double-click the gray window status bar to open the **SMMU.Register.Global** window [A]. Scroll down to the SMMU_sGFSR [**B**] or the SMMU_GERROR register. The global faults are described in the column on the right [**C**].

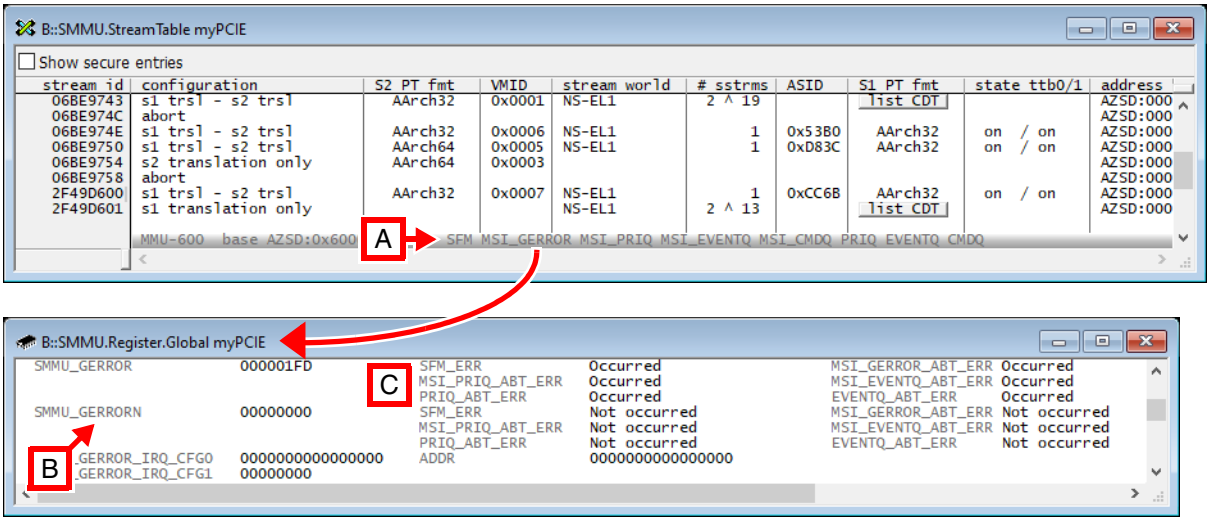


A Codes of global faults (for MMU-500 in this screen shot).

B The information about the global faults is derived from the register SMMU_sGFSR (secure global fault status register).

C Descriptions of the global faults in the **SMMU.Register.Global** window.

MMU-600 and newer:
These codes for the global errors are SFM, MSI_GERROR, MSI_PRIQ, MSI_EVENTQ, MSI_CMDQ, PRIQ, EVENTQ, CMDQ [**A**].
These flags correspond to the flags of the SMMU_GERROR register.



A Codes of global error flags (for MMU-600 in this screen shot).

B The information about the global error flags set is derived from an XOR operation for the registers SMMU_GERROR and SMMU_GERRORN.

C Descriptions of the global error flags in the **SMMU.Register.Global** window.

Finding streams which are in a fault / error state

MMU-400, MMU-401 and MMU-500:
A red letter in a **stage 1 cbndx state** column or a **stage 2 state** column of the **SMMU.StreamTable** window indicates a fault in a context bank. For descriptions of these faults, see **state** column.

MMU-600 and newer:
Use the Event Queue Window **SMMU.DumpQueue.Event** to view error events.
The command supplies options to filter and view events for a certain *<stream_id>* and/or *<substream_id>* range and it is possible to filter certain event types.

In **SMMU.StreamTable** or **SMMU.CtxtDescTable** window, use the popup menu entry **Dump associated Queue Entries** to dump queue entries for specific stream entry or context descriptor table entry.

SMMU.StreamTblEntry — Access to a stream table entry

MMU-600 and newer only

The **SMMU.StreamTblEntry** command group allows to view the details of the translation context associated with a Stream Table Entry and/or a stage 1 Context Descriptor. Every STE is identified by its *<stream_id>*. A CD is identified by both a *<stream_id>* and a *<substream_id>*. In case a stream table entry supports only a single stage 1 CD the *<substream_id>* can be omitted.

The **SMMU.StreamTblEntry** command group provides the following commands:

SMMU.StreamTblEntry.Register	Shows the registers of a STE or a CD.
SMMU.StreamTblEntry.list	Lists the page table associated with stage 1 or stage 2 translation in a compact format.
SMMU.StreamTblEntry.Dump	Dumps the page table entries associated with stage 1 or stage 2 translation page wise.

The three SMMU.StreamTblEntry commands feature common options:

- **/SUBstream** <*substream_id*>: apply the command for a CD with the <*substream_id*>
- **/SECure**: target the secure SMMU entries with the command

MMU-600 and newer only

Format:	**SMMU.StreamTableEntry.Dump** *<args>*
<args>:	*<name> <stream_id>* [*<address>* | *<range>* [*<ttb_address>*]] [***/SubStreamID <substream_id>***] [**/IntermediatePT**] [**/SECure**]

Opens the **SMMU.StreamTblEntry.Dump** window for the specified *<stream_id>*. This window dumps the page table content page-wise. If you prefer a compact view, use command **SMMU.StreamTblEntry.list**

If option **/SECure** is specified, the command targets the secure SMMU view.

You can dump any stage 1 or the stage 2 page table associated with the STE specified by *<stream_id>*.

To dump the stage 2 page table of the STE, specify only option **/IntermediatePT**.

To dump the stage 1 page table defined by a Context Descriptor of the STE, you must additionally specify the Substream ID of the Context Descriptor using option **/SubStreamID *<substream_id>***.

If no valid translation context is defined, the window displays the error message “registerset undefined”.

For a description of the columns in the **SMMU.StreamTableEntry.Dump** window, click here.

Arguments:

<name>	For a description of *<name>*, etc., click here.
<stream_id>	Defines the STE of which a page table has to be dumped.
<address> | *<range>*	If specified, start the dump with *<address>* or, alternatively, limit the dumped address range to address to *<range>.*
<ttb_address>	If specified, *<ttb_address>* will be used as page table base address. The other page table parameters are still extracted from the STE and/or CD context.
/SubStreamID *<substream_id>*	Omit this option to view translation table entries of stage 2. Include this option to view the stage 1 translation table entries of the Context Descriptor with substream *<substream_id>*. If the STE has only one Context Descriptor, you can omit option **/SubStreamID *<substream_id>***. In this case, the stage 1 page table of the Context Descriptor with substream 0 will be displayed. I
IntermediatePT	Omit this option to view translation table entries of stage 1. Include this option to view translation table entries of stage 2. In SMMUs that support only stage 2 page tables, this option can be omitted.

Examples:

```
;Dump the stage 2 page table of the STE with Stream ID 0x6BE974B for SMMU
"myGPU"
SMMU.StreamTblEntry.Dump myGPU 0x6BE974B /IntermediatePE

;Dump the stage 1 page table of Substream ID 0x2 which belongs to the STE
with Stream ID 0x6BE974B.
SMMU.StreamTblEntry.Dump myGPU 0x6BE974B /SubStreamID 0x2

;As above, but start dumping at address 0x80000000
SMMU.StreamTblEntry.Dump myGPU 0x6BE974B 0x80000000 /SubStreamID 0x2
```

To display an SMMU page table page-wise via the user interface TRACE32 PowerView, see here.

SMMU.StreamTblEntry.list — List page table entries

MMU-600 and newer only

Format:	**SMMU.StreamTableEntry.list***<args>*
<args>:	*<name> <stream_id>* [*<address>* | *<range>* [*<ttb_address>*]] [***/SubStreamID <substream_id>***] [**/IntermediatePT**] [**/SECure**]

Opens the **SMMU.StreamTblEntry.list** window for the specified *<stream_id>*. This window shows a compact list of consecutive address ranges in the page table which have a uniform, valid translation.

The syntax and arguments are identical to command **SMMU.StreamTblEntry.Dump** and are described there.

MMU-600 and newer only

Format:	**SMMU.Register.StreamTblEntry** *<args>*
<args> :	*<name> <stream_id>* [**/SubStreamID <substream_id>**] [**/SECure**]

If specified without option **/SubStreamID *<substream_id>***, this is an alias for command **SMMU.Register.StreamTblEntry.** It opens the peripheral register window for the SMMU named *<name>* and displays the registers of the Stream Table Entry which is specified by *<stream_id>*.

If specified with option **/SubStreamID *<substream_id>***, this command opens the peripheral register window for the SMMU named *<name>* and displays the registers of the Context Descriptor with substream *<substream_id>*, belonging to the Stream Table Entry with *<stream_id>*.

If option **/SECure** is specified, the command targets the secure SMMU view.

Example:

```
;list the registers of the Stream Table Entry with Stream ID 0x6B9743
from the secure Stream Table of SMMU "myGPU"
SMMU.StreamTable myGPU 0x6B9743 /SECure

;list the registers of the Context Descriptor with Substream ID 0x3,
belonging to the secure Stream Table Entry with Stream ID 0x6B9743
SMMU.StreamTable myGPU 0x6B9743 /SubStreamID 0x3 /SECure
```

Target Adaption

Probe Cables

For debugging two kind of probe cable can be used to connect the debugger to the target:
“Debug Cable” and “CombiProbe”

For off-chip program and data trace an additional trace probe cable “Preprocessor” is needed.

Interface Standards JTAG, Serial Wire Debug, cJTAG

Debug Cable and CombiProbe support JTAG (IEEE 1149.1), Serial Wire Debug (CoreSight ARM), and Compact JTAG (IEEE 1149.7, cJTAG) interface standards. The different modes are supported by the same connector. Only some signals get a different function. The mode can be selected by debugger commands. This assumes of course that your target supports this interface standard.

Serial Wire Debug is activated/deactivated by **SYStem.CONFIG DEBUGPORTTYPE [SWD | JTAG]**. In a multidrop configuration you need to specify the address of your debug client by **SYStem.CONFIG SWDPTARGETSEL**.

cJTAG is activated/deactivated by **SYStem.CONFIG DEBUGPORTTYPE [CJTAG | JTAG]**. Your system might need bug fixes which can be activated by **SYStem.CONFIG CJTAGFLAGS**.

Serial Wire Debug (SWD) and Compact JTAG (cJTAG) require a Debug Cable version V4 or newer (delivered since 2008) or a CombiProbe (any version) and one of the newer base modules (Power Debug Pro, Power Debug Interface USB 2.0/USB 3.0, Power Debug Ethernet, PowerTrace or Power Debug II).

Connector Type and Pinout

Debug Cable

Adaptation for ARM Debug Cable: See **https://www.lauterbach.com/adarmdbg.html**.

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).

CombiProbe

Adaptation for ARM CombiProbe: See **“Debug Probe Connectors”** in Arm Debug and Trace Interface Specification, page 17 (app_arm_target_interface.pdf).

The CombiProbe will always be delivered with 10-pin, 20-pin, 34-pin connectors. The CombiProbe can not detect which one is used. If you use the trace of the CombiProbe you need to inform about the used connector because the trace signals can be at different locations: **SYStem.CONFIG CONNECTOR [MIPI34 | MIPI20T]**.

If you use more than one CombiProbe cable (twin cable is no standard delivery) you need to specify which one you want to use by **SYStem.CONFIG DEBUGPORT [DebugCableA | DebugCableB]**. The CombiProbe can detect the location of the cable if only one is connected.

Preprocessor

Adaptation for ARM ETM Preprocessor Mictor: See **https://www.lauterbach.com/adetmmictor.html**.

Adaptation for ARM ETM Preprocessor MIPI-60: See **https://www.lauterbach.com/adetmmipi60.html**.

Adaptation for ARM ETM Preprocessor HSSTP: See **https://www.lauterbach.com/adetmhsstp.html**.