ARMv8-A/-R Debugger

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ..... 🗁

ICD In-Circuit Debugger ..... 🗁

Processor Architecture Manuals ..... 🗁

ARM/CORTEX/XSCALE ..... 🗁

ARMv8-A/-R Debugger 1

History 7

Warning 9

Introduction 10

Brief Overview of Documents for New Users 10

Demo and Start-up Scripts 11

Quick Start of the JTAG Debugger 13

Configure Debugger for SoC Specific Reset Behavior 17

Troubleshooting 31

Communication between Debugger and Processor cannot be established 31

FAQ 32

Trace Extensions 32

Quick Start for Multicore Debugging 33

SMP Debugging - Quick Start 34

1. How to Debug a System with Multiple Identical Cores 34

2. Set up the SMP Debug Scenario 35

3. Enter Debug Mode 36

4. Switch Debug View between Cores 36

5. Write a Start-up Script Summary 36

AMP Debugging - Quick Start 37

1. How to Debug a System with Multiple Heterogenous Cores 37

2. Starting the TRACE32 PowerView GUIs 37

3. Master-Slave Concept 38

4. Setting up the Multicore Environment 38

5. Synchronized Go / Step / Break 39

6. Write a Start-up Script Summary 39

ARM Specific Implementations 40

AArch Mode Support 40

©1989-2020 Lauterbach GmbH

AArch64 and AArch32 Debugging 40
AArch64 and AArch32 Switching 41
TrustZone Technology 43
AArch64 Secure Model 43
AArch32 Secure Model 44
Debug Permission 44
Checking Debug Permission 44
Checking Secure State 45
Changing the Secure State from within TRACE32 45
AArch64 System Registers Access 46
AArch32 Coprocessor Registers Access 46
Accessing Cache and TLB Contents 46
Breakpoints and Vector Catch Register 46
Breakpoints and Secure Modes 46
big.LITTLE 47
Debugger Setup 47
Consequence for Debugging 48
Requirements for the Target Software 48
big.LITTLE MP 48
Breakpoints 49
Software Breakpoints 49
On-chip Breakpoints for Instructions 49
On-chip Breakpoints for Data 49
Example for Standard Breakpoints 50
Secure, Non-Secure, Hypervisor Breakpoints 51
Example for ETM Stopping Breakpoints 57
Access Classes 58
System Registers (AArch64 Mode) 66
Coprocessors (AArch32 Mode) 69
Accessing Memory at Run-time 73
Semihosting 77
AArch64 HLT Emulation Mode 78
AArch64 DCC Communication Mode (DCC = Debug Communication Channel) 79
AArch32 SVC (SWI) Emulation Mode 80
AArch32 DCC Communication Mode (DCC = Debug Communication Channel) 81
Virtual Terminal 83
Large Physical Address Extension (LPAE) 84
Consequence for Debugging 84
Virtualization Extension, Hypervisor 85
Consequence for Debugging 85
Debug Field 86
Run Mode 86
Run-time Measurements 87

©1989-2020 Lauterbach GmbH

Trigger 87

ARM specific SYStem Commands 88

SYStem.CLOCK Inform debugger about core clock 88
SYStem.CONFIG.state Display target configuration 88
SYStem.CONFIG Configure debugger according to target topology 89
<parameters> describing the “DebugPort” 96
<parameters> describing the “JTAG” scan chain and signal behavior 101
<parameters> describing a system level TAP “Multitap” 105
<parameters> configuring a CoreSight Debug Access Port “DAP” 107
<parameters> describing debug and trace “Components” 111
<parameters> which are “Deprecated” 121
SYStem.CONFIG.EXTWDTDIS Disable external watchdog 126
SYStem.CONFIG GICD Generic Interrupt Controller Distributor (GIC) 127
SYStem.CONFIG GICR Generic Interrupt Controller Redistributor 130
SYStem.CONFIG GICC Generic Interrupt Controller physical CPU interface 131
SYStem.CONFIG GICH Generic Interrupt Controller virtual interface control 132
SYStem.CONFIG GICV Generic Interrupt Controller virtual CPU interface 133
SYStem.CONFIG SMMU Internal use 134
SYStem.CPU Select the used CPU 136
SYStem.JtagClock Define the frequency of the debug port 136
SYStem.LOCK Tristate the JTAG port 138
SYStem.MemAccess Run-time memory access 139
SYStem.Mode Establish the communication with the target 142
SYStem.Option Special setup 144
SYStem.Option Address32 Define address format display 144
SYStem.Option AHBHPROT Select AHB-AP HPROT bits 144
SYStem.Option AXI32 Use 32-bit atomic AXI accesses instead of 64-bit 145
SYStem.Option AXIACEEnable ACE enable flag of the AXI-AP 145
SYStem.Option AXICACHEFLAGS Select AXI-AP CACHE bits 145
SYStem.Option AXIHPROT Select AXI-AP HPROT bits 146
SYStem.Option BreakOS Allow break during OS-unlock 147
SYStem.Option CacheStatusCheck Check status bits during cache access 148
SYStem.Option CFLUSH FLUSH the cache before step/go 148
SYStem.Option CLTAPKEY Set key values for CLTAP operation 148
SYStem.Option CoreSightRESet Assert CPU reset via CTRL/STAT 148
SYStem.Option CTITimerStop Stop system timer when CPU stops 149
SYStem.Option DAPDBGPWRUPREQ Force debug power in DAP 149
SYStem.Option DAP2DBGPWRUPREQ Force debug power in DAP2 150
SYStem.Option DAPNOIRCHECK No DAP instruction register check 151
SYStem.Option DAPREMAP Rearrange DAP memory map 151
SYStem.Option DAPSYSPWRUPREQ Force system power in DAP 152
SYStem.Option DAP2SYSPWRUPREQ Force system power in DAP2 152
SYStem.Option DBGSPR Use debugger view for SPR access 153

©1989-2020 Lauterbach GmbH

SYStem.Option DBGUNLOCK Unlock debug register via OSLAR 153
SYStem.Option DCacheMaintenance Data cache maintenance strategy 153
SYStem.Option DEBUGPORTOptions Options for debug port handling 154
SYStem.Option DIAG Activate more log messages 155
SYStem.Option DUALPORT Implicitly use run-time memory access 155
SYStem.Option DisMode Define disassembler mode 156
SYStem.Option EnReset Allow the debugger to drive nRESET (nSRST) 157
SYStem.Option eXclusiveMONitor Support for exclusive monitors 157
SYStem.Option HRCWOVerRide Enable override mechanism 157
SYStem.Option ICacheMaintenance I-Cache maintenance strategy 158
SYStem.Option IMASKASM Disable interrupts while single stepping 158
SYStem.Option IMASKHLL Disable interrupts while HLL single stepping 159
SYStem.Option INTDIS Disable all interrupts 159
SYStem.Option IntelSOC Slave core is part of Intel® SoC 159
SYStem.Option MACHINESPACES Address extension for guest OSes 160
SYStem.Option MEMORYHPROT Select memory-AP HPROT bits 160
SYStem.Option MemStatusCheck Check status bits during memory access 161
SYStem.Option MMUPhysLogMemaccess Memory access preferences 161
SYStem.Option MMUSPACES Separate address spaces by space IDs 162
SYStem.Option NOMA Use alternative memory access 163
SYStem.Option NoPRCRReset Disable warm reset via PRCR 164
SYStem.Option OSUnlockCatch Use the 'OS Unlock Catch' debug event 164
SYStem.Option OVERLAY Enable overlay support 165
SYStem.Option PALLADIUM Extend debugger timeout 165
SYStem.Option PWRDWN Allow power-down mode 166
SYStem.Option PAN Overwrite CPSR.PAN setting 166
SYStem.Option PWRREQ Request core power 166
SYStem.Option ResBreak Halt the core after reset 167
SYStem.Option ResetDetection Choose method to detect a target reset 168
SYStem.RESetOut Assert nRESET/nSRST on JTAG connector 168
SYStem.Option RESetREGister Generic software reset 169
SYStem.Option RisingTDO Target outputs TDO on rising edge 169
SYStem.Option SLaVeSOFTRESet Allow soft reset of slave cores 170
SYStem.Option SMPMultipleCall Send start event to each SMP core 170
SYStem.Option SOFTLONG Use 32-bit access to set breakpoint 170
SYStem.Option SOFTQUAD Use 64-bit access to set breakpoint 170
SYStem.Option STEPSOFT Use software breakpoints for ASM stepping 171
SYStem.Option SOFTWORD Use 16-bit access to set breakpoint 171
SYStem.Option TURBO Disable cache maintenance during memory access 171
SYStem.state Display SYStem window 172
SYStem.Option SYSPWRUPREQ Force system power 172
SYStem.Option TRST Allow debugger to drive TRST 173
SYStem.Option WaitDAPPWR Wait for DAP power after DAP power request 173

©1989-2020 Lauterbach GmbH

SYStem.Option WaitDBGREG Wait for core debug registers after reset 174
SYStem.Option WaitIDCODE IDCODE polling after deasserting reset 175
SYStem.Option WaitReset Wait with JTAG activities after deasserting reset 176
SYStem.Option ZoneSPACES Enable symbol management for ARM zones 177
Overview of Debugging with Zones 178
Operation System Support - Defining a Zone-specific OS Awareness 181
SYStem.Option ZYNQJTAGINDEPENDENT Configure JTAG cascading 183

ARM specific Functions 184
STATE.NOCTIACCESS() 184
STATE.NOCPUACCESS() 184

ARM specific Benchmarking Commands 185
BMC.<counter>.CountEL<x> Select exception level events to be counted 185
BMC.EXPORT Export benchmarking events from event bus 187
BMC.LongCycle Configure cycle counter width 188
BMC.PRESCALER Prescale the measured cycles 189

ARM specific TrOnchip Commands 189
TrOnchip.ContextID Enable context ID comparison 190
TrOnchip.CONVert Allow extension of address range of breakpoint 191
TrOnchip.MachineID Extend on-chip breakpoint/trace filter by machine ID 192
TrOnchip.MatchASID Extend on-chip breakpoint/trace filter by ASID 193
TrOnchip.MatchMachine Extend on-chip breakpoint/trace filter by machine 193
TrOnchip.MatchZone Extend on-chip breakpoint/trace filter by zone 194
TrOnchip.RESERVE Exclude breakpoint or watchpoint from debugger usage 195
TrOnchip.RESet Set on-chip trigger to default state 195
TrOnchip.Set Set bits in the vector catch register 196
TrOnchip.StepVector Step into exception handler 198
TrOnchip.StepVectorResume Catch exceptions and resume single step 198
TrOnchip.VarCONVert Convert breakpoints on scalar variables 199
TrOnchip.state Display on-chip trigger window 200

Cache Analysis and Maintenance 201
TRACE32 Cache Support by CPU Type 202

CPU specific MMU Commands 204
MMU.DUMP Page wise display of MMU translation table 204
MMU.List Compact display of MMU translation table 215
MMU.SCAN Load MMU table from CPU 217
TRACE32 TLB Support by CPU Type 219

CPU specific SMMU Commands 220
SMMU Hardware system MMU (SMMU) 220
SMMU.ADD Define a new hardware system MMU 224
SMMU.Clear Delete an SMMU 225
SMMU.Register Peripheral registers of an SMMU 226

©1989-2020 Lauterbach GmbH

SMMU.Register.ContextBank Display registers of context bank 227
SMMU.Register.Global Display global registers of SMMU 228
SMMU.Register.StreamMapRegGrp Display registers of an SMRG 229
SMMU.RESet Delete all SMMU definitions 230
SMMU.SSDtable Display security state determination table 231
SMMU.StreamMapRegGrp Access to stream map table entries 233
SMMU.StreamMapRegGrp.ContextReg Display context bank registers 234
SMMU.StreamMapRegGrp.Dump Page-wise display of SMMU page table 236
SMMU.StreamMapRegGrp.list List the page table entries 238
SMMU.StreamMapTable Display a stream map table 239

Target Adaption 246
Probe Cables 246
Interface Standards JTAG, Serial Wire Debug, cJTAG 246
Connector Type and Pinout 247
Debug Cable 247
CombiProbe 247
Preprocessor 247

©1989-2020 Lauterbach GmbH

Version 18-Nov-2020

History

17-Jun-20	New function STATE.NOCTIACCESS().
17-Jun-20	New function STATE.NOCPUACCESS().
04-Mar-20	New command SYStem.Option.AXI32.
12-Jul-19	Renamed some TrOnchip commands to Break.CONFIG.*<sub_cmd>*. For a list of renamed commands, see "Deprecated vs. New Commands".
10-Jul-19	New commands: SYStem.CpuBreak and SYStem.CpuSpot.
10-Jul-19	Updated SYStem.MemAccess: (a) new subcommand SYStem.MemAccess StopAndGo, (b) renamed SYStem.MemAccess CPU to SYStem.MemAccess Enable, (c) updated SYStem.MemAccess DAP.
10-Jul-19	The command group SYStem.CpuAccess.* is deprecated, use SYStem.MemAccess StopAndGo and SYStem.CpuBreak Denied instead.
24-Jun-19	Extended benchmark counter description for Armv8-R.
18-Apr-19	New chapter "Configure Debugger for SoC Specific Reset Behavior". New commands SYStem.Option CTITimerStop, SYStem.Option WaitDBGREG, SYStem.Option WaitDAPPWR.
27-Mar-19	New sections "Coprocessor Converter Dialog" and "SPR Converter Dialog".
22-Mar-19	Changed default value of SYStem.Option.NoPRCRReset from 'OFF' to 'ON'.
22-Feb-19	New command: SYStem.CONFIG.EXTWDTDIS.
22-Jan-19	Added descriptions for Armv8.4-A related commands: BMC.<counter>.CountEL<x> and TrOnchip.Set.
22-Jan-19	New chapters "TRACE32 Cache Support by CPU Type" and "TRACE32 TLB Support by CPU Type".
22-Jan-19	Updated chapters "How to Create Valid Access Class Combinations", "AArch64 and AArch32 Switching", and "AArch64 Secure Model".

©1989-2020 Lauterbach GmbH

26-Nov-18	Updated SYStem.JtagClock.
25-Oct-18	Removed SYStem.Option DBGAKC as it does not apply for ARMv8 based architectures.
21-Sep-18	Added description for the command TrOnchip.MatchZone.
01-Aug-18	Extended chapter “Access Classes” with description of the access classes MSPR, HSPR, ZSPR, NSPR.
16-Jul-18	Added descriptions and examples for the command groups SYStem.CONFIG GICD, SYStem.CONFIG GICR, SYStem.CONFIG GICC, SYStem.CONFIG GICH, and SYStem.CONFIG GICV.
27-Mar-18	New command SYStem.Option.WaitIDCODE.
23-Jan-18	New chapter “Accessing Memory at Run-time”.
11-Jan-18	Added description for SYStem.Option CoreSightRESet.
11-Jan-18	Added description for the new internal command SYStem.CONFIG.SMMU.

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

Introduction

This manual serves as a guideline for debugging Cortex-A/R (ARMv8, 32/64-bit) cores and describes all processor-specific TRACE32 settings and features.

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

Brief Overview of Documents for New Users

Architecture-independent information:

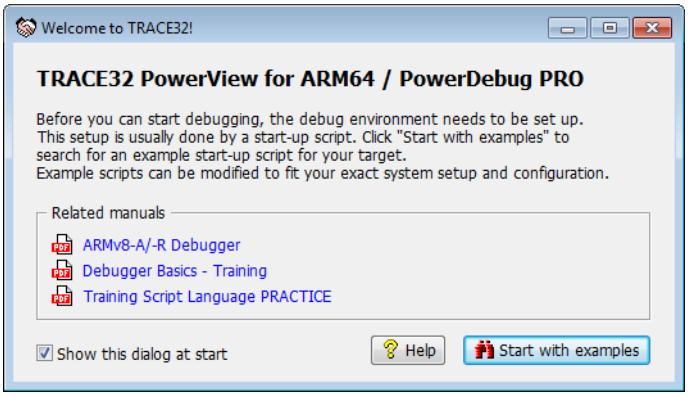
- **"Debugger Basics - Training"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.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

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.
- This manual does not cover the Cortex-A/R (ARMv7, 32-bit) cores. If you are using this processor architecture, please refer to **"ARM Debugger"** (debugger_arm.pdf).
- This manual does not cover the Cortex-M processor architecture. If you are using this processor architecture, please refer to **"Cortex-M Debugger"** (debugger_cortexm.pdf) for details.

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

©1989-2020 Lauterbach GmbH



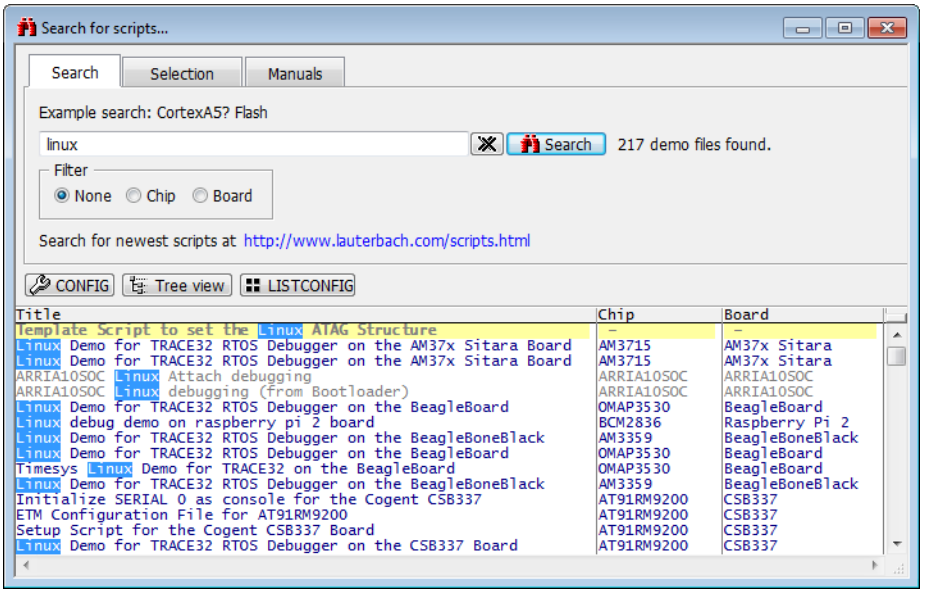
Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known Cortex-A/R (ARMv8, 32/64-bit)-based hardware.

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

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

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



©1989-2020 Lauterbach GmbH

You can also inspect the demo folder manually in the system directory of TRACE32.
The ~~/demo/arm64/ and the ~~/demo/arm/ folders contain:

hardware/	Ready-to-run debugging and flash programming demos for evaluation boards. **Recommended for getting started!**
bootloader	Examples for uboot, uefi and other bootloaders
compiler/	Hardware independent compiler examples.
etc/	Various examples, e.g. data trace, terminal application, …
fdx/	Example applications for the FDX feature.
flash/	Binaries for target based programming and example declarations for internal flash.
kernel/	Various OS Awareness examples.

©1989-2020 Lauterbach GmbH

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

```
B::

RESet
```

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

2. Specify the CPU/SoC specific settings.

```
SYStem.CPU <cpu_type>                ; Default <cpu_type>: CortexA53

SYStem.Option EnReset [ON|OFF]       ; Default: ON
```

Usually the access ports, the core debug base and the CTI base are set when the CPU/SoC is selected, so the following **SYStem.CONFIG** commands would not be required. However, for generic cores like Cortex-A53 or Cortex-A57, these settings are not known. The configuration may contain default values that do not meet your use case.
Please check the **SYStem.CONFIG.state /COmponents** window to find out if these addresses are set. If not, you have to set them manually, e.g. in a script:

```
SYStem.CONFIG COREDEBUG Base DAP:<core_base>

SYStem.CONFIG CTI Base DAP:<cti_base>

SYStem.CONFIG DEBUGACCESSPORT <apb_num>

SYStem.CONFIG APBACCESSPORT <apb_num>

SYStem.CONFIG AXIACCESSPORT <axi_num>

SYStem.CONFIG MEMORYACCESSPORT <axi_num>
```

All other default values are set in such a way that it should be possible to work without modification. This might not be the best configuration for your target.

In most cases DEBUGACCESSPORT will be an equivalent to APBACCESSPORT, and MEMORYACCESS will be an equivalent to AXIACCESSPORT (or AHBACCESSPORT if an AHB but no AXI is not present). It is recommended to always specify the access port for debug, memory and the APB, AHB or AXI bus.

©1989-2020 Lauterbach GmbH

3. Enter debug mode.

```
SYStem.Mode Up                    ; Reset target and enter debug mode
```

This command resets the CPU/SoC and enters debug mode. After this command is executed it should be possible to access memory and registers. The default case assumes that the debug interface is accessible while the CPU/SoC is held in reset. If this is not the case, SYStem.Mode Up may fail with an error. Please note that the reset behavior can be controlled with additional options. Please see the **SYStem.Option** section.

Another possibility to enter debug mode is to attach to the CPU and to stop it:

```
SYStem.Mode Attach                ; Attach without reset
Break.direct                      ; Enter debug mode
```

This might be helpful in case a reset of the target CPU is not desired or when the debug interface is not immediately accessible during or shortly after a reset of the CPU.

4. Load the program.

```
Data.LOAD.ELF armf            ; .ELF specifies the format
                              ; armf is the file name
```

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

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

A typical core configuration might look like this. All addresses and access port values are only examples.

```
RESet

SYStem.CPU    CortexA53                     // Select your SoC/CPU/Core
SYStem.CONFIG COREDEBUG Base DAP:0x80010000 // Set your core base here
SYStem.CONFIG CTI       Base DAP:0x80020000 // Set your CTI base here
SYStem.CONFIG DEBUGACCESSPORT  0.
SYStem.CONFIG APBACCESSPORT    0.
SYStem.CONFIG MEMORYACCESSPORT 1.
SYStem.CONFIG AXIACCESSPORT    1.
```

©1989-2020 Lauterbach GmbH

A typical start sequence is shown below.

```
SYStem.Mode Up

// Load position independent demo application to 0x2E000000
DO ~~/demo/arm64/compiler/gnu-pic/demo_sieve.cmm 0x2E000000

// Specify HLL source file if located elsewhere
sYmbol.SourcePATH ~~/demo/arm64/compiler/arm/sieve.c

// Example Breakpoint Setup
Break.Delete /all                       // Reset Breakpoint
                                        // configuration
Go.direct main                          // Run to main function
Break.Set 0x2E009080 /Program           // Set breakpoint to address
                                        // 0x2E009080
WinCLEAR                                // Close all windows

// Open useful windows
WinPOS 0.0 0.0 75. 23. 24. 1. W000
WinTABS 10. 10. 25. 62.
List.Mix                                // Open source code window   *)

WinPOS 80.5 17.5 52. 20. 0. 0. W001
Register.view /SpotLight                // Open register window      *)
                                        // and show changes of registers
WinPOS 0.0 29.7 41. 6. 5. 0. W002
Frame.view /Locals /Caller              // Open the stack frame with
                                        // local variables           *)
WinPOS 46.8 29.7 28. 6. 0. 0. W003
Var.Watch xx c p                        // Add variables to watch
                                        // window                    *)
WinPOS 80.5 0.0 52. 12. 0. 0. W004
PER.view                                // Show clearly arranged
                                        // peripherals in window     *)
```

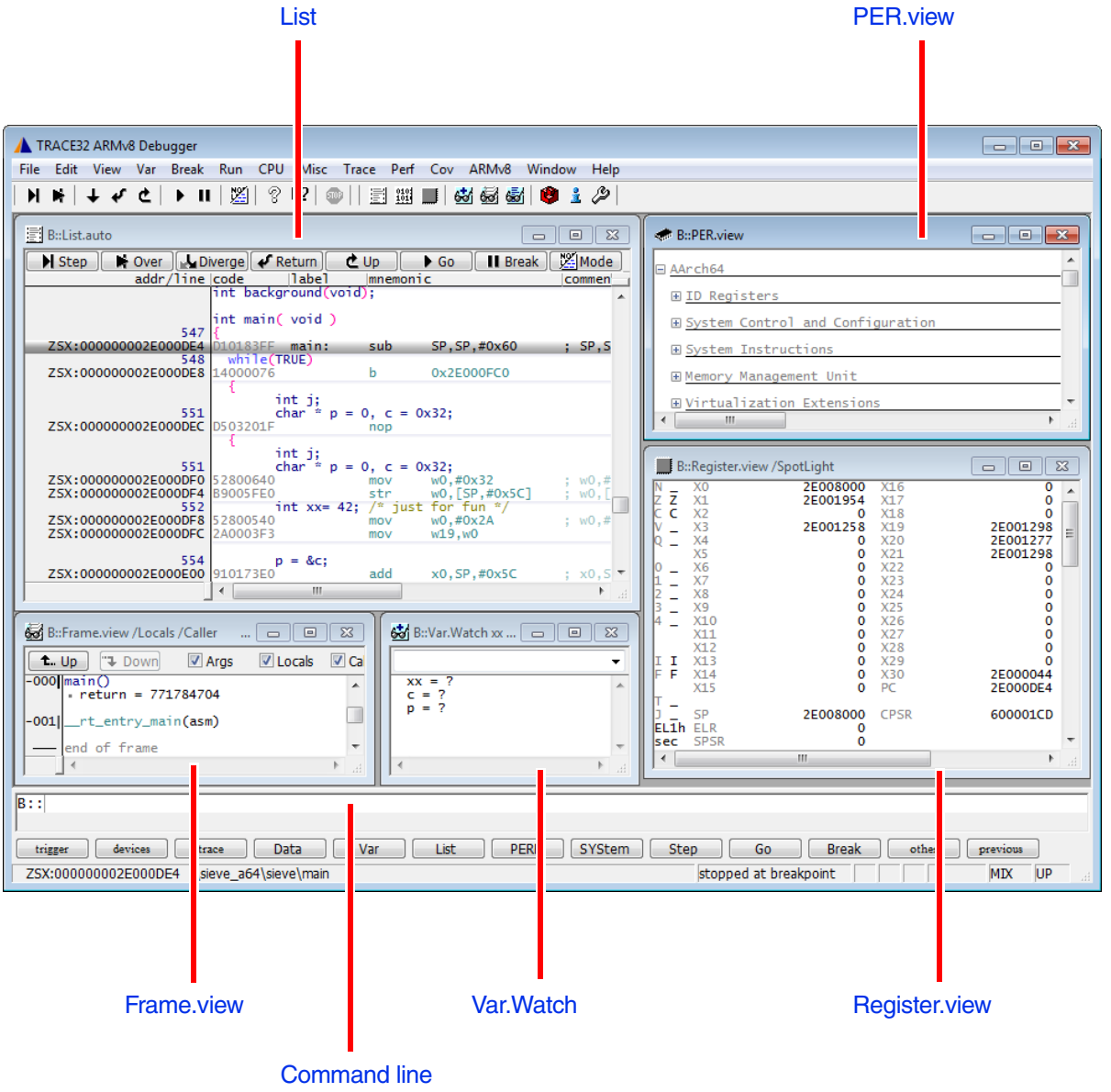
*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

Depending on your processor you might need additional settings. These sequences can be written altogether into a PRACTICE start-up script like *<your_own_startup>*.cmm. The configuration can be reproduced by calling this script:

```
CD.DO <your_own_startup>.cmm
```

©1989-2020 Lauterbach GmbH

A debug session might look like this:



©1989-2020 Lauterbach GmbH

Configure Debugger for SoC Specific Reset Behavior

All scenarios discussed in this section will focus on the first core in a SoC. This is usually the boot core which is therefore the first core to become active after a reset or power cycle. This section shows how the debugger can be configured to gain control over the boot core as soon as possible.

Multicore systems might require additional steps to gain control over the secondary cores. Most of this procedure is SoC specific. Therefore, only a rough outline will be given towards the end of this section.

In this section:

- Terms, Abbreviations, and Definitions
- Connect to SoC while in Reset
- Connect to SoC when JTAG is not Available in Reset
- Connect to SoC when CoreSight Subsystem is not Available in Reset
- Connect to SoC when Multiple Debug Resources are not Available in Reset
- Connect to SoC when Debug Registers are not Available in Reset
- Connect to SoC when Device is not Available in Reset for an Unknown Reason
- Connect to Secondary Cores
- Troubleshooting
- Summary

Terms, Abbreviations, and Definitions

nSRST	This is the physical reset line, for resetting the SoC. If this line is asserted, the entire SoC or a part of it, e.g. the Arm cores, will be kept in reset. This behavior is SoC specific.
nTRST	The physical reset line that resets the JTAG TAP controller, if asserted.
JTAG-ID	An identification code that can be read by the debugger on JTAG level. This feature is provided by the JTAG TAP itself and is therefore independent of the presence of a CPU or additional debug infrastructure.
DAP	The Arm Debug Access Port which is part of the Arm CoreSight system.
SoC	The whole processing entity that includes the discussed Arm cores, optionally non-Arm architecture cores and custom peripherals.
Debug resource	This term will be used to summarize the terms “JTAG”, “CoreSight subsystem (DAP)” or just “DAP” and “core debug registers”. A “debug resource” is therefore a module or set of registers needed for debugging.
Debug power	The debug power domain in a SoC. Usually power to this domain is already present or can be requested via a DAP control register. Some essential core debug registers are located in the debug power domain.

©1989-2020 Lauterbach GmbH

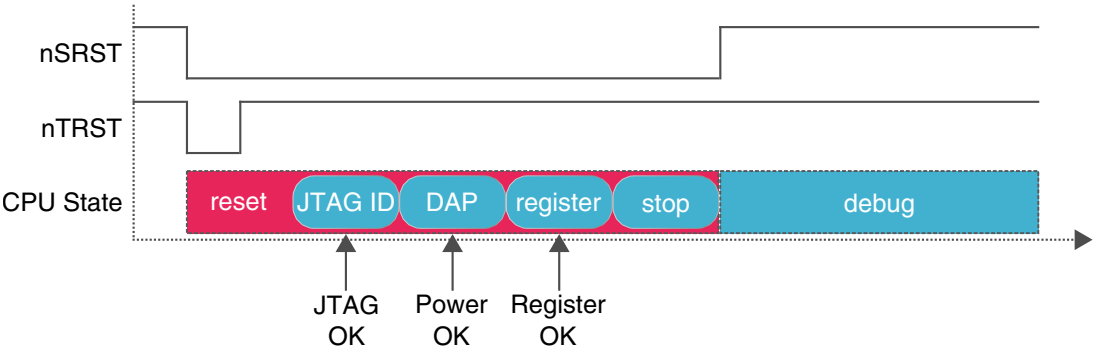
Without additional configuration, the debugger will try the following to connect to a core in the SoC during **SYStem.Mode Up**.

Initial Situation:

- The debug resources, i.e. JTAG, the CoreSight subsystem, etc., are accessible in reset state.
- The core debug registers are accessible in reset state.

Debugger Connect Sequence of SYStem.Mode Up:

- Assert the nSRST and nTRST line.
- Release the nTRST line, but keep nSRST asserted, i.e. keep SoC/CPU/core in reset.
- Configure core for debugging. This means, the debugger will check the JTAG ID, make sure the debug power domain is active, and check if the core debug registers are accessible. Then the debugger will stop the core. If possible, the core is configured for being caught at the reset vector after reset release. This is SoC specific.
- Release the nSRST line, the boot core shall directly stop in debug mode.
- Collect all information about the current state of the core. The system is now ready for debug.



Example:

```
RESet                       // Reset debugger configuration.
                            // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.              // Only connect to boot core (multicore systems)
SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

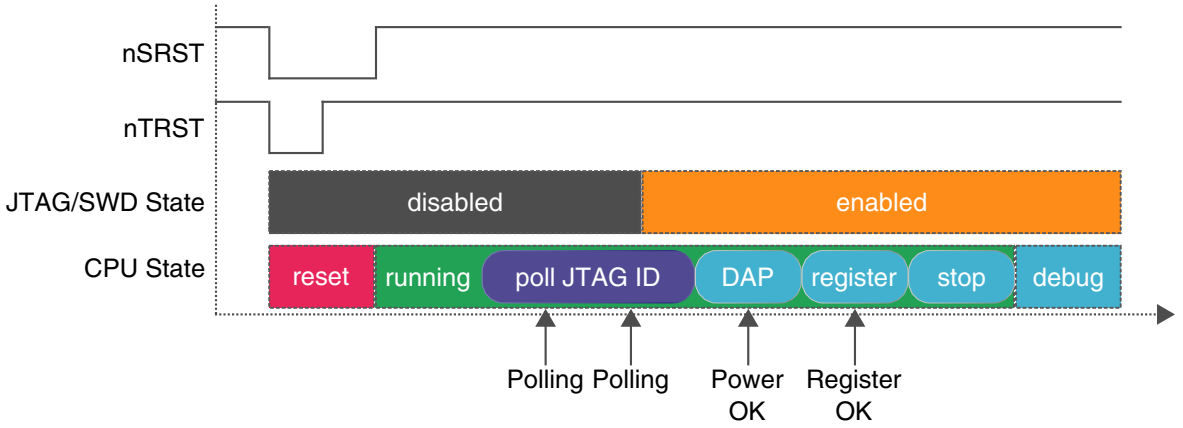
The following scenario assumes that the JTAG interface is not available to the debugger while the SoC is kept in reset.

Initial Situation:

- The boot process on the SoC will unlock the JTAG interface. If this is not happening, the debugger will be kept out of the device and cannot connect for debugging.

Debugger Connect Sequence of SYStem.Mode Up:

- Assert the nSRST and nTRST line.
- Release the nTRST line, release nSRST line early.
- SoC/core starts execution.
- After releasing reset, the debugger will continuously poll the JTAG ID-Code until a meaningful value is read. The intention is to stop the core as soon as possible.
- Once a valid ID-Code is read, the debugger will make sure debug power is available, and check if the core debug registers are accessible. Then the debugger will stop the boot core.
- Collect all information about the current state of the core. The system is now ready for debug.



Consequence:

- The core might not be halted at the reset vector, this is SoC specific. Parts of the SoC might already be initialized. This means, debugging will not start in a reset context.

Workaround:

Place an endless loop in your boot code to prevent further code execution. Depending on the SoC boot process only one of the following items apply:

- The boot core you want to connect to has to unlock JTAG itself. Therefore, place the endless loop immediately after the JTAG interface has been unlocked...
- ...or JTAG is unlocked in an earlier boot phase of the SoC, e.g. a separate core for reset and power management. Therefore, you can place the endless loop at the first instruction of the application core you want to debug.

©1989-2020 Lauterbach GmbH

Benefits of this option:

- The debugger will connect as soon as possible to the CPU. This means the amount of executed code is minimized.
- The wait time is dynamic within the given *<time>* range. For details, please see **SYStem.Option WaitIDCODE**. If boot timings change, the wait time adapts automatically.

Example:

```
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option ResBreak OFF      // Early release of nSRST
SYStem.Option WaitIDCODE 500ms  // Poll 500ms or less for ID-Code
SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

The following scenarios focus on the state of the CoreSight subsystem (DAP):

1. The debug power domain cannot be activated in reset, this is the most prominent case.
2. The debug power domain can be activated in reset, i.e. nSRST is still asserted.

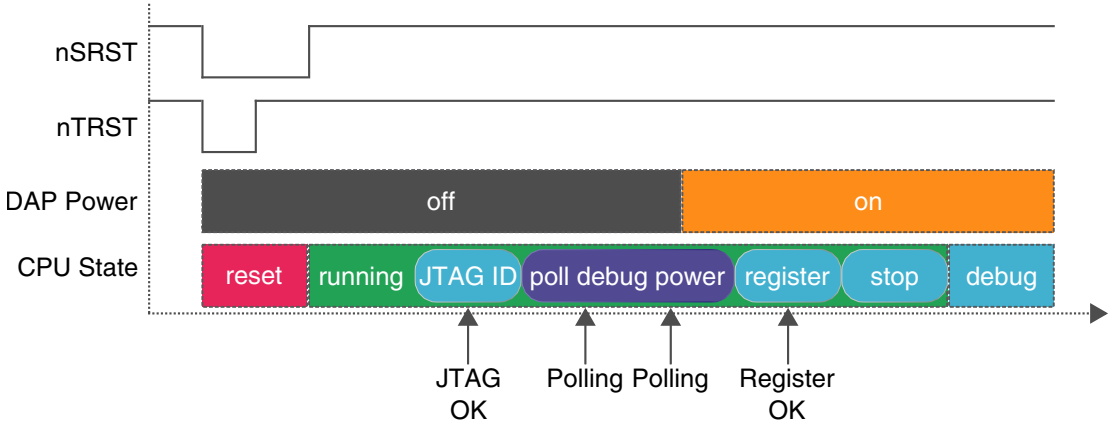
The following sequence and figure shows case 1). The examples will show case 1) and 2).

Initial Situation:

- The debug power domain cannot be activated while the SoC is in reset.
- The nSRST line has to be released shortly after its assertion.

Debugger Connect Sequence of SYStem.Mode Up:

- Assert the nSRST and nTRST line.
- Release the nTRST line, release nSRST line early.
- JTAG is available, the JTAG-ID can immediately be checked.
- The debugger requests power to the debug power domain and polls the power state.
- After the debug power domain is available, the debugger will stop the boot core.
- Collect all information about the current state of the core. The system is now ready for debug.



Consequence:

- The core might execute some code. The peripherals of the SoC might already be changed when the core is stopped in debug state. You will not debug in a reset context.

Benefit of this option:

- The debugger will connect as soon as possible to the CPU. This means the amount of executed code is minimized.
- The wait time is dynamic within the given *<time>* range. For details, please see **SYStem.Option WaitDAPPWR**. If boot timings change, the wait time adapts automatically.

©1989-2020 Lauterbach GmbH

Example case 1)

```
// This example assumes that the DAP cannot be powered in reset
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option ResBreak OFF      // Early release of nSRST line.
SYStem.Option WaitDAPPWR 50ms   // Poll 50ms or less for DAP power
SYStem.Mode Up
```

Example case 2)

```
// This example assumes that the DAP can be powered in reset
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option WaitDAPPWR 50ms   // Poll 50ms or less for DAP power
SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

The following scenarios focus on the state of the core debug registers:

1. The debug register are not available in reset. This is the most prominent use case.
2. The debug registers will become available after reset has been asserted, if the SoC is kept in reset, i.e. nSRST is kept asserted.

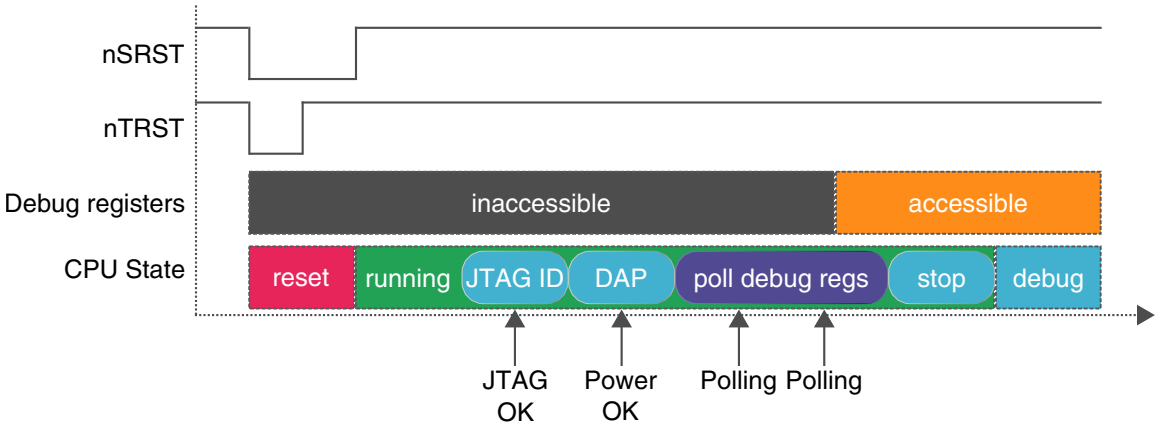
The following sequence and figure shows case 1). The examples will show case 1) and 2).

Initial situation:

- The cores debug registers cannot be accessed while the SoC is in reset.

Debugger Connect Sequence of SYStem.Mode Up:

- Assert the nSRST and nTRST line.
- Release the nTRST line, release nSRST line early.
- JTAG is available, the JTAG-ID can immediately be checked.
- The debug power domain is available.
- The debugger will poll the debug register state of the core until they become available. The debugger will stop the boot core.
- Collect all information about the current state of the core. The system is now ready for debug.



Consequence:

- The core might execute some code. The peripherals of the SoC might already be changed when the core is stopped in debug state. You will not debug in a reset context.

Benefit:

- The debugger will connect as soon as possible to the CPU. This means the amount of executed code is minimized.
- The wait time is dynamic within the given *<time>* range. For details, please see **SYStem.Option WaitDBGREG**. If boot timings change, the wait time adapts automatically.

©1989-2020 Lauterbach GmbH

Example case 1)

```
// Assumes that the core debug registers are not accessible in reset
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option ResBreak OFF      // Early release of nSRST line.
SYStem.Option WaitDBGREG 50ms   // Poll 50ms or less for core debug
                                // registers to become accessible
SYStem.Mode Up
```

Example case 2)

```
// Assumes that the core debug registers are accessible in reset
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option WaitDBGREG 50ms   // Poll 50ms or less for core debug
                                // registers to become accessible
SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

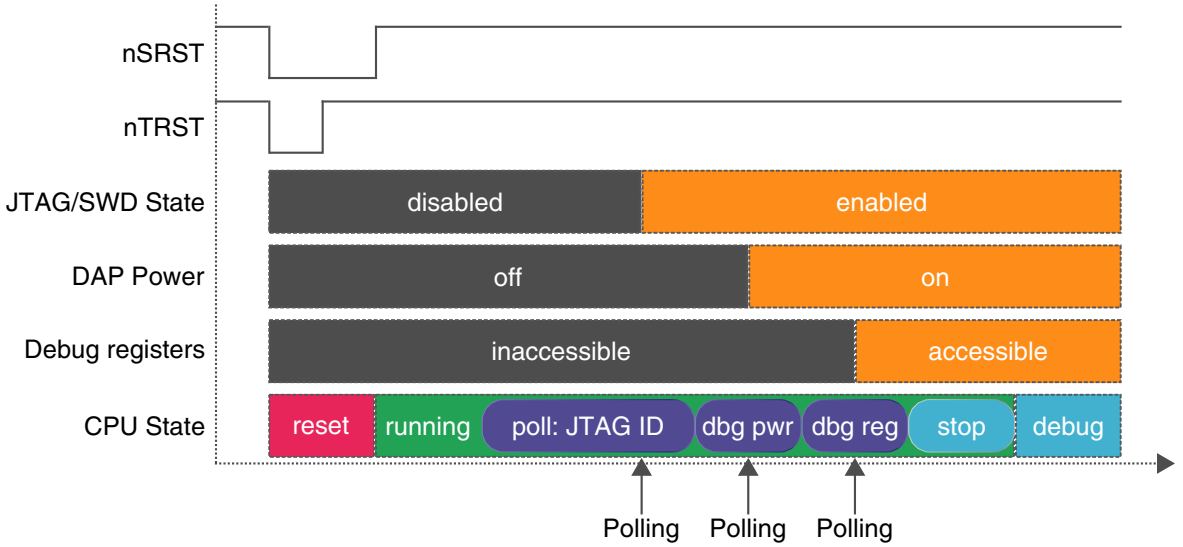
We will now focus on a scenario, where the debugger has to wait for all the three debug resources, because each single resource needs a certain enable time.

Initial situation:

- Neither JTAG, nor the debug power domain, nor the debug registers are accessible in reset.

Debugger Connect Sequence of SYStem.Mode Up:

- Assert the nSRST and nTRST line.
- Release the nTRST line, release nSRST line early.
- After releasing reset, the debugger will continuously poll the JTAG ID-Code until a meaningful value is read.
- The debugger continues and requests power to the debug power domain; however, this is not immediately happing. The debugger polls the power state until the debug power is available.
- Next the debugger will check the accessibility of the debug registers. Those are not immediately available, so the debugger has to wait until they are available.
- The debugger will stop the boot core.
- Collect all information about the current state of the core. The system is now ready for debug.



©1989-2020 Lauterbach GmbH

Consequence:

- The core might execute some code. The peripherals of the SoC might already be changed when the core is stopped in debug state. You will not debug in a reset context.

Benefit:

- The debugger will connect as soon as possible to the CPU. This means the amount of executed code is minimized.
- The wait time is dynamic within the given *<time>* range. For details, please see:
  - **SYStem.Option WaitIDCODE**
  - **SYStem.Option WaitDAPPWR**
  - **SYStem.Option WaitDBGREG**

Example:

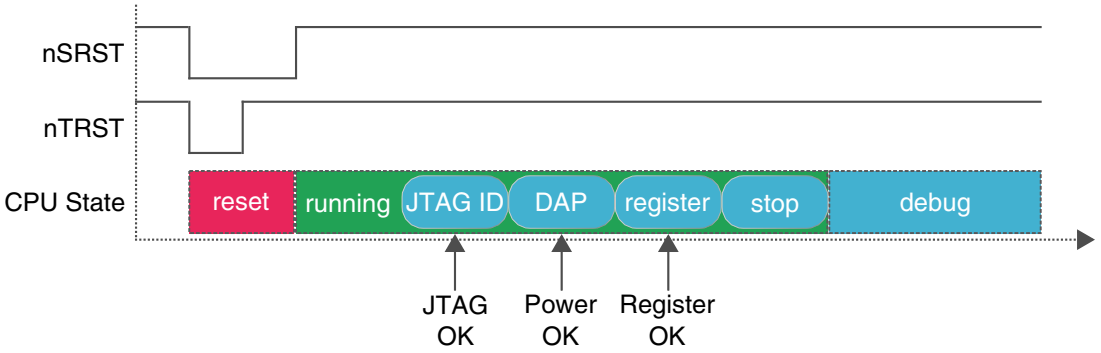
```
// Assumes that the following debug resources are not available in reset:
//   * JTAG
//   * Debug power domain
//   * Debug registers
RESet                          // Reset debugger configuration.
                               // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                 // Only connect to boot core
                               // (multicore systems)
SYStem.Option ResBreak OFF     // Early release of nSRST line
SYStem.Option WaitIDCODE ON    // Poll 1 second or less for JTAG
SYStem.Option WaitDAPPWR ON    // Poll 1 second or less for DAP power
SYStem.Option WaitDBGREG ON    // Poll 1 second or less for dbg registers

SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

All scenarios discussed so far assume that the debugger can poll some kind of status to determine when the core will be available for debug. However, it is possible that none of the three wait methods will succeed. Let's summarize what has been discussed so far:

- **SYStem.Option WaitIDCODE**: JTAG is not available in reset.
- **SYStem.Option WaitDAPPWR**: The debug power domain is not available in reset.
- **SYStem.Option WaitDBGREG**: Core debug registers are not available in reset.

A rather generic approach does not to try to check any status at all. Instead, the debugger releases the reset line early when **SYStem.Option ResBreak OFF** is used. The SoC can then execute an application (boot) code for some time before the debugger tries to bring the device in debug mode:



Consequences:

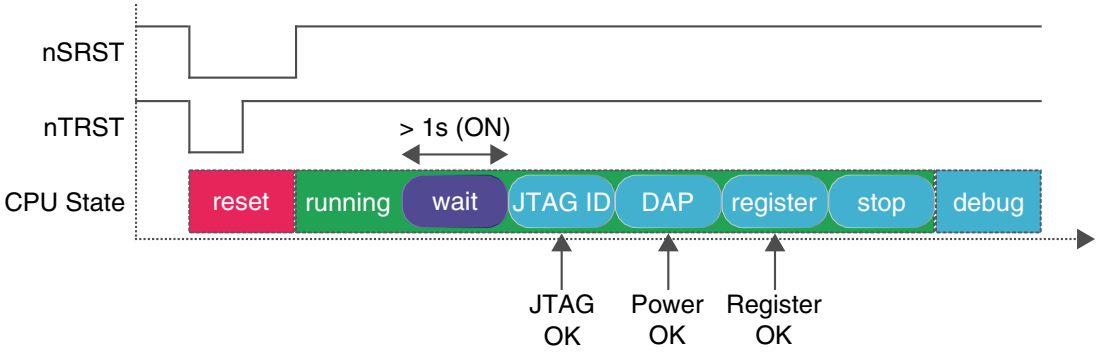
- The execution time of the CPU before it is brought to debug mode is not really known.
- The CPU might execute more code than expected or necessary for a debug connection.
- If boot timings of the CPU change, the wait time might not fit anymore, and e.g. PRACTICE start-up scripts (*.cmm) that have been working before might fail suddenly. Manual maintenance of the script is required.

Example:

```
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option ResBreak OFF      // Early release of reset line (nSRST)
SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

By using **SYStem.Option ResBreak OFF**, the debugger adds some additional wait time after reset has been released. This wait time might not be enough. In this case it is required to add an additional wait time with **SYStem.Option WaitReset**:



Consequences:

- The CPU might execute more code than expected or necessary for a debug connection.
- If boot timings of the CPU change, the wait time might not fit anymore, and e.g. PRACTICE start-up scripts (*.cmm) that have been working before might fail suddenly. Manual maintenance of the script is required.

Benefit:

- The additional wait time is known.

Example:

```
RESet                           // Reset debugger configuration.
                                // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.                  // Only connect to boot core
                                // (multicore systems)
SYStem.Option ResBreak OFF      // Early release of reset line (nSRST)
SYStem.Option WaitReset 30ms    // 30ms of additional wait time
                                // after reset has been deasserted
SYStem.Mode Up
```

©1989-2020 Lauterbach GmbH

Connect to Secondary Cores

In a multicore system, it might be required to connect to the secondary cores, too. Once the boot core is stopped in debug mode, the secondary cores might still be in a power down or reset state. In this case, a SoC-specific sequence can be executed to “kick” the secondary cores into active and debug state. Usually such a sequence is implemented in a custom start-up script.

Example:

```
// Assumption: The debugger has already connected to core 0 (boot core)
// In this example: There are four Arm cores in the SoC
DO <path>/<to>/<kick_cores_script>.cmm    // Kick secondary cores, e.g.
                                          // by custom script
SYStem.Mode Down                          // Detach single core session
                                          // to configure SMP session
CORE.ASSIGN 1. 2. 3. 4.                   // Assign all cores
SYStem.Mode Attach                        // Reattach to all cores
```

Troubleshooting

Situation	**SYStem.Mode Up** does not work.
Symptom	Debug port fail or target in reset.
Possible Cause	Not all debug resources are available during **SYStem.Mode Up** connection attempt.
Cross-check	Connect to the first core only in the SoC without issuing a reset using **SYStem.Mode Attach**.
Purpose	To evaluate if debug connection is possible at all.

Example:

```
// Evaluate if debug connection is possible at all
RESet                       // Reset debugger configuration.
                            // Does not toggle the reset line (nSRST)
SYStem.CPU <cpu>
CORE.ASSIGN 1.              // At first try core0 only
SYStem.Mode Attach
```

If even an attach operation does not work with your SoC, the device may intentionally not be enabled for debugging or there might be other issues, e.g. electrical issues. Please see the chapter “**Communication between Debugger and Processor cannot be established**”.

©1989-2020 Lauterbach GmbH

Summary

Let's sum up the wait methods discussed above. The strategy to connect to a core via **SYStem.Mode Up** SoC is as follows:

- Try to get a good understanding of the boot process on your SoC.
- Try to choose one or more wait methods in the following order:
  - **SYStem.Option WaitIDCODE**
  - **SYStem.Option WaitDAPPWR**
  - **SYStem.Option WaitDBGREG**
  - If those methods do not work, try to use **SYStem.Option ResBreak OFF**, probably in addition with **SYStem.Option WaitReset**.
  - If all options fail, try to connect to the first core via **SYStem.Mode Attach**.

©1989-2020 Lauterbach GmbH

Communication between Debugger and Processor cannot be established

Typically **SYStem.Mode Up** is the first command of a debug session for which communication with the target (e.g. evaluation board) is required. Error messages like “debug port fail” or “debug port time out” while executing this command may have the reasons below. “target processor in reset” is just a follow-up error message. Open the **AREA.view** window to see all error messages.

- The target has no power or the debug cable is not connected to the target. This results in the error message “target power fail”.
- You did not select the correct core type **SYStem.CPU** *<type>*.
- There is an issue with the JTAG interface. See **“ARM JTAG Interface Specifications”** (app_arm_jtag.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 of 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 cannot communicate with the debugger while in reset. Try **SYStem.Mode Attach** followed by **Break.direct** instead of **SYStem.Up** or use **SYStem.Option EnReset OFF**.
- The default frequency of the JTAG/SWD/cJTAG debug port is too high, especially if you emulate your core or if you use an FPGA-based target. In this case try **SYStem.JtagClock 50kHz** and optimize the speed when you got it working.
- The core has no power or is kept in reset.
- The core has no clock.
- The target needs a certain setup time after the reset release. Per default, **SYStem.Up** will try to catch the CPU at the reset vector, i.e. there is no time for any initialization. You can try to grant a certain setup time to the target using **SYStem.Option WaitIDCODE**, **SYStem.Option**

©1989-2020 Lauterbach GmbH

WaitDAPPWR, **SYStem.Option WaitDBGREG**, **SYStem.Option ResBreak** and/or **SYStem.Option WaitReset**. As an alternative, try to use **SYStem.Mode Attach**. Please see "Configure Debugger for SoC Specific Reset Behavior" for more details on these options.

- Your Core-base and/or CTI-base is not set. See **SYStem.CONFIG COREDEBUG Base** and **SYStem.CONFIG CTI Base**. Both base addresses are needed. The Core-base is needed for the communication and the CTI-base is needed for the start/stop control.
- There is a watchdog which needs to be deactivated.
- Your target needs special debugger settings. Check the directory ~~/demo/arm64 to see if there is a suitable script file *.cmm
- for your target.
- You might have several TAP controllers in a JTAG-chain. Example: The TAP of the DAP could be in a chain with other TAPs from other CPUs. In this case you have to check your pre- and post-bit configuration. See for example **SYStem.CONFIG IRPRE** or **SYStem.CONFIG DAPIRPRE**.

FAQ

Please refer to our Frequently Asked Questions page on the Lauterbach website.

Trace Extensions

There following types of trace extensions are available for Arm:

- **ARM-ETMv4:** an Embedded Trace Macrocell is integrated into the core. The Embedded Trace Macrocell provides program flow only plus trigger and filter features.

  Please note that in case of CoreSight ETMv4 you need to inform the debugger in the start-up script about the location of the trace control register and funnel configuration. See **SYStem.CONFIG ETM Base**, **SYStem.CONFIG FUNNEL Base**, **SYStem.CONFIG TPIU Base**, **SYStem.CONFIG ETMFUNNELPORT**. In case a HTM or ITM module is available and shall be used you need also settings for that.

  For detailed information about the usage of ARM ETMv4, please refer to the online-help books:

  - **"ARM-ETM Trace"** (trace_arm_etm.pdf)
  - **"ARM-ETM Programming Dialog"** (trace_arm_etm_dialog.pdf)
  - **"ARM-ETM Training"** (training_arm_etm.pdf)

©1989-2020 Lauterbach GmbH

Quick Start for Multicore Debugging

This chapter will give you a quick start regarding Multicore Processing:

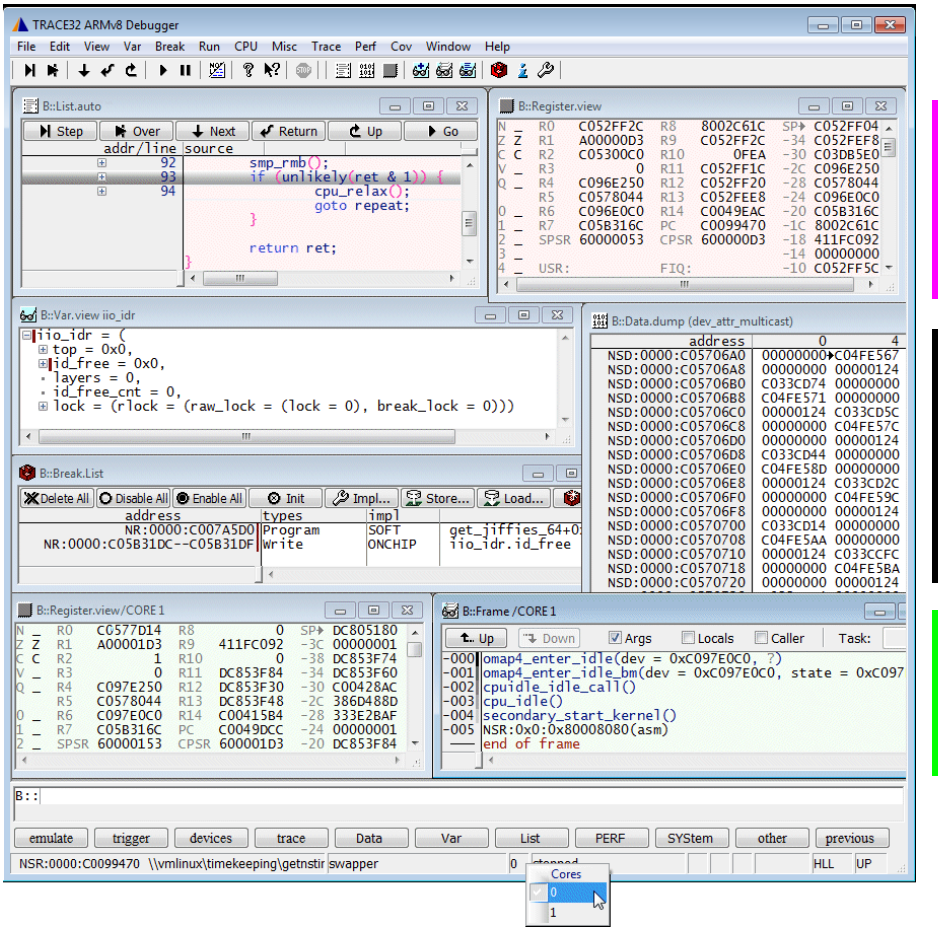
- **SMP Debugging - Quick Start** (Symmetric Multiprocessing)
- **AMP Debugging - Quick Start** (Asymmetric Multiprocessing)

Please see also the **Demo and Start-up Scripts** chapter for ready-to-run PRACTICE (*.cmm) start-up scripts.

©1989-2020 Lauterbach GmbH

1. How to Debug a System with Multiple Identical Cores

A multicore system used for Symmetric Multiprocessing has two or more cores that are identical or have at least a compatible instruction set. This means that an application might switch between the cores. To debug an SMP system you start only one TRACE32 PowerView GUI:



Core-specific information (here: Currently selected core)

Information common for all cores

Core-specific information (here: Core 1)

All cores can be accessed using this one GUI. Core specific information can be displayed in separate windows. The cores will be started, stepped (HLL step) and halted together. The assembler step is **not** synchronized and will be done **independently** for each core. TRACE32 takes care that software and on-chip breakpoints will have an effect on whatever core the task will run.

SYStem.Option settings and the selected JTAG clock affect all cores. For the start-up the first TRACE32 instance gets control over the reset signals. **SYStem.CONFIG Slave ON** may only be used if none of the SMP cores shall control the reset lines and initialize the JTAG interface.

Devices that have a Cross-Trigger-Matrix (CTM) implemented allow **Symmetric Multiprocessing (SMP)** with low start/stop latencies. If there is no CTM implemented, SMP debugging is still possible but the debugger must continuously poll the state of the cores and synchronize the state of the other cores which causes much higher start/stop latencies.

©1989-2020 Lauterbach GmbH

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 in advance on which core the task you are interested in will be executed.

2. Set up the SMP Debug Scenario

The selection of the proper SMP SoC causes the debugger to connect to all assigned SMP-able cores on start-up (e.g. by **SYStem.Mode Up**). To select an SMP SoC use **SYStem.CPU**.

If a predefined SoC is available, the following settings could be skipped, as they are already set by the SoC selection. Having selected a SoC, you can check this settings in **SYStem.CONFIG**. It is recommend to check if your SoC is already available in the CPU selection.

If you have no predefined SMP SoC but an SMP-able core setup you need to specify the number of cores you intend to SMP-debug yourself by **SYStem.CONFIG CoreNumber** *<number>*. You need then to select the cores (of all cores) you want to debug using **CORE.ASSIGN**, e.g.:

```
SYStem.CPU CORTEXA53                          ; Select CPU or chip
SYStem.CONFIG CoreNumber 4.                  ; Total number of cores
CORE.ASSIGN 1. 2.                             ; Assign SMP cores
```

In this example you have 2 out of 4 cores grouped into one SMP system. As a next step you need to specify how each core can be accessed in your SMP setup. For known SoCs this settings might already be set. Check the **SYStem.CONFIG /COmponents** window.

Example: If you have several cores connected to one common DAP you have to specify multiple core base and CTI base addresses:

```
; Base and CTI addresses, for cores 1 & 2
SYStem.CONFIG COREDEBUG Base DAP:0x80010000 DAP:0x80110000
SYStem.CONFIG CTI       Base DAP:0x80020000 DAP:0x80120000
```

Some setups might also require to set up PRE and POST bits, e.g. if your cores are in a JTAG daisy chain:

```
SYStem.CONFIG DAPIRPRE  5.        ; One trailing IR in the chain
SYStem.CONFIG DAPIRPOST 10.       ; 10 = 3 + 7 (add precedent IR)
SYStem.CONFIG DAPDRPRE  1.        ; One trailing bypass DR in the chain
SYStem.CONFIG DAPDRPOST 2.        ; 2 = 1 + 1 (all precedent bypass DR)
```

In this example the chain would look like this:

IR-Path: TDI ---> [TAP, IR=3] ---> [TAP, IR=7] ---> [DAP-TAP, IR=4] ---> [TAP, IR=5] ---> TDO

DR-Path: TDI ---> [TAP, DR=1] ---> [TAP, DR=1] ---> [DAP-TAP, DR=32] ---> [TAP, DR=1] ---> TDO

©1989-2020 Lauterbach GmbH

3. Enter Debug Mode

```
SYStem.Up
```

This will bring all cores into debug mode that have been assigned to the SMP system using the **CORE.ASSIGN** command. The default case assumes that the debug interface is accessible while the CPU is held in reset. If this is not the case, **SYStem.Mode Up** may fail with an error. In this case try to attach to the CPU and to stop it:

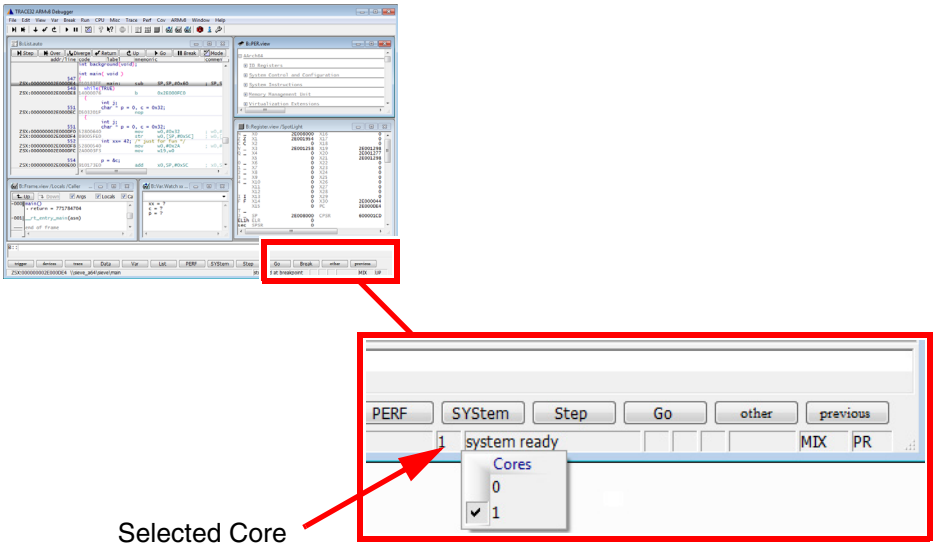
```
SYStem.Mode Attach                  ; Attach without reset
Break.direct                        ; Enter debug mode
```

4. Switch Debug View between Cores

The command **CORE.select** allows to switch between cores (also visible and changeable via the state line). Commands may also offer the **/CORE** *<core>* option, e.g.:

```
CORE.select 0.                      ; select core 0
Register.view                       ; view registers of current core 0
Register.view /CORE 1.              ; view registers of core 1
```

The number of the currently selected core is shown in the state line at the bottom of the main window. You can use this to switch the GUIs perspective to the other cores when you right-click on the core number there.



5. Write a Start-up Script Summary

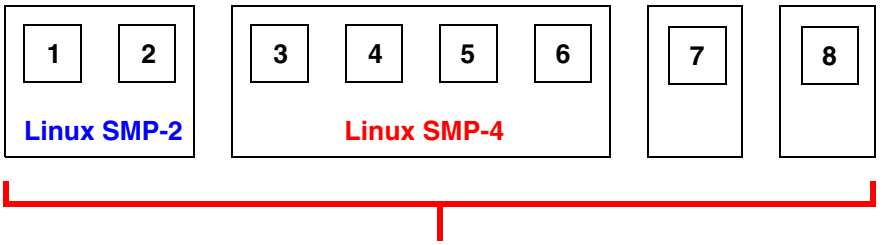
The SMP setup sequence can be written altogether into a *.cmm setup script like *<smp_setup>*.cmm. The setup can be reproduced by calling this script:

```
CD.DO <smp_setup>.cmm
```

©1989-2020 Lauterbach GmbH

1. How to Debug a System with Multiple Heterogenous Cores

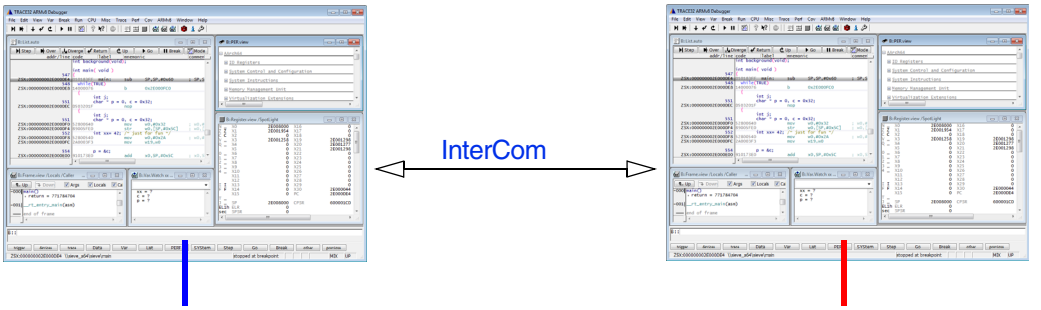
A multicore system used for Asymmetric Multiprocessing (AMP) has specialized cores which are used for specific tasks. An AMP system may consist of independent single cores or of independent SMP-systems (subsystems). Independent means that e.g. tasks cannot be assigned to other AMP cores/subsystems in an arbitrary fashion:



AMP system with two independent SMP systems and two single cores

In this case, tasks of the Linux SMP-2 system could not be assigned to the Linux SMP-4 system or the single cores. However, tasks can be assigned within the SMP system, in this case core 1 and core 2.

To debug such a system, you need to open a separate TRACE32 graphical user interface (GUI), one for each core or each subsystem. For simplicity, only two GUIs for two SMP systems are shown here:



SMP System GUI (**Linux SMP-2**)

SMP System GUI (**Linux SMP-4**)

Each GUI will debug one subsystem. The GUIs are able to communicate using the **InterCom** command.

2. Starting the TRACE32 PowerView GUIs

To set up an AMP multicore debugging scenario, multiple TRACE32 instances (GUIs) need to be started. Please make sure the **"File config.t32"** (installation.pdf) adds at least:

```
CORE=<integer>                                  ; e.g. CORE=1

IC=NETASSIST                                    ; enable InterCom
PORT=<port_number>                              ; e.g. PORT=20000
```

©1989-2020 Lauterbach GmbH

For details on CORE=, please refer to **"Config file settings for single device solution"** in ICD Debugger User's Guide, page 27 (debugger_user.pdf).

For details on **InterCom**, please refer to **"Config file settings for both single and multi device solutions"** in ICD Debugger User's Guide, page 29 (debugger_user.pdf).

3. Master-Slave Concept

An AMP system may share the reset line. Only one TRACE32 instance is allowed to control the reset line to avoid unwanted resets by other TRACE32 instances. The TRACE32 instance that controls the reset line is called "master" and all the other TRACE32 instances are referred to as "slaves".

For AMP systems, we recommend to write a start-up script, which is intended to run on the "master" instance controlling the "slave" instances via **InterCom** commands.

Hint: Setting up user-defined commands with **ON CMD** improves readability, e.g.:

```
; startup_script.cmm running on GUI0:
&addressGUI2="127.0.0.1:20002"                  ; address and port of
                                                ; GUI2
ON CMD CORE2 GOSUB                              ; define command "CORE2"
(
     LOCAL &params
     ENTRY %Line &params
     InterCom.execute &addressGUI2 &params      ; execute command on
                                                ; remote GUI
     RETURN
)
CORE2 PRINT "executed on core 2"                ; use the user-defined
                                                ; command: text will be
                                                ; printed on GUI2
```

4. Setting up the Multicore Environment

After starting the TRACE32 PowerView GUIs, each instance assumes to be connected to a separate chip/subsystem by default. Mounting the cores into the same chip makes TRACE32 aware of the resources to be shared between the cores. This is especially important for multi-core synchronization and shared resources like the on- and off-chip trace.

Before bringing the system up, use the **SYStem.CONFIG.CORE** command on each GUI to mount all cores into one chip, e.g.:

```
CORE0 SYStem.CONFIG CORE 1. 1.                  ; GUIO: core 0 in chip 1
CORE1 SYStem.CONFIG CORE 2. 1.                  ; GUI1: core 1 in chip 1
CORE2 SYStem.CONFIG CORE 3. 1.                  ; GUI2: core 2 in chip 1
```

You need to set up each core in each GUI separately. For each Core/GUI this it no different from setting up a single core setup. An example setup for a single ARMv8 core can be found in the **Quick Start of the JTAG Debugger chapter**.

©1989-2020 Lauterbach GmbH

5. Synchronized Go / Step / Break

The **SYnch** command allows for start stop synchronization between multiple GUIs. For details please refer to **"Start Stop Synchronisation"** in ICD Debugger User's Guide, page 41 (debugger_user.pdf).

6. Write a Start-up Script Summary

The AMP setup sequence can be written altogether into a *.cmm setup script like *<amp_setup>*.cmm. The setup can be reproduced by calling that script:

```
CD.DO <amp_setup>.cmm
```

©1989-2020 Lauterbach GmbH

AArch Mode Support

Cores based on the ARMv8 architecture may switch operation between 64-bit and 32-bit execution. For this the ARMv8 architecture defines two modes:

AArch64	Supports execution of the 64-bit instruction set A64. The instruction length is fixed to 32-bit. The operand can be 64-bit wide (e.g. General Purpose Registers).
AArch32	Supports execution of the 32-bit instruction sets A32 (ARM) and T32 (Thumb). Acts as a compatibility mode to run 32-bit applications, e.g. legacy code from the ARMv7 or other 32-bit ARM cores.

For a detailed description, refer to the ARMv8 and / or ARMv7 documents of the manufacturer.

AArch64 and AArch32 Debugging

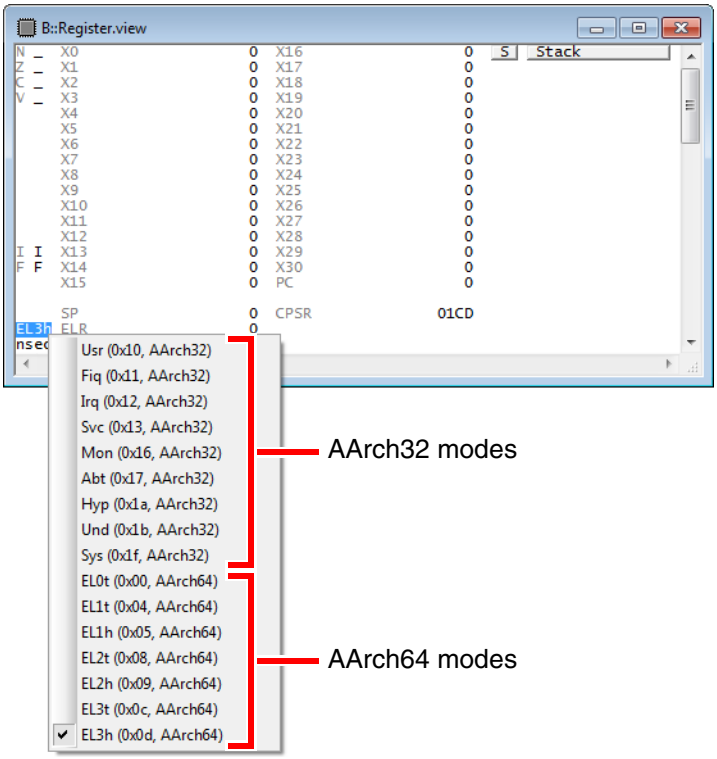
When the debugger establishes a debug connection to the target (e.g. **SYStem.Mode Up**, **SYStem.Mode Attach**) it will automatically determine which mode the core is currently using. The AArch mode will mainly affect:

- **Register Window**: In the AArch64 mode the **Register.view** window will be displayed in a new format. In the AArch32 mode the register window layout from precedent ARM architectures, like the ARMv7 architecture, will be used. You can watch both layouts simultaneously by enlarging the register window. Some registers of the inactive AArch mode might not be accessible.
- **List Window**: The instruction opcodes in the **List.auto** window will be disassembled according to the current **DisMode** setting. It is recommended to use **SYStem.Option DisMode AUTO** and to let the debugger decide which disassembly mode is used. To choose a custom disassembly mode use **SYStem.Option DisMode** or use an appropriate **access class**.
- **Software Breakpoints**: Software Breakpoint instructions differ in AArch64 from the ones in AArch32. To ensure correct use of Software Breakpoints you should set the correct **DisMode** using **SYStem.Option DisMode**. It is recommended to use **SYStem.Option DisMode AUTO**.
- **Co-Processor (C15/C14) and System Register (SPR) access:** In AArch64 mode, System Registers are used instead of the AArch32 Co-Processor registers. Both register types are related but the accessibility depends on the CPU mode and is handled differently in most cases. Most CP-Registers are architecturally mapped to SPRs.
- **Secure mode handling:** In AArch64 mode, the influence of the secure bit is different from the AArch32 mode. In AArch32 the secure mode will decide which of the banked secure/non-secure registers is currently available. This does not apply for most registers in AArch64 mode. The accessibility of registers is mainly determined by the current exception level of the CPU.
- **FPU.view Window**: The FPU/VPU unit has been reworked in ARMv8. The layout of the **FPU.view** window changes according to the AArch mode that was detected by the debugger. You can watch both layouts simultaneously by enlarging the **FPU** window.

©1989-2020 Lauterbach GmbH

AArch Mode Switch by Debugger

The AArch mode of the CPU can be switched in the **Register.view** window.



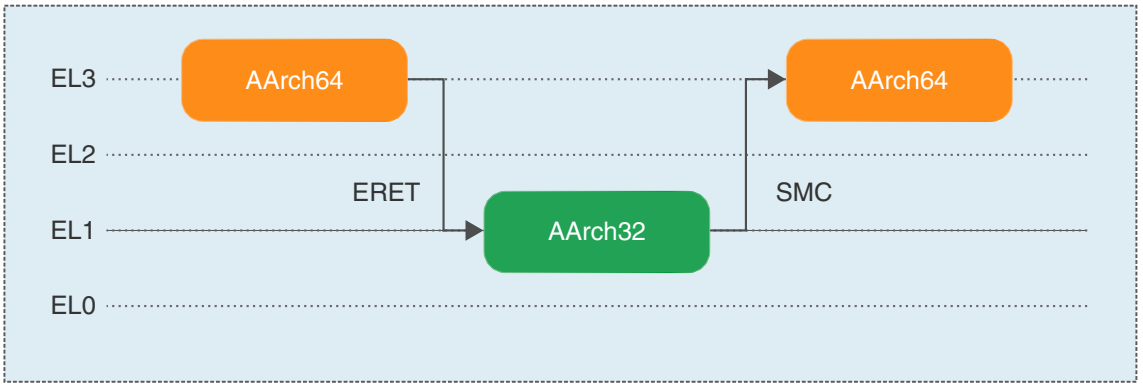
Not every transition is possible or allowed. The debugger will print an error message an a reason to **AREA.view** in an error case. Forbidden transitions are:

- Any switch from an AArch64/AArch32 secure mode to AArch64/AArch32 EL2/hyp (ARMv8.0-A - ARMv8.3-A). Please note that this may be a legal transition since ARMv8.4-A.
- A switch from AArch64 EL3 mode to AArch32 mon mode
- A switch from AArch32 mon mode to AArch64 EL3
- A switch to an AArch64 EL when the CPU is in AArch32 only mode
- A switch to an AArch32 mode when the corresponding EL does not support AArch32

©1989-2020 Lauterbach GmbH

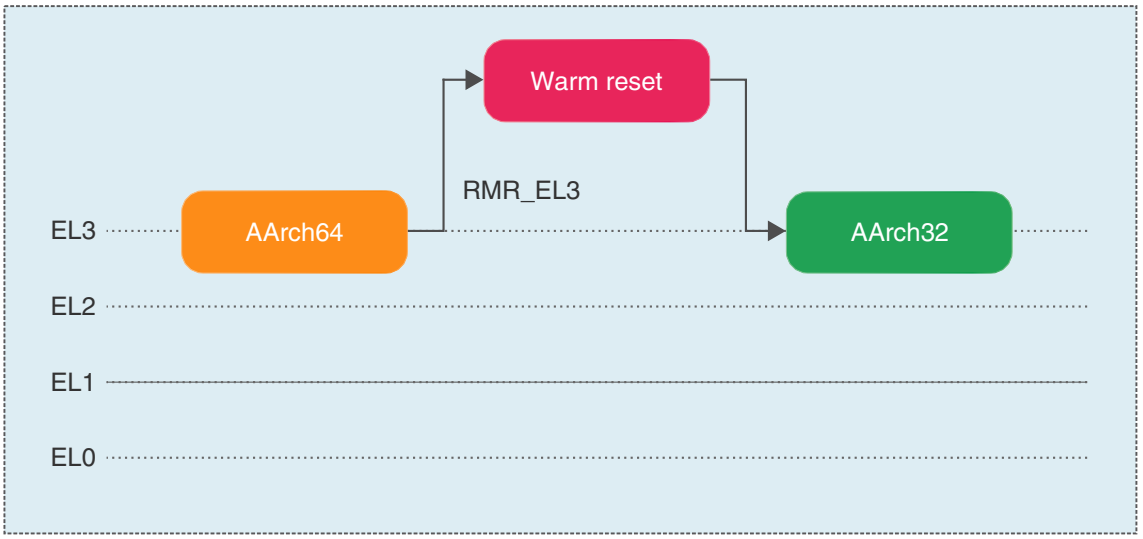
AArch Mode Switch by Application

For an application there are two possibilities to switch the AArch mode. An application might configure the lower Exception Levels (ELx) an cause an exception return (ERET) to switch to a lower EL. It is also possible for an application to go to a higher level by causing an exception call, e.g. a secure monitor call (SMC).



Example transition from AArch64 EL3 to AArch32 EL1 using an exception return (ERET) and transition back to AArch64 EL3 using a secure monitor call (SMC).

The above approach does not allow to switch the AArch mode of EL3. A special mechanism is needed. The RMR_ELx registers (if implemented) can be used to cause a reset and define a new AArch mode.



Example transition from AArch64 EL3 to AArch32 EL3 using the RMR_EL3 register.

The Vector catch debug events (**TrOnchip.Set**) can be configured to stop the core on such a reset event.

The debugger keeps track of the application mode switches. However, the debugger cannot know that the target switched the AArch mode while the application is running. The debugger detects a switch the next time the target enters debug mode. A change might happen every time the target has the opportunity to execute code:

- Assembler single step or HLL single step
- Go - Break
- Go - Self triggered halt (Breakpoints, Exceptions, catch events, etc.)

©1989-2020 Lauterbach GmbH

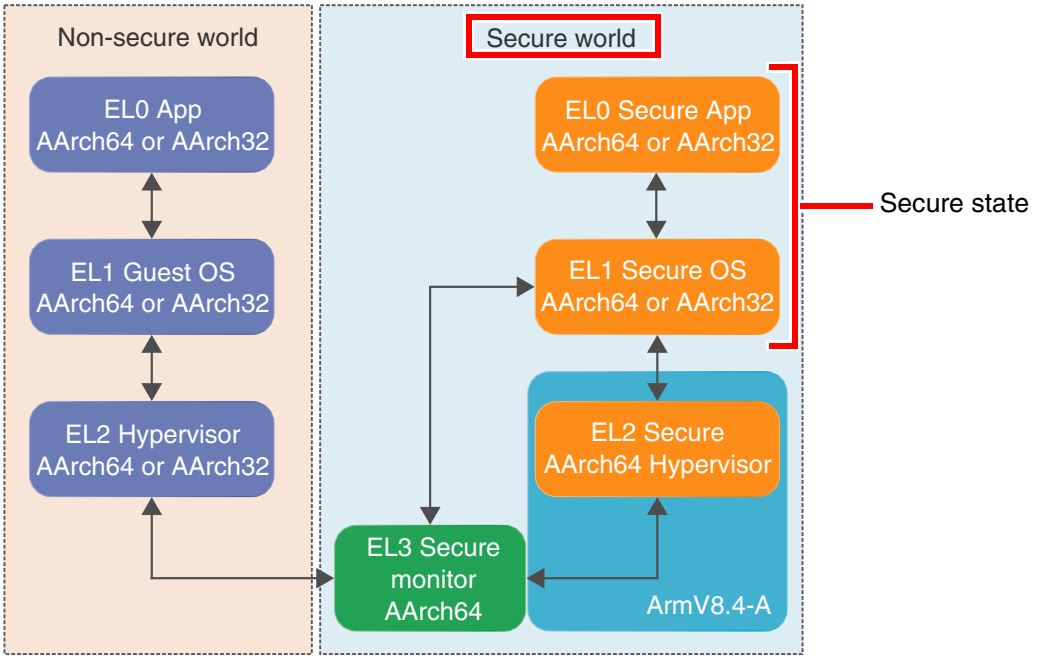
TrustZone Technology

Some processors might integrate ARM's TrustZone technology, a hardware security extension, to facilitate the development of secure applications.

The TrustZone technology 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.

AArch64 Secure Model

For switching purposes, TrustZone provides a secure 'EL3' mode. Reset enters the secure world:



When EL3 is operating in AArch64 mode, lower exception modes may either operate in AArch64 or AArch32 mode. The availability of these modes for each exception level depends on the specific implementation.

Only when the core is in the secure world, core and debugger can access the secure memory. In AArch64 mode, there are no secure/non-secure banked registers. Instead, the accessibility of the registers depends only on the exception level the CPU operates in. A few registers are only available in secure mode, e.g. secure timer registers.

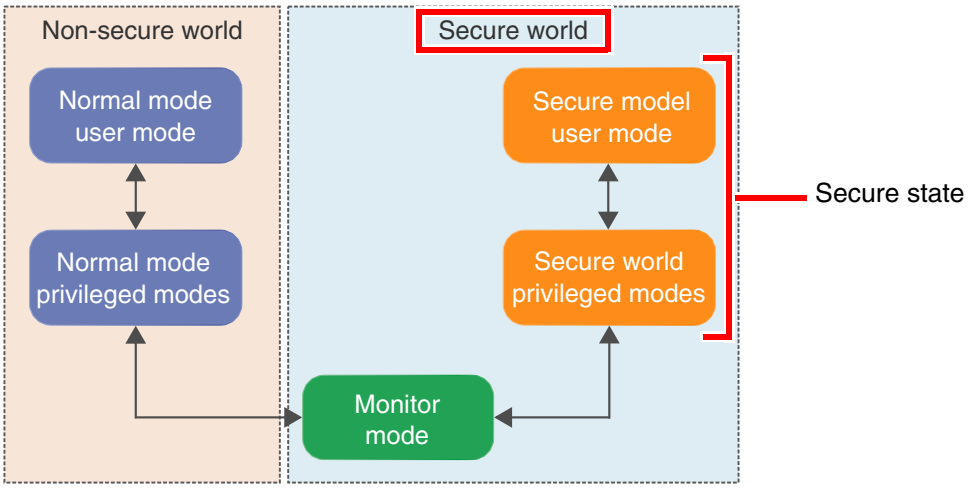
Armv8.4-A Secure Hypervisor Extension

Armv8.4-A adds a secure Hypervisor to the security model. This extension is only available in AArch64. This means there is no secure AArch32 Hypervisor.

©1989-2020 Lauterbach GmbH

AArch32 Secure Model

For switching purposes, TrustZone provides a secure 'monitor' mode. Reset enters the secure world:



When the monitor mode operates in AArch32 mode, all lower exception levels / modes must also operate in AArch32 mode.

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 cannot be entered in a mode where debugging is not allowed. Breakpoints will not work there. **Break** or **SYStem.Mode Up** will work at the moment a mode is entered where debugging is allowed.

Checking Debug Permission

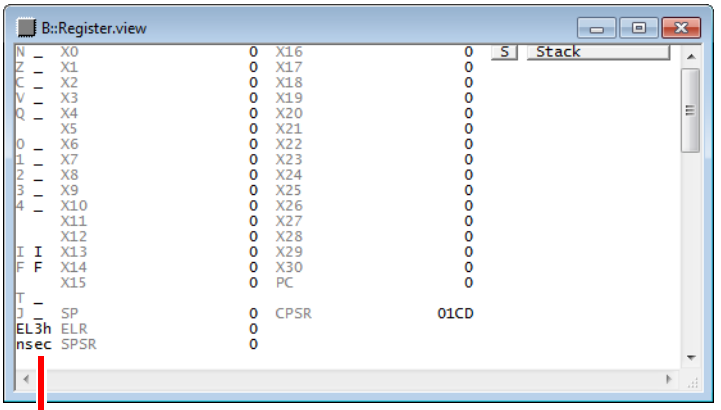
The DBGAUTHSTAT (Debug Authentication Status Register), bit [5:4] shows the signal level of SPIDEN.

In the SDER/SDER_EL3 (Secure Debug Enable Register) you can see the SUIDEN flag assuming you are in the secure state which allows reading the SDER register.

©1989-2020 Lauterbach GmbH

Checking Secure State

In the peripheral file, the SCR register bit 0 (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.



Current secure state and mode

In AArch64 mode, for most SPR registers the access does not depend on the security state.

In AArch32 monitor mode, which is also indicated in the **Register.view** window, the core is always in a secure state independent of the NS bit (non-secure bit) described above. From AArch32 monitor mode, you can access the secure CP15 registers if “NS=secure” and the non-secure CP15 registers if “NS=non-secure”.

To check the secure state, e.g. in a PRACTICE script (*.cmm) or via the TRACE32 command line, use:

```
PRINT Register(NS)            ; PRINT current secure mode bit to AREA.view
&secureMode=Register(NS)      ; Read current secure mode bit into a macro
```

Changing the Secure State from within TRACE32

From the TRACE32 PowerView GUI, you can switch between secure mode (NS=0) and non-secure mode (NS=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 registers 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.

©1989-2020 Lauterbach GmbH

AArch64 System Registers Access

The secure mode does not affect the access to most system registers (SPR). Mainly the exception level (EL) of the CPU is relevant. Some registers are only accessible in secure mode, e.g. secure timer registers. See also **System Registers (AArch64 mode)**.

You can force the SPR access in another mode by using the access class “MSPR:” for EL3, “HSPR:” for EL2/hypervisor, “ZSPR:” for secure and “NSPR:” for non-secure respectively.

AArch32 Coprocessor Registers Access

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/EL3 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 implemented’.

Breakpoints and Vector Catch Register

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

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

Breakpoints and Secure Modes

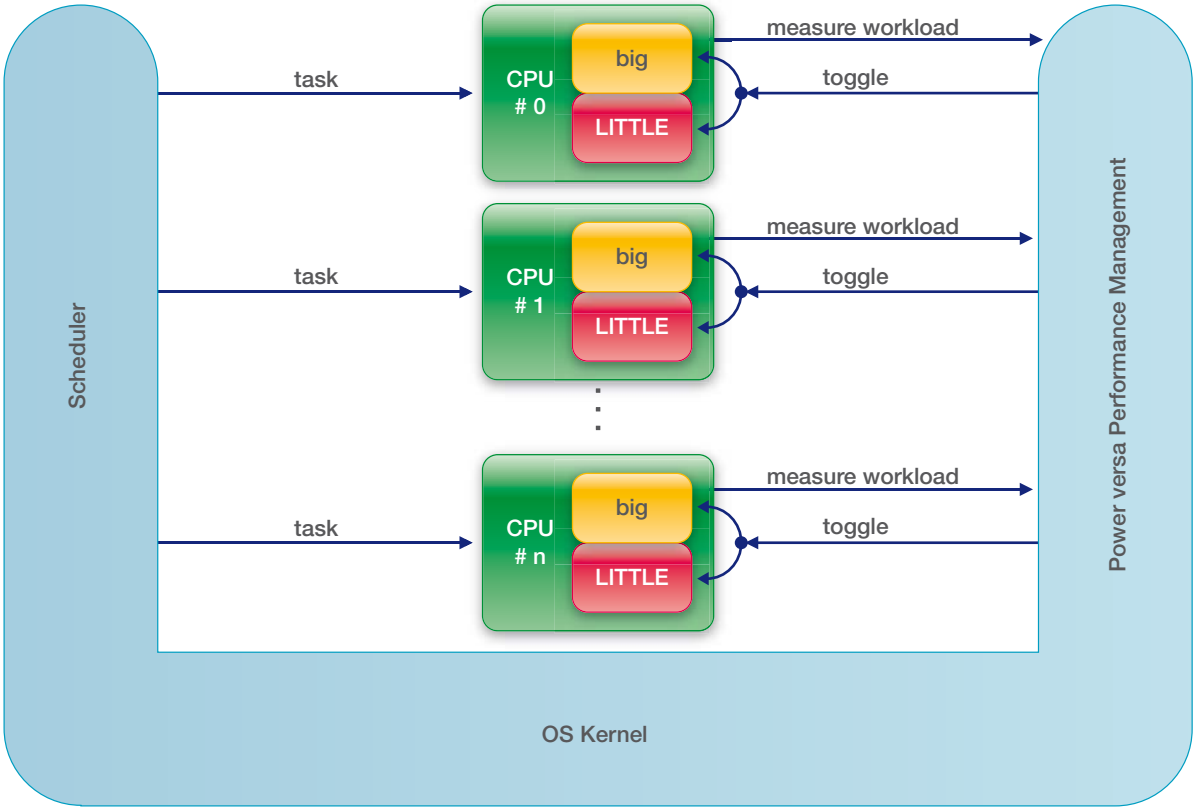
The security concept of the ARMv8 architecture allows to specify breakpoints that cause a halt event only for a certain secure mode (EL3/monitor, secure, non-secure, hypervisor).

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

©1989-2020 Lauterbach GmbH

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 transition 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-A57/A72/A73/A75/A76 (as 'big') and Cortex-A35/A53/A55 (as 'LITTLE').

Debugger Setup

Example for a symmetric big.LITTLE configuration (2 Cortex-A57, 2 Cortex-A53):

```
SYStem.CPU CORTEXA57A53
SYStem.CONFIG CoreNumber 4.
CORE.ASSIGN BIGLITTLE 1. 2. 3. 4.
SYStem.CONFIG.COREDEBUG.Base <CA57_1> <CA53_2> <CA57_3> <CA53_4>
SYStem.CONFIG.CTI.Base <CA57CTI_1> <CA53CTI_2> <CA57CTI_3> <CA53CTI_4>
```

©1989-2020 Lauterbach GmbH

Example for a non-symmetric big.LITTLE configuration (1 Cortex-A57, 2 Cortex-A53):

```
// NONE is a dummy core as partner for the 2nd little core
SYStem.CPU CORTEXA57A53
SYStem.CONFIG CoreNumber 4.
CORE.ASSIGN BIGLITTLE 1. 2. NONE 4.
SYStem.CONFIG.COREDEBUG.Base <CA57_1> <CA53_2> <dummy_3> <CA53_4>
SYStem.CONFIG.CTI.Base <CA57CTI_1> <CA53CTI_2> <CA57CTI_3> <CA53CTI_4>
```

Consequence for Debugging

Using ‘BIGLITTLE’ for the core assignments will have the following effects on the debug session:

- The shown core numbers are extended by ‘b’ = ‘big’ or ‘l’ = ‘LITTLE’.
- The core status (active or powered down) can be checked with **TargetSystem.state** 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 (e.g. Cortex-A57 or Cortex-A53).

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 multi-core system, but with mixed core types.

Therefore for the setup, we need **SYStem.CPU CORTEXA57A53**, but we use **CORE.ASSIGN** instead of **CORE.ASSIGN BIGLITTLE**.

Example for a symmetric big.LITTLE MP configuration (2 Cortex-A57, 2 Cortex-A53):

```
SYStem.CPU CORTEXA57A53
SYStem.CONFIG CoreNumber 4.
CORE.ASSIGN 1. 2. 3. 4.
SYStem.CONFIG.COREDEBUG.Base <CA57_1> <CA53_2> <CA57_3> <CA53_4>
SYStem.CONFIG.CTI.Base <CA57CTI_1> <CA53CTI_2> <CA57CTI_3> <CA53CTI_4>
```

©1989-2020 Lauterbach GmbH

Breakpoints

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is temporarily patched by a breakpoint code. 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. Those CPU resources only allow to set single address instruction breakpoints.

If an instruction range shall be covered, you need to invoke the ETM, see **ETM.StoppingBreakPoints** in trace_arm_etm.pdf.

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 watch points (WP).

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 Value 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.

Family	On-chip breakpoints	Instruction breakpoints	Read/Write breakpoint	Data Value breakpoints
Cortex-A3x **Cortex-A5x** **Cortex-A7x**	2 … 16 instruction 2 … 16 read/write	2 … 16 single address	2 … 16 single address	—

©1989-2020 Lauterbach GmbH

Example for Standard Breakpoints

Assume you have a target with

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

The command to set up 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 with “myFunc” in RAM.

```
Break.Set 0x100000 /Program          ; Software BP on RAM address

Break.Set myFunc /Program            ; Software BP on symbol in RAM

Break.Set 0x100 /Program             ; On-chip BP on flash address
```

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

```
Break.Set 0x100000 /Program          ; Software BP on address

Break.Set myFunc /Program            ; Software BP on symbol

Break.Set 0x108000 /Write            ; On-chip write breakpoint
                                     ; on address 0x108000
```

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 on specific addresses

```
Break.Set 0x108000 /Write            ; On-chip write breakpoint 1

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

©1989-2020 Lauterbach GmbH

Secure, Non-Secure, Hypervisor Breakpoints

TRACE32 will set any breakpoint to work in any secure and non-secure mode. As of build 59757, 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/hypervisor modes.

Enable and Use Secure, Non-Secure and Hypervisor Breakpoints

First enable the symbol management for Arm zones with the **SYStem.Option ZoneSPACES** command:

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

Second, configure breakpoints to match in different zones:

```
Break.CONFIG.MatchZone ON                  ; Match in different 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

Data.LOAD.ELF armf M:     ; Load application, symbols for EL3 (AArch64)
```

Please refer to the **SYStem.Option.ZoneSPACES** and **Break.CONFIG.MatchZone** command for additional code loading examples.

Now breakpoints can be used 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:, H:
                          ; or M: 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/hypervisor or EL3 type.
```

©1989-2020 Lauterbach GmbH

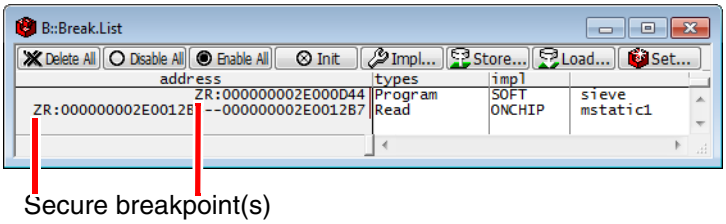
Example 1: Load secure application and set breakpoints

```
SYStem.Option.ZoneSPACES ON    ; Enable symbol management
Break.CONFIG.MatchZone         ; Enable bp match in different zones

// Load demo application and tell TRACE32 that it is secure (Z:)
Data.LOAD.ELF ~~/demo/arm64/compiler/arm/sieve_a64.elf 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 and breakpoints are configured to match in different secure zones. 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.

©1989-2020 Lauterbach GmbH

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 a secure type. This is shown by the “Z:” access class that is set at the beginning of the breakpoint addresses:



Set Breakpoints and Enforce Secure Mode

TRACE32 allows the user to specify whether a breakpoint should be set for secure, non-secure, hypervisor or EL3 (AArch64) 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()

Break.Set M:main            ; Enforce EL3 (AArch64) 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

Var.Break.Set (M:struct1) ; Enforce EL3 (AArch64) read/write breakpoint
```

For 64-bit modes “Z:”, “N:”, “H:” or “M:” determine in which exception level the CPU shall stop:

BP access class	EL3 sec	EL3 nsec	EL2	EL1/EL0 sec	EL1/EL0 nsec
M: (secure)	halt	halt	no halt	no halt	no halt
Z: (secure)	no halt	no halt	no halt	halt	no halt
H: (hypervisor)	no halt	no halt	halt	no halt	no halt
N: (non-secure)	no halt	no halt	no halt	no halt	halt

©1989-2020 Lauterbach GmbH

For 32-bit modes “Z:”. “N:” or “H:” determine in which mode the CPU shall stop:

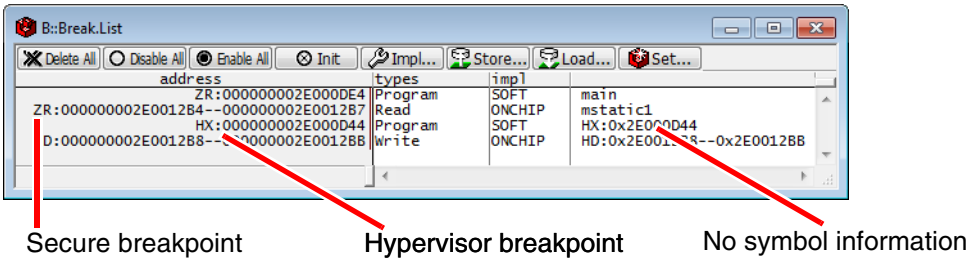
BP access class	mon sec	mon nsec	hyp	svc/usr sec	svc/usr nsec
Z: (secure)	halt	halt	no halt	halt	no halt
H: (hypervisor)	no halt	no halt	halt	no halt	no halt
N: (non-secure)	no halt	no halt	no halt	no halt	halt

Example 2: Load secure application and set hypervisor breakpoint.

```
SYStem.Option.ZoneSPACES ON     ; Enable symbol management
Break.CONFIG.MatchZone          ; Enable bp match in different zones

// Load demo application and tell TRACE32 by 'Z:' that it is secure
Data.LOAD.ELF ~~/demo/arm64/compiler/arm/sieve_a64.elf Z:

Break.Set main                  ; Auto configured secure breakpoint

// 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.

Then four breakpoints are set. Two do not have any access class specified. TRACE32 will use the symbol information to make them secure breakpoints. Two breakpoints are defined as hypervisor breakpoints with “H:”, i.e. the symbol information is explicitly overwritten. **Break.List** now shows a mixed breakpoint setup:



Secure breakpoint

Hypervisor breakpoint

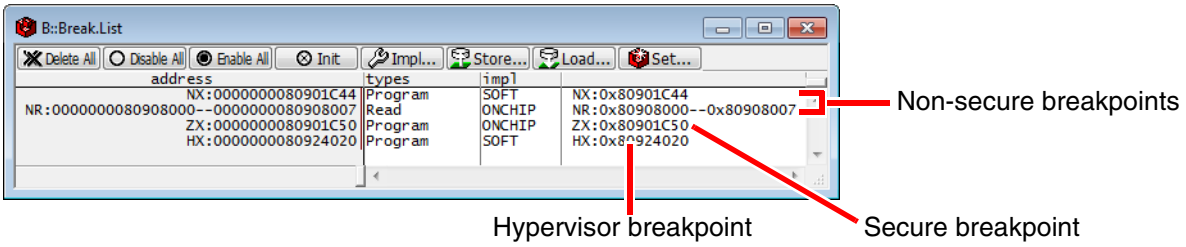
No symbol information

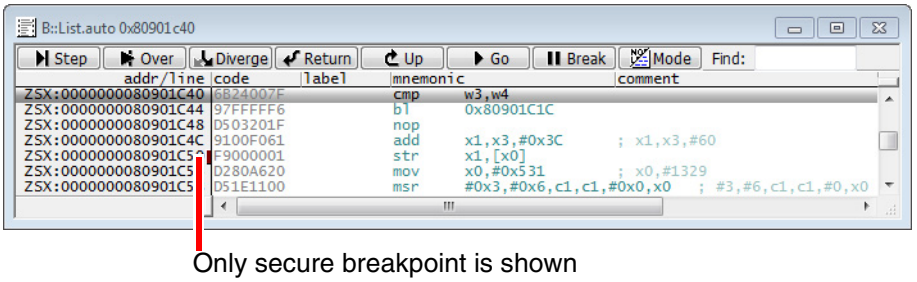
NOTE:	If a breakpoint is explicitly set in another mode, there might be no symbol information loaded for this mode. This means **Break.List** can only display the address of the breakpoint but not the corresponding symbol.

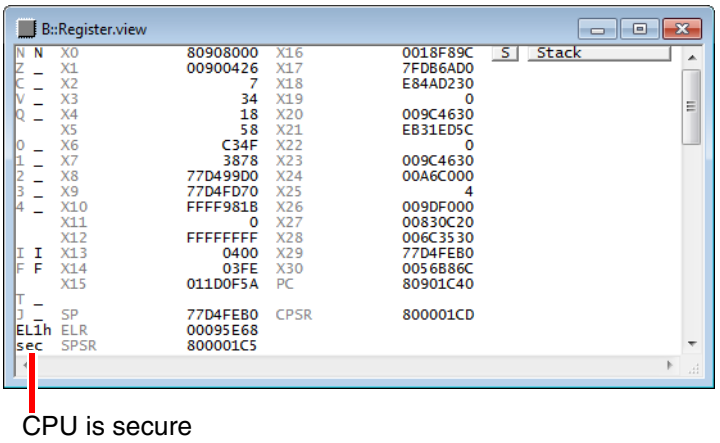
©1989-2020 Lauterbach GmbH

Summary of Breakpoint Configuration

TRACE32 can show 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. The window **Register.view** 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 on-chip breakpoints configuration will be placed in the breakpoint/watchpoint registers of the ARMv8 CPU. 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/hypervisor mode.

©1989-2020 Lauterbach GmbH

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).

©1989-2020 Lauterbach GmbH

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 Glossary"** (glossary.pdf).

In this section:

- Description of the Individual Access Classes
- Combinations of Access Classes
- How to Create Valid Access Class Combinations
- Access Class Expansion by TRACE32

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.
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

©1989-2020 Lauterbach GmbH

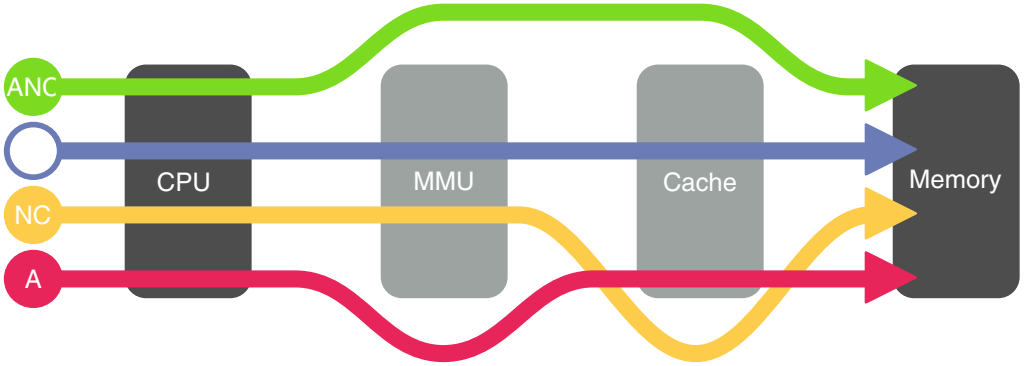
Access Class	Description
DAP, DAP2, AHB, AHB2, APB, APB2, AXI, AXI2	Memory access via bus masters, so named Memory Access Ports (MEM-AP), provided by a Debug Access Port (DAP). The DAP is a CoreSight component mandatory on Cortex based devices. Which bus master (MEM-AP) is used by which access class (e.g. AHB) is defined by assigning a MEM-AP number to the access class: SYStem.CONFIG DEBUGACCESSPORT *<mem_ap#>* -> “DAP” SYStem.CONFIG AHBACCESSPORT *<mem_ap#>* -> “AHB” SYStem.CONFIG APBACCESSPORT *<mem_ap#>* -> “APB” SYStem.CONFIG AXIACCESSPORT *<mem_ap#>* -> “AXI” You should assign the memory access port connected to an AHB (AHB MEM-AP) to “AHB” access class, APB MEM-AP to “APB” access class and AXI MEM-AP to “AXI” access class. “DAP” should get the memory access port where the debug register can be found which typically is an APB MEM-AP (AHB MEM-AP in case of a Cortex-M). There is a second set of access classes (DAP2, AHB2, APB2, AXI2) and configuration commands (e.g. SYStem.CONFIG DAP2AHBACCESSPORT *<mem_ap#>*) available in case there are two DAPs which needs to be controlled by the debugger.
E	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)
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.
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-A 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

©1989-2020 Lauterbach GmbH

Access Class	Description
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

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
```

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

How to Create Valid Access Class Combinations

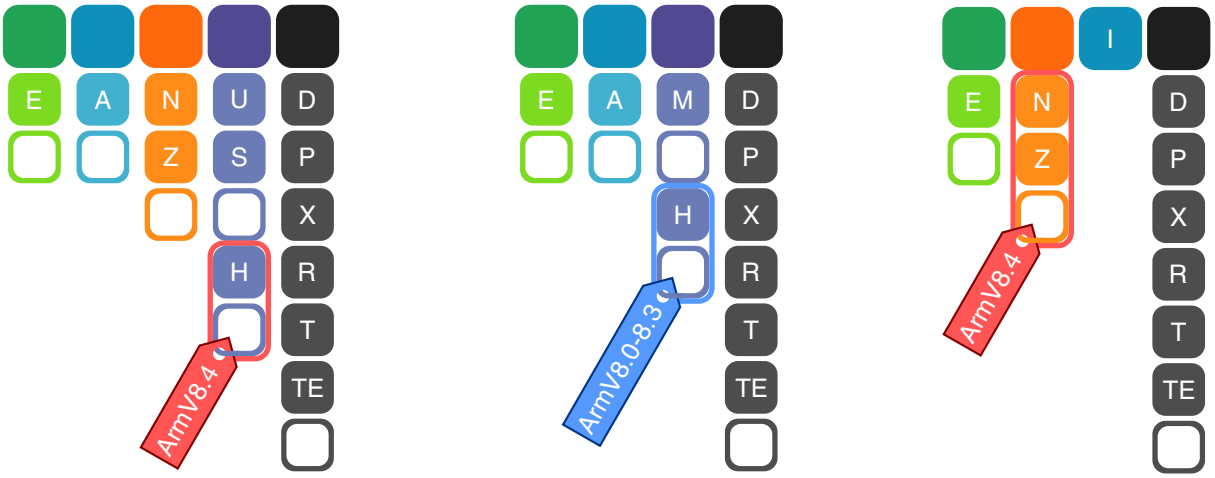
The illustrations below will show you how to combine access class specifiers for frequently-used access class combinations.

Rules to create a valid access class combination:

- From each column of an illustration 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.

©1989-2020 Lauterbach GmbH

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

©1989-2020 Lauterbach GmbH

CoreSight Access

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

Example combinations:

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

Cache and Virtual Memory Access

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

©1989-2020 Lauterbach GmbH

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:

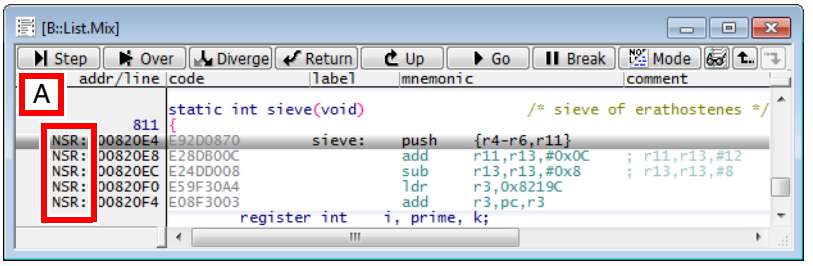
- The current CPU context (architecture specific)
- The used window type (e.g. **Data.dump** window for data or **List.Mix** window for code)
- Symbol information of the loaded application (e.g. combination of code and data)
- Segments that use different instruction sets
- Debugger specific settings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

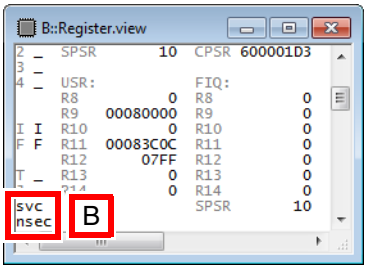
Examples: Memory Access through CPU

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.

©1989-2020 Lauterbach GmbH

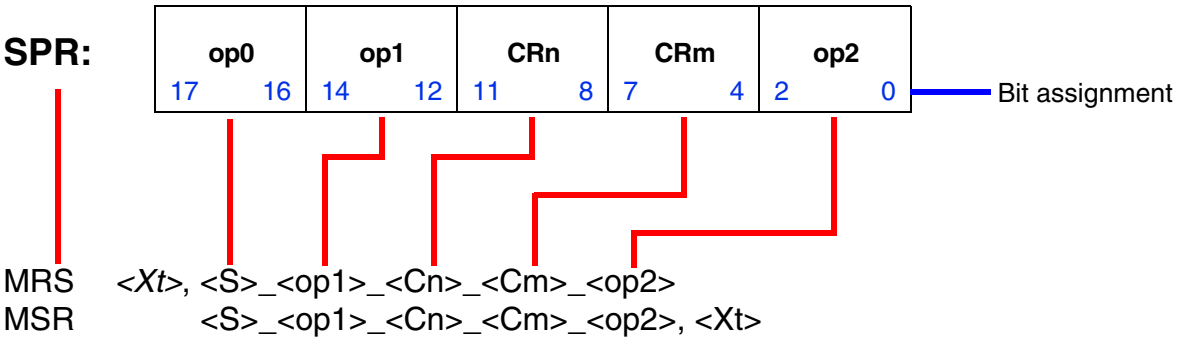
Only the AArch64 mode provides System Registers (SPR). Their functionality is similar to Coprocessor registers. Use the SPR access class for a read or write. SPR registers can also be accessed in the **PER.view** window.

A TRACE32 SPR address takes five parameters <op0>, <op1>, <CRn>, <CRm>, <op2> which are defined for a MRS or MSR instruction:

```
MRS <Xt>, <S>_<op1>_<Cn>_<Cm>_<op2> | MSR <S>_<op1>_<Cn>_<Cm>_<op2>, <Xt>
```

NOTE:
- The parameter *<S>* is also called *<op0>* in the 64-bit instruction set.
- SYS instructions equal MSR or MRS instructions with <S> = 1

Each parameter is a nibble (4-bit) in the SPR address. The parameter arrangement is:



SPR example register	Parameters, binary (hex) representation	Debugger access
VTTBR_EL2 (64-bit read-write register)	op0=11 (0x3) op1=100 (0x4) CRn=0010 (0x2) CRm=0001 (0x1) op2=000 (0x0)	spr:0x34210
IFSR32_EL2 (32-bit read-write register)	op0=11 (0x3) op1=100 (0x4) CRn=0101 (0x5) CRm=0000 (0x0) op2=001 (0x1)	spr:0x34501
AT S12E1R (64-bit write-only system instruction)	op0=01 (0x1) op1=100 (0x4) CRn=0111 (0x7) CRm=1000 (0x8) op2=101 (0x5)	spr:0x14785

©1989-2020 Lauterbach GmbH

Example 1: Write a 64-bit value to VTTBR_EL2 via the TRACE32 command line.

```
Data.Set SPR:0x34210 %Quad 0x20080000000        // Write to VTTBR_EL2
```

Example 2: Single read of a 32-bit value from IFSR32_EL2 via the TRACE32 command line.

```
Data.In SPR:0x34501 /Long                       // Read from IFSR_EL2
```

Example 3: Write 64-bit virtual address to AT S12E1R system instruction via the TRACE32 command line.

```
Data.Set SPR:0x14785 %Quad 0x20080C08280        // Execute system instr.
```

NOTE:	Unlike the AArch32 TrustZone concept, there are no secure / non-secure banked registers. The accessibility of a SPR depends mainly on the current exception level of the CPU, in a few cases a SPR might only be accessible in secure mode.

SPR Access in Specific CPU Mode

The debugger will access the SPR register in the current CPU mode if only the “SPR” access class is specified during the access. This means that a register might not be accessible by the debugger if the CPU has no access to register from its current mode.

You can enforce a mode switch during the access by adding the access class specifiers “M”, “H”, “Z” or “N”:

MSPR	Access SPR register in EL3 (AArch64) mode
HSPR	Access SPR register in EL2 (AArch64) mode
ZSPR	Access SPR register in secure EL1 (AArch64) mode
NSPR	Access SPR register in non-secure EL1 (AArch64) mode

The following examples assume that CPU is in non-secure EL1 mode.

Example 1: Access SCR_EL3, which can only be done in EL3 mode.

```
Data.Set MSPR:0x36110 %Long 0x401               // Switch to EL3 mode
```

Example 2: Access HCR_EL2, which can only be done in EL2 mode.

```
Data.In HSPR:0x34110 /Quad                      // Switch to EL2 mode
```

©1989-2020 Lauterbach GmbH

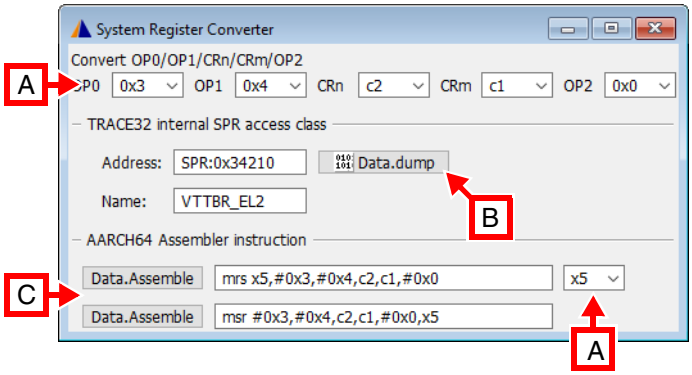
SPR Converter Dialog

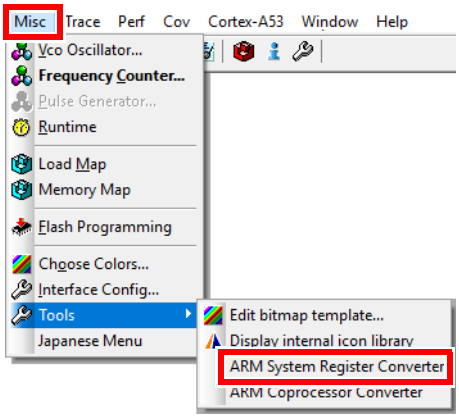
The demo directory offers a **System Register Converter** dialog, which allows you to assemble or display the MSR/MRS opcodes for a known register parameter set.

To display the **System Register Converter** dialog, run this command:

```
DO ~~/demo/arm64/etc/systemregister/systemregister_converter.cmm
```

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





A Edit SPR parameters and GPR here.

B Open **Data.dump** window at current SPR address.

C Assemble MRS/MSR instruction at current PC location.

SPR access in per file

Usually per files use the “SPR” access class to access system registers. This means that the command “PER” will open a per file window that shows you the current CPU view, i.e. system registers might not be accessible and be displayed as “????????” dependant on the current CPU mode.

You can explicitly access system registers with higher access priviledges by using **SYStem.Option.DBGSPR** ON.

©1989-2020 Lauterbach GmbH

Only the AArch32 mode provides Coprocessor registers. Coprocessors 14 and Coprocessors 15 are accessible (if available) in the processor.

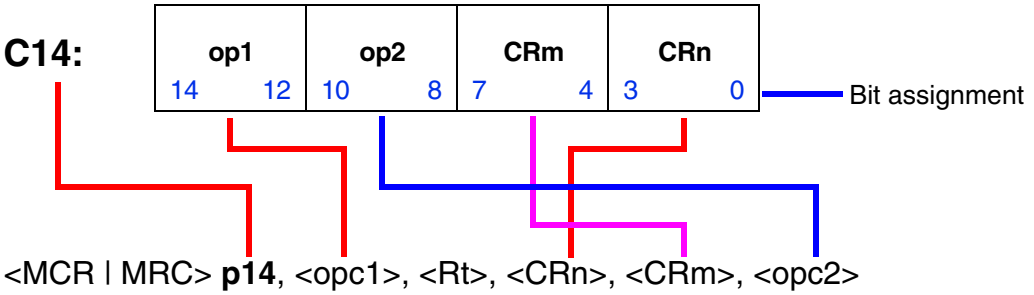
Coprocessor 14 (CP14), 32-bit Access

CP14 registers allow to control debug and trace components. Use the access class “C14” to access CP14 registers. Please refer to the ARM documentation for details. CP14 registers can also be controlled in the **PER.view** window.

The TRACE32 C14 address takes four parameters <CRn>, <CRm>, <op1>, <op2> which are defined for a MCR or MRC instruction.

```
<MCR|MRC> p14, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>
```

Each parameter is a nibble (4bit) in the C14 address. The order of the parameters is:



C14 example register	Parameters, binary (hex) representation	Debugger access
DBGAUTHSTATUS (read-only register)	op1=000 (0x0) CRn=0111 (0x7) CRm=1110 (0xE) op2=110 (0x6)	C14:0x06E7
DBGDTRTX (read-write register)	op1=000 (0x0) CRn=0011 (0x3) CRm=0010 (0x2) op2=000 (0x0)	C14:0x0023

Example 1: Single read of DBGAUTHSTATUS via the TRACE32 command line.

```
Data.In C14:0x06E7 /Long                    // Single read access to
                                            // DBGAUTHSTATUS
```

Example 2: Write to DBGDTRTX via the TRACE32 command line.

```
Data.Set C14:0x0023 %Long 0x10438012        // Write 0x10438012 to
                                            // DBGDTRTX
```

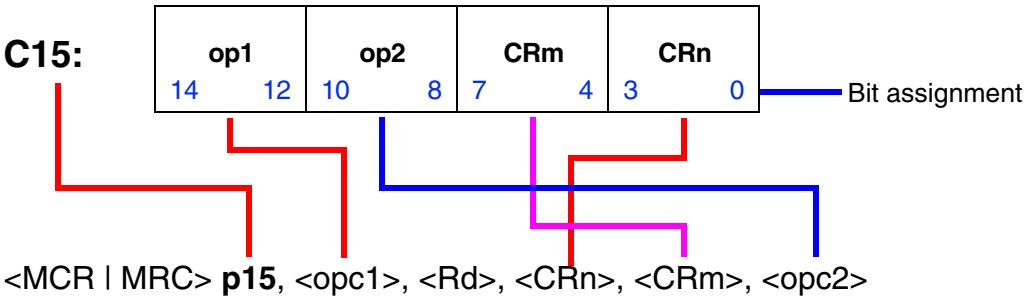
©1989-2020 Lauterbach GmbH

C15 allows the control of basic CPU functions. Use the access class “C15” to access CP15 registers. Please refer to the Arm documentation for details. CP15 registers can also be controlled in the **PER.view** window.

The TRACE32 C15 address takes four parameters <CRn>, <CRm>, <op1>, <op2> which are defined for a MCR or MRC instruction.

```
<MCR|MRC> p15, <op1>, <Rt>, <CRn>, <CRm>, <opc2>
```

Each parameter is a nibble (4bit) in the C15 address. The arrangement of the parameters is as follows:



C15 example register	Parameters, binary (hex) representation	Debugger access
HMAIR1 (read-write register)	op1=100 (0x4) CRn=1010 (0xA) CRm=0010 (0x2) op2=001 (0x1)	C15:0x412A
SCR (read-write register)	op1=000 (0x0) CRn=0001 (0x1) CRm=0001 (0x1) op2=000 (0x0)	C15:0x0011

Example 1: Single read of HMAIR1 via the TRACE32 command line.

```
Data.In C15:0x412A /Long                    // Single read access to
                                            // HMAIR1
```

Example 2: Write to SCR via the TRACE32 command line.

```
Data.Set C15:0x0011 %Long 0x1               // Write 0x1 to SCR
                                            // (switch to non-secure)
```

NOTE: On devices having a TrustZone (e.g. Cortex-A53) there are some banked CP15 registers, one for secure and one for non-secure mode. The “C15:” access class provides the view of the current mode of the core. With “ZC15:” and “NC15:” you can access the secure / non-secure bank independent of the current core mode.

©1989-2020 Lauterbach GmbH

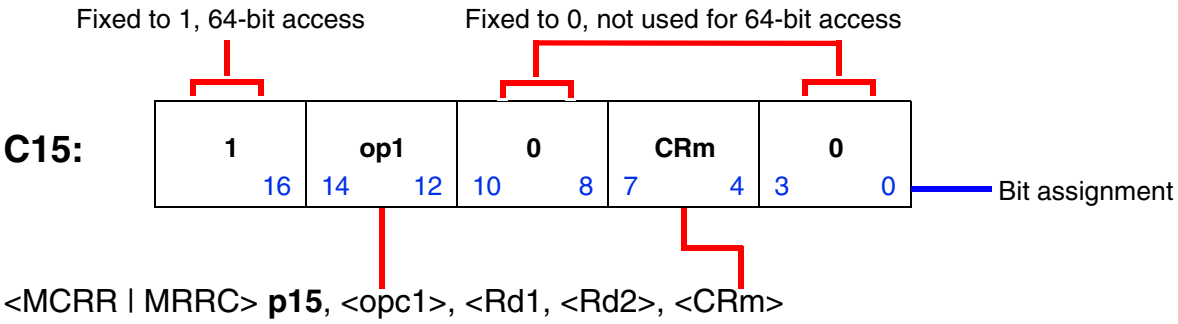
Coprocessor 15 (CP15), 64-bit Access

Some registers of the coprocessor registers in AArch32 are 64-bit wide. Those registers require a different access. The 64-bit CP15 registers can also be controlled in the **PER** window.

The TRACE32 C15 address takes two parameters <CRm>, <op1> which are defined for a MCRR or MRRC instruction.

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

Each parameter is a nibble (4bit) in the C15 address. The order of the parameters is:



Unlike the 32-bit coprocessor address, a “fifth” parameter at bit 16 is needed that is fixed to 1. This shows TRACE32 that a 64-bit Coprocessor register is accessed. The parameters <op2> and <CRn> are not used.

Example	Parameters, binary (hex) representation	Debugger access
HTTBR (read-write register)	op1=100 (0x4) CRm=0010 (0x2)	C15:0x14020

Example 1: Single read of HTTBR via the TRACE32 command line.

```
Data.In c15:0x14020 /Quad                    // Single read access to
                                             // HTTBR
```

Example 2: Write to HTTBR via the TRACE32 command line.

```
Data.Set c15:0x14020 %Quad 0xFE8000000       // Write 0xFE8000000 to
                                             // HTTBR
```

NOTE:	On devices having a TrustZone (e.g. Cortex-A53) there are some banked CP15 registers, one for secure and one for non-secure mode. The “C15:” access class provides the view of the current mode of the core. With “ZC15:” and “NC15:” you can access the secure / non-secure bank independent of the current core mode.

©1989-2020 Lauterbach GmbH

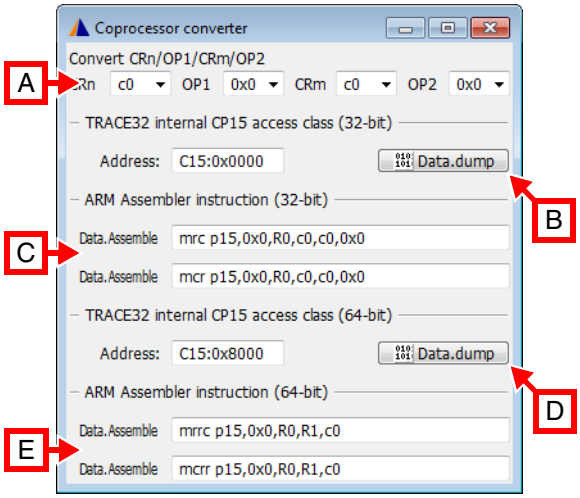
Coprocessor Converter Dialog

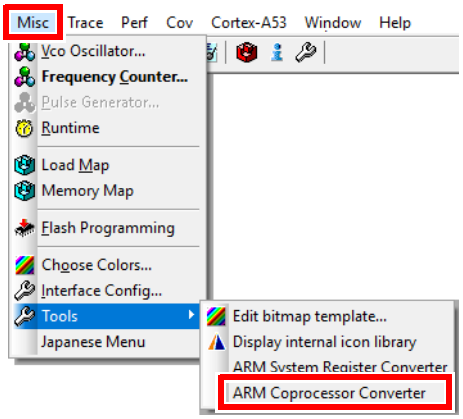
The demo directory offers a **Coprocessor converter** dialog, which allows you to assemble or display the MCR/MRC (32-bit Coprocessor registers) or MCRR/MRRC (64-bit Coprocessor register) opcodes for a known register parameter set.

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.

Coprocessor access in per file

Usually per files use the “C1x” access class to access coprocessors. This means that the command **PER** will open a per file window that shows you the current CPU view, i.e. coprocessors might not be accessible and be displayed as “????????” dependant on the current CPU mode.

For enforcing a specific secure access mode, you can pass options to the PER command:

```
PER , /Secure                // Use “ZC1x” instead of “C1x” in per file
PER , /NonSecure             // Use “NC1x” instead of “C1x” in per file
```

©1989-2020 Lauterbach GmbH

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.

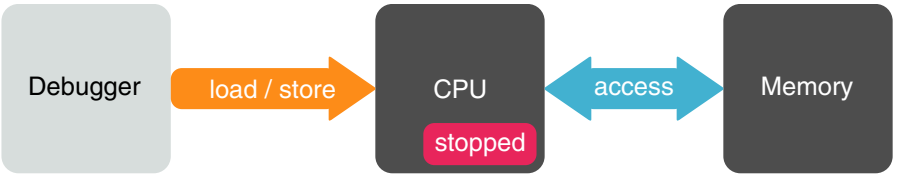
In this section:

- Intrusive and Non-intrusive Run-time Access
- Cache Coherent Non-intrusive Run-time Access
- Performing Intrusive and Non-intrusive Run-time Accesses with TRACE32
- Performing Cache Coherent Non-intrusive Run-time Accesses with TRACE32
- Additional Considerations

Intrusive and Non-intrusive Run-time Access

Intrusive run-time access

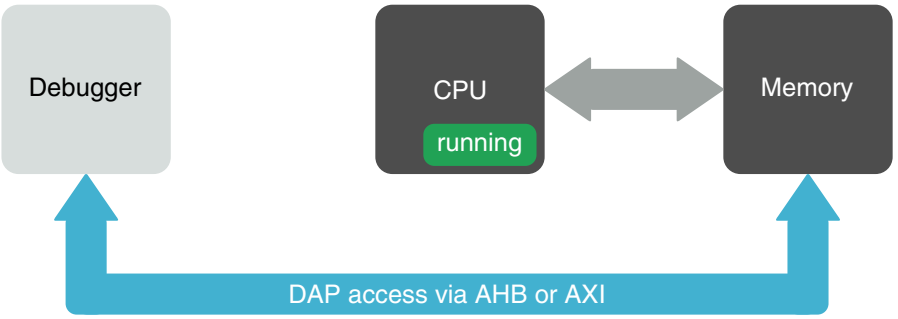
Intrusive means that the CPU is periodically stopped and restarted, so that the debugger can access the memory content through the CPU using load / store commands.



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.

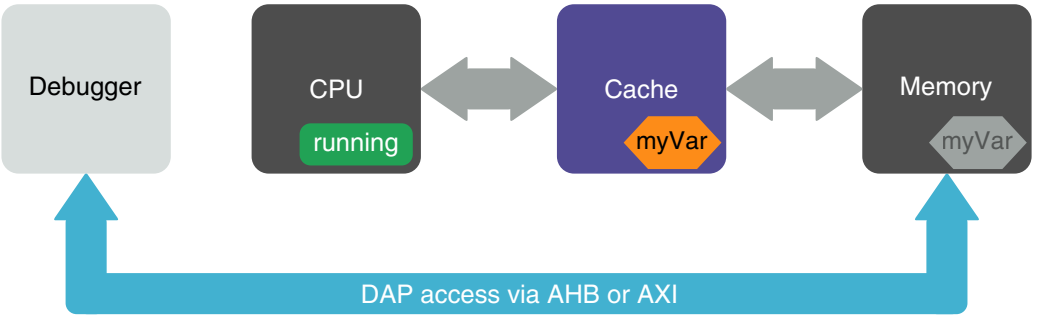


The debugger cannot read through the CPU while it is running and continuously accessing memory. Therefore the debugger has to use a DAP access, i.e. the AHB or AXI bus. The CPU is bypassed, which will equal a physical memory access. This way the real-time constrains are preserved. This access method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

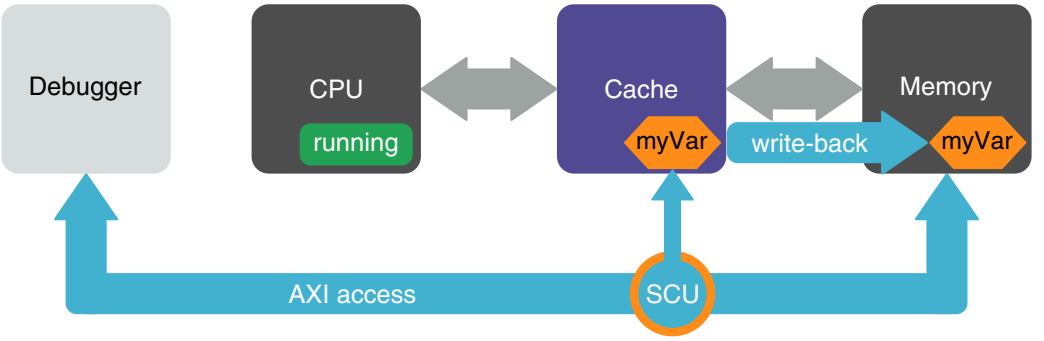
©1989-2020 Lauterbach GmbH

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
```

©1989-2020 Lauterbach GmbH

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
```

©1989-2020 Lauterbach GmbH

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).

©1989-2020 Lauterbach GmbH

Semihosting

Semihosting is a technique for application programs 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.

In AArch32 mode, 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.

In AArch64 mode, a semihosting call is done without exception using the HLT instruction together with a certain constant number to indicate a semihosting request.

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 operations used for keyboard input, screen output, and file I/O directly. These 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 or HLT event.

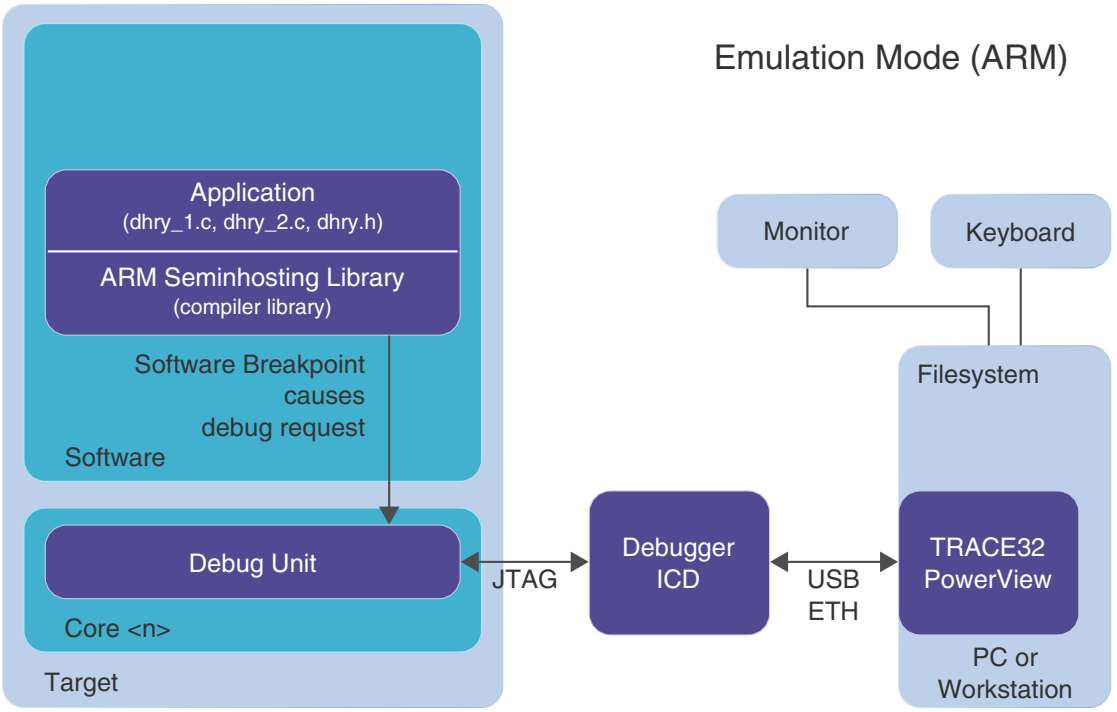
©1989-2020 Lauterbach GmbH

There is no need to set any additional breakpoints since the HLT instruction itself will stop the core. The immediate of the HLT instruction has to be 0xF000 to indicate a semihosting request. The debugger will restart the core after the semihosting data is processed.

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.

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 for AArch64 can be found in
~~/demo/arm64/etc/semihosting_arm_emulation/aarch64/halt_armv8.cmm



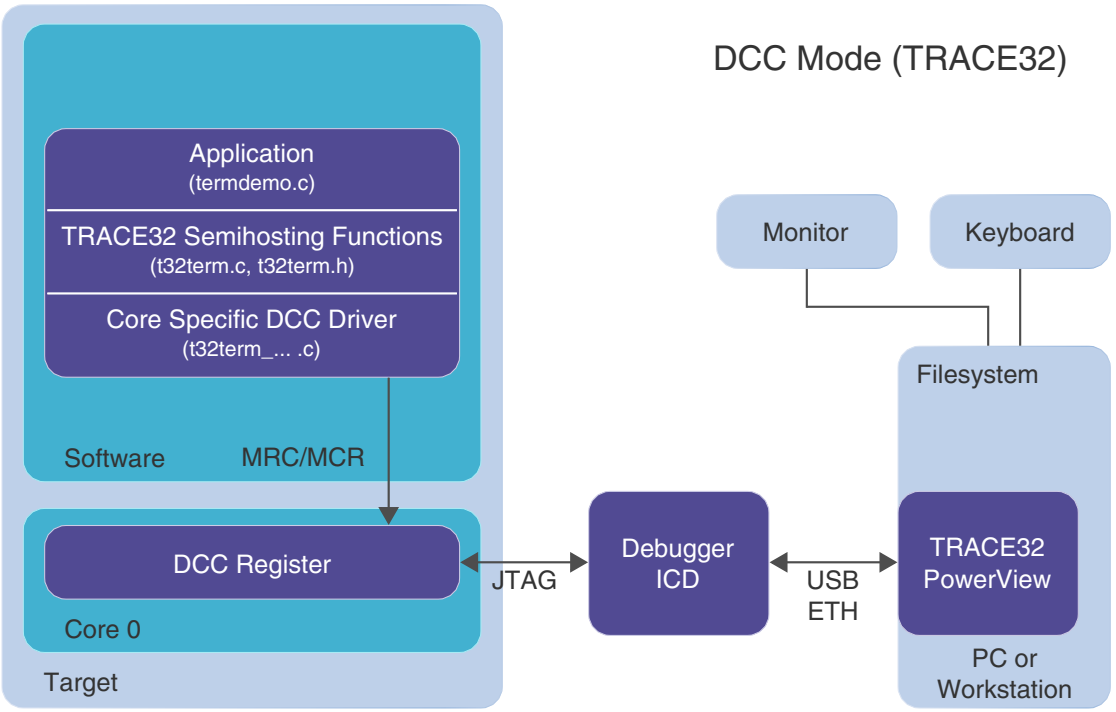
©1989-2020 Lauterbach GmbH

In AArch64 you cannot use the ARM library for semihosting (DCC mode (ARM) as in AArch32 mode) since the HLT instruction will stop the target. Therefore an exception handler cannot be executed which would handle the DCC communication.

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.

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.

Find examples and source codes in ~~/demo/arm64/etc/semihosting_trace32_dcc/aarch32 and ~~/demo/arm64/etc/semihosting_trace32_dcc/aarch64



©1989-2020 Lauterbach GmbH

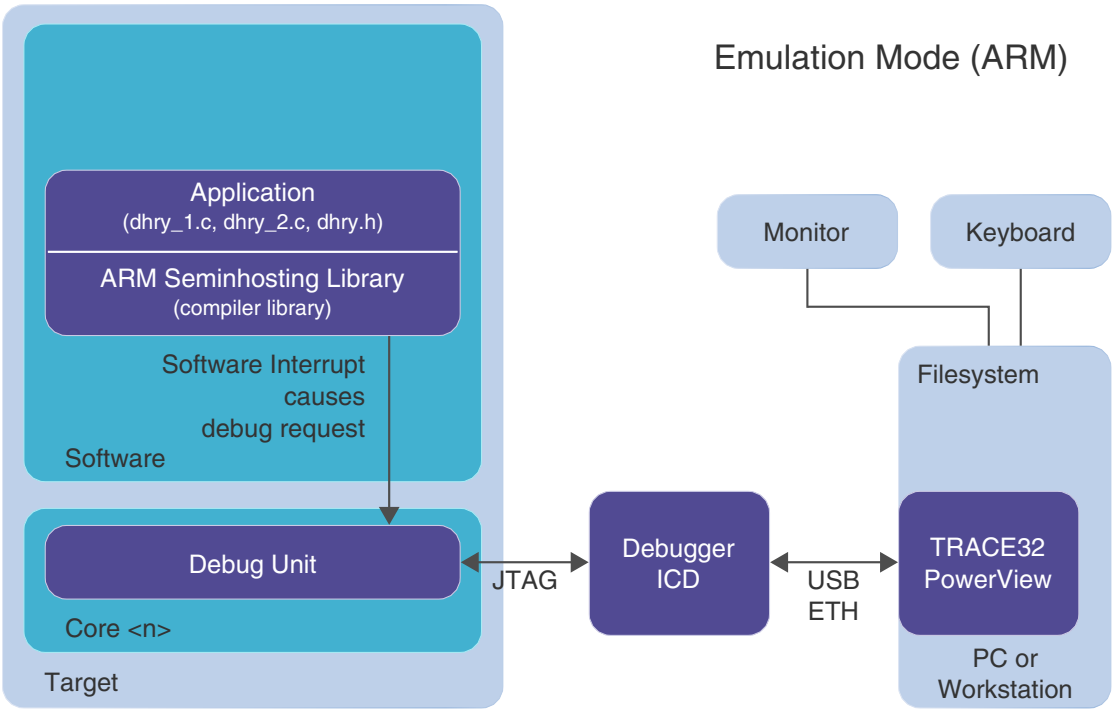
A breakpoint placed on the SVC exception entry stops the application. The debugger handles the request while the application is stopped, provides the required communication with the host, and restarts the application at the address which was stored in the link register ELR on the SVC exception call. 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.

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. 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 ELR. You could for example place an 'ERET' command at that address and hand the return address in ELR. 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 for AArch32 can be found in
~~/demo/arm64/etc/semihosting_arm_emulation/aarch32/swisoft_armv8.cmm

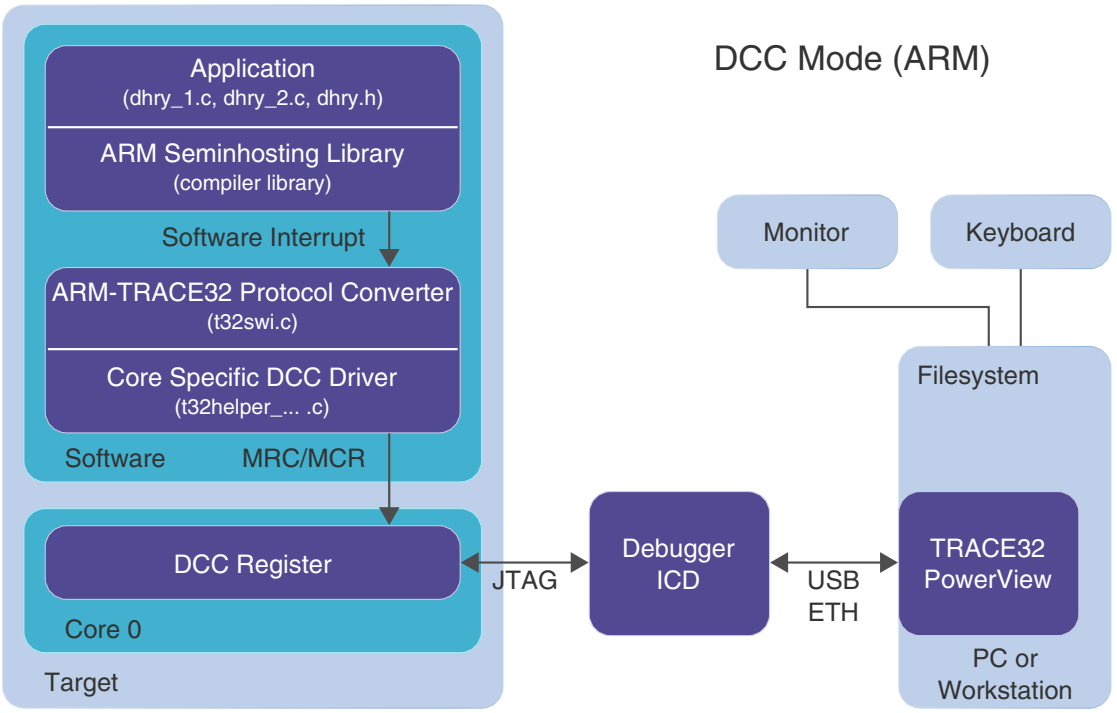


©1989-2020 Lauterbach GmbH

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. This mode can only be used if the DCC is provided by the target.

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.

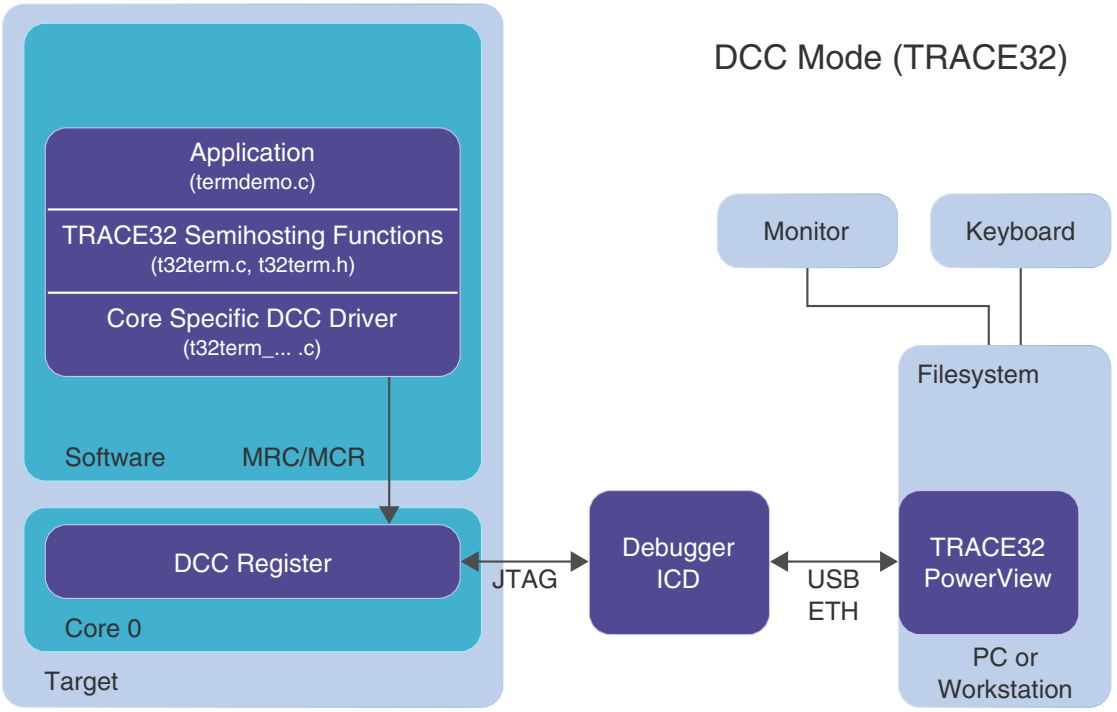
An example (swidcc_x.cmm) and the source of the ARM compatible semihosting handler (t32swi.c, t32helper_x.c) can be found in ~~/demo/arm64/etc/semihosting_arm_dcc/aarch32



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.

©1989-2020 Lauterbach GmbH

For examples and source codes, see ~~/demo/arm64/etc/semihosting_trace32_dcc/aarch32



©1989-2020 Lauterbach GmbH

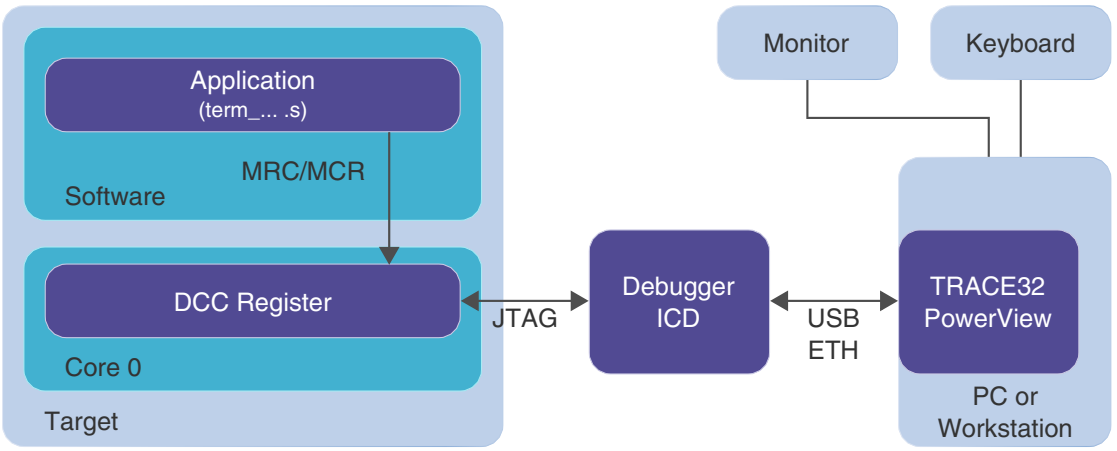
The command **TERM** opens a terminal window which allows to communicate with the ARM core over the Debug Communications Channel (DCC). All data received from the DCC are displayed and all data inputs to this window are sent to the DCC. Communication occurs byte wide or up to four bytes per transfer. 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 TRACE32 directory ~~/demo/arm64/etc/virtual_terminal contains examples for the different ARM families which demonstrate how the communication works.

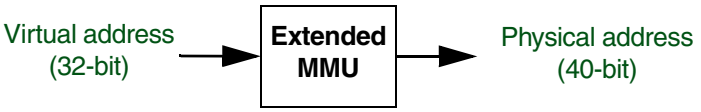
Virtual Terminal



©1989-2020 Lauterbach GmbH

Large Physical Address Extension (LPAE)

LPAE is an optional extension in AArch32 mode. 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.



Virtual address (32-bit) --> Extended MMU --> Physical address (40-bit)

It is for example implemented on Cortex-A53 and Cortex-A57.

Please note that in AArch64 mode there is always a physical address space above 32-bit accessible.

Consequence for Debugging

Only the physical address have been extended in AArch32, 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 in AArch32. The reason is that the load and store instructions to access memory only support 32-bit addresses in AArch32 mode. However, 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. AXI will accept full 64-bit addresses.

```
Data.dump AXI:0x0280004000
```

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.

©1989-2020 Lauterbach GmbH

Virtualization Extension, Hypervisor

The ‘Virtualization Extension’ is an optional extension. It can be found on ARMv8 based devices like Cortex-A53, Cortex-A55, Cortex-A72 etc. It adds a ‘Hypervisor/EL2’ processor mode used to switch between different guest operating systems. The extension assumes **LPAE** in AArch32 and **TrustZone**/EL3. 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.

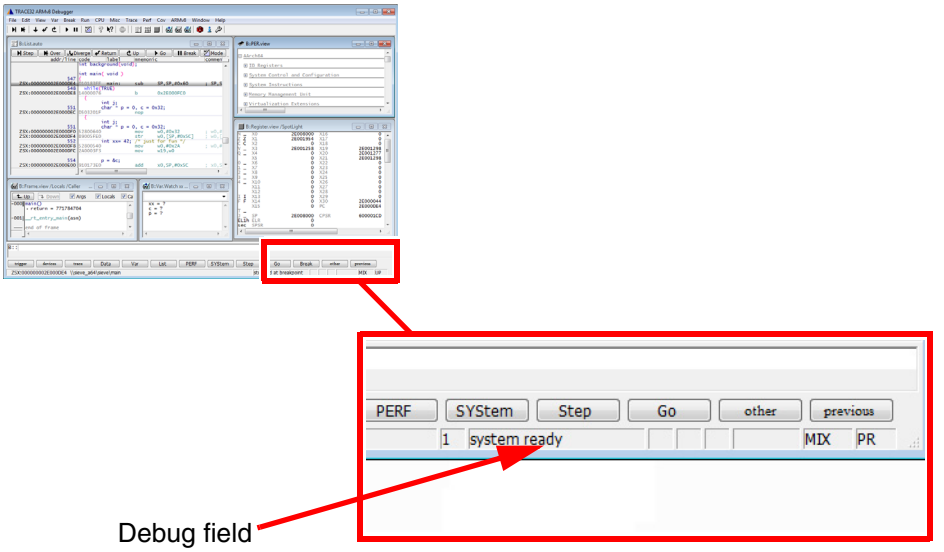
You can force a certain view/translation by switching to another mode or by using the access classes “Z:” (secure), “N:” (non-secure), “H:” (Hypervisor/EL2) or “M:” (AArch64 EL3).

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

©1989-2020 Lauterbach GmbH

Debug Field

The debug field shows various state information, e.g. run state of the CPU, halt reasons, system modes, etc. Not all information is debugger specific. See also "**State Line**" (ide_user.pdf).



Run Mode

The following debugger specific run states are displayed in the debug field:

Run state	Description
running (power down)	The core is not powered.
no CTI	The core is not powered and the CTI is inaccessible. Armv8.3 DoPD is not implemented.
no CPU	The core is not powered and the CTI is accessible. Armv8.3 DoPD is not implemented.
CTI hlt	The core is not powered and the CTI is accessible. A halt event is pending, waiting to stop the CPU. Armv8.3 DoPD is not implemented.
no CTI: DoPD	The core is not powered and the CTI is inaccessible Armv8.3 DoPD is not implemented.
no CPU: DoPD	The core is not powered and the CTI is accessible Armv8.3 DoPD is implemented.
CTI hlt DoPD	The core is not powered and the CTI is accessible. A halt event is pending, waiting to stop the CPU. Armv8.3 DoPD is implemented.

©1989-2020 Lauterbach GmbH

running (secured)	Core is secured. Access by debugger not possible. Armv8.3 DoPD is implemented.
running (sticky reset)	Core is running with sticky reset bit set.
running (OS lock)	Core is running with OS-lock bit set.
running (OS lock/catch)	Core is running with OS-lock bit set and OS catch event is enabled for this core when **SYStem.Option BreakOS** is **OFF**.
disabled	The debugger detected that CSW.DeviceEn is low. The debugger assumes the device is currently suspended. It will wait up to 10 minutes for the device to become available.

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.

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.

©1989-2020 Lauterbach GmbH

ARM specific SYStem Commands

The debugger and the target behavior can be configured via the TRACE32 command line or via the **SYStem.state** window.

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** | **DAP** | **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**.

©1989-2020 Lauterbach GmbH

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**.
DAP	The **DAP** tab informs the debugger about an ARM CoreSight Debug Access Port (DAP) and about how to control the DAP to access chip-internal memory busses (AHB, APB, AXI) or chip-internal JTAG interfaces. For descriptions of the commands on the **DAP** tab, see **DAP**.
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** | **CJTAGSWD**] **NIDNTTRSTTORST** [**ON** | **OFF**]

©1989-2020 Lauterbach GmbH

<parameter>: (DebugPort cont.)	**NIDNTPSRISINGEDGE** [**ON** | **OFF**] **NIDNTRSTPOLARITY** [**High** | **Low**] **PortSHaRing** [**ON** | **OFF** | **Auto**] **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>*
<parameter>: (JTAG cont.)	**DAP2DRPOST** *<bits>* **DAP2DRPRE** *<bits>* **DAP2IRPOST** *<bits>* **DAP2IRPRE** *<bits>* **DAPDRPOST** *<bits>* **DAPDRPRE** *<bits>* **DAPIRPOST** *<bits>* **DAPIRPRE** *<bits>*
<parameter>: (JTAG cont.)	**DRPOST** *<bits>* **DRPRE** *<bits>* **ETBDRPOST** *<bits>* **ETBDRPRE** *<bits>* **ETBIRPOST** *<bits>* **ETBIRPRE** *<bits>* **IRPOST***<bits>* **IRPRE** *<bits>*
<parameter>: (JTAG cont.)	**NEXTDRPOST** *<bits>* **NEXTDRPRE** *<bits>* **NEXTIRPOST***<bits>* **NEXTIRPRE** *<bits>* **RTPDRPOST** *<bits>* **RTPDRPRE** *<bits>* **RTPIRPOST** *<bits>* **RTPIRPRE** *<bits>*

©1989-2020 Lauterbach GmbH

<parameter>: (JTAG cont.)	**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>: **(DAP)**	**AHBACCESSPORT** *<port>* **APBACCESSPORT** *<port>* **AXIACCESSPORT** *<port>* **COREJTAGPORT** *<port>* **DAP2AHBACCESSPORT** *<port>* **DAP2APBACCESSPORT** *<port>* **DAP2AXIACCESSPORT** *<port>* **DAP2COREJTAGPORT** *<port>*
<parameter>: (DAP cont.)	**DAP2DEBUGACCESSPORT** *<port>* **DAP2JTAGPORT** *<port>* **DAP2AHBACCESSPORT** *<port>* **DEBUGACCESSPORT** *<port>* **JTAGACCESSPORT** *<port>* **MEMORYACCESSPORT** *<port>*
<parameter>: **(COmponents)**	**ADTF.Base** *<address>* **ADTF.RESET** **ADTF.Type** [**NONE** | **ADTF** | **ADTF2** | **GEM**] **AET.Base** *<address>* **AET.RESET** **BMC.Base** *<address>* **BMC.RESET** **CMI.Base** *<address>* **CMI.RESET**

©1989-2020 Lauterbach GmbH

<parameter>:
(COmponents cont.)

CMI.TraceID *<id>*
COREDEBUG.Base *<address>*
COREDEBUG.RESET
CTI.Base *<address>*
CTI.Config [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**]
CTI.RESET
DRM.Base *<address>*
DRM.RESET

<parameter>:
(COmponents cont.)

DTM.RESET
DTM.Type [**None** | **Generic**]
DWT.Base *<address>*
DWT.RESET
EPM.Base *<address>*
EPM.RESET
ETB2AXI.Base *<address>*
ETB2AXI.RESET

<parameter>:
(COmponents cont.)

ETB.ATBSource *<source>*
ETB.Base *<address>*
ETB.NoFlush [**ON** | **OFF**]
ETB.RESET
ETB.Size *<size>*
ETF.ATBSource *<source>*
ETF.Base *<address>*
ETF.RESET
ETM.Base *<address>*

<parameter>:
(COmponents cont.)

ETM.RESET
ETR.ATBSource *<source>*
ETR.Base *<address>*
ETR.RESET
FUNNEL.ATBSource *<sourcelist>*
FUNNEL.Base *<address>*
FUNNEL.Name *<string>*
FUNNEL.PROGrammable [**ON** | **OFF**]

<parameter>:
(Components cont.)

FUNNEL.RESET
HSM.Base *<address>*
HSM.RESET
HTM.Base *<address>*
HTM.RESET
HTM.Type [**CoreSight** | **WPT**]
ICE.Base *<address>*
ICE.RESET

©1989-2020 Lauterbach GmbH

<parameter>: (Components cont.)

ITM.Base *<address>*
ITM.RESET
L2CACHE.Base *<address>*
L2CACHE.RESET
L2CACHE.Type [**NONE** | **Generic** | **L210** | **L220** | **L2C-310** | **AURORA** | **AURORA2**]
OCP.Base *<address>*
OCP.RESET
OCP.TraceID *<id>*

<parameter>: (Components cont.)

OCP.Type *<type>*
PMI.Base *<address>*
PMI.RESET
PMI.TraceID *<id>*
RTP.Base *<address>*
RTP.PerBase *<address>*
RTP.RamBase *<address>*
RTP.RESET

<parameter>: (Components cont.)

SC.Base *<address>*
SC.RESET
SC.TraceID *<id>*
STM.Base *<address>*
STM.Mode [**NONE** | **XTIv2** | **SDTI** | **STP** | **STP64** | **STPv2**]
STM.RESET
STM.Type [**None** | **GenericARM** | **SDTI** | **TI**]
TPIU.ATBSource *<source>*
TPIU.Base *<address>*
TPIU.RESET
TPIU.Type [**CoreSight** | **Generic**]

<parameter>: **(Deprecated)**

BMCBASE *<address>*
BYPASS *<seq>*
COREBASE *<address>*
CTIBASE *<address>*
CTICONFIG [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**]
DEBUGBASE *<address>*
DTMCONFIG [**ON** | **OFF**]

<parameter>: (Deprecated cont.)

DTMETBFUNNELPORT *<port>*
DTMFUNNEL2PORT *<port>*
DTMFUNNELPORT *<port>*
DTMTPIUFUNNELPORT *<port>*
DWTBASE *<address>*
ETB2AXIBASE *<address>*
ETBBASE *<address>*

©1989-2020 Lauterbach GmbH

<parameter>: (Deprecated cont.)	**ETBFUNNELBASE** *<address>* **ETFBASE** *<address>* **ETMBASE** *<address>* **ETMETBFUNNELPORT** *<port>* **ETMFUNNEL2PORT** *<port>* **ETMFUNNELPORT** *<port>* **ETMTPIUFUNNELPORT** *<port>* **FILLDRZERO** [**ON** | **OFF**]
<parameter>: (Deprecated cont.)	**FUNNEL2BASE** *<address>* **FUNNELBASE** *<address>* **HSMBASE** *<address>* **HTMBASE** *<address>* **HTMETBFUNNELPORT** *<port>* **HTMFUNNEL2PORT** *<port>* **HTMFUNNELPORT** *<port>* **HTMTPIUFUNNELPORT** *<port>*
<parameter>: (Deprecated cont.)	**ITMBASE** *<address>* **ITMETBFUNNELPORT** *<port>* **ITMFUNNEL2PORT** *<port>* **ITMFUNNELPORT** *<port>* **ITMTPIUFUNNELPORT** *<port>* **PERBASE** *<address>* **RAMBASE** *<address>* **RTPBASE** *<address>*
<parameter>: (Deprecated cont.)	**SDTIBASE** *<address>* **STMBASE** *<address>* **STMETBFUNNELPORT***<port>* **STMFUNNEL2PORT***<port>* **STMFUNNELPORT***<port>* **STMTPIUFUNNELPORT***<port>* **TIADTFBASE** *<address>* **TIDRMBASE** *<address>*
<parameter>: (Deprecated cont.)	**TIEPMBASE** *<address>* **TIICEBASE** *<address>* **TIOCPBASE** *<address>* **TIOCPTYPE** *<type>* **TIPMIBASE** *<address>* **TISCBASE** *<address>* **TISTMBASE** *<address>*

©1989-2020 Lauterbach GmbH

<parameter>: (Deprecated cont.)	**TPIUBASE** *<address>* **TPIUFUNNELBASE** *<address>* **TRACEETBFUNNELPORT** *<port>* **TRACEFUNNELPORT***<port>* **TRACETPIUFUNNELPORT** *<port>* **view**

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

This is a common description of the **SYStem.CONFIG** command group for the ARM, CevaX, TI DSP and Hexagon debugger. Each debugger will provide only a subset of these commands. Some commands need a certain CPU type selection (**SYStem.CPU** *<type>*) to become active and it might additionally depend on further settings.

Ideally you can select with **SYStem.CPU** the chip you are using which causes all setup you need and you do not need any further **SYStem.CONFIG** command.

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

Syntax Remarks

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

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

©1989-2020 Lauterbach GmbH

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 uTrace is used.

CORE *<core>* *<chip>*

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

Because of the default setting of this command

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

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

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

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

©1989-2020 Lauterbach GmbH

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> derived from `CORE=` parameter in the configuration file (config.t32), usually 1. If you start multiple debugger instances with the help of t32start.exe, you will get ascending values (1, 2, 3,...).

CoreNumber *<number>*

Number of cores to be considered in an SMP (symmetric multiprocessing) debug session. There are 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** | **CJTAGSWD**]

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:

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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 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 ON**.

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

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).

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

<parameters> describing the “JTAG” scan chain and signal behavior

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.

©1989-2020 Lauterbach GmbH

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.
Default: ON if `CORE=...` >1 in the configuration file (e.g. config.t32).
For CortexM: Please check also
SYStem.Option DISableSOFTRES [ON | OFF]

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.

©1989-2020 Lauterbach GmbH

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.

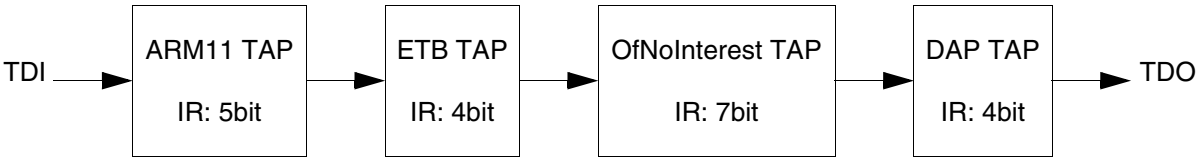
NEXT: If a memory access changes the JTAG chain and the core TAP position then you can specify the new values with the NEXT... parameter. After the access for example the parameter NEXTIRPRE will replace the IRPRE value and NEXTIRPRE becomes 0. Available only on ARM11 debugger.
-> NEXTDRPOST, NEXTDRPRE,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

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

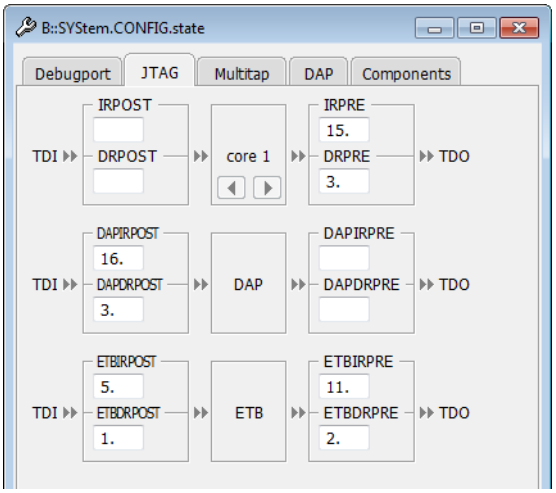
CHIP: Definition of a TAP or TAP sequence in a scan chain that needs a different Instruction Register (IR) and Data Register (DR) pattern than the default BYPASS (1...1) pattern.
-> CHIPDRPOST, CHIPDRPRE, CHIPIRPOST, CHIPIRPRE.

©1989-2020 Lauterbach GmbH

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.
```

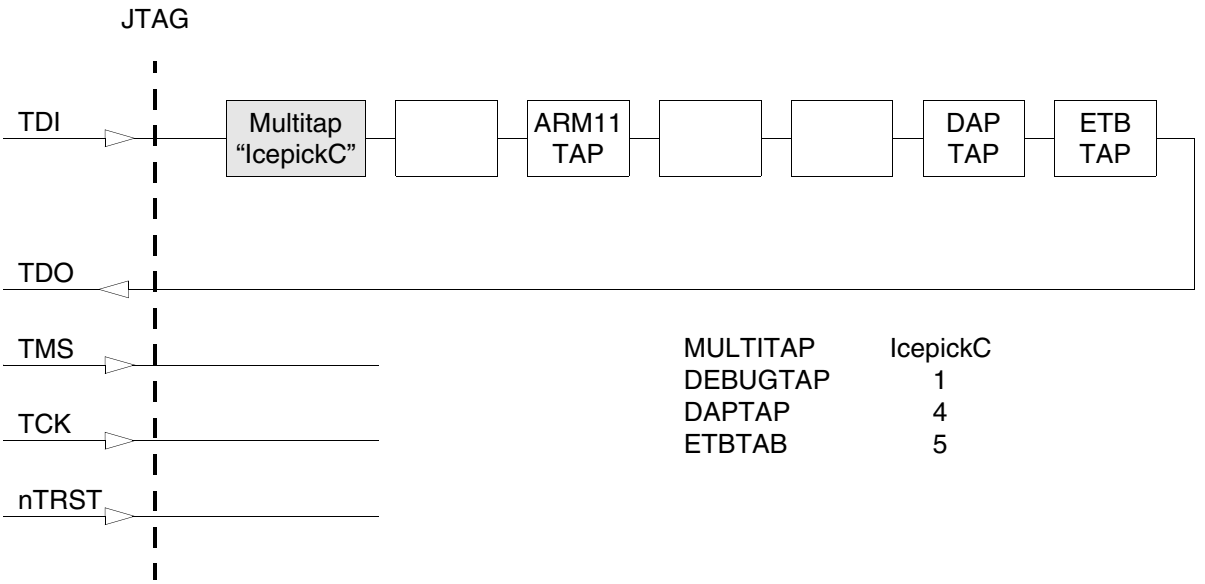


©1989-2020 Lauterbach GmbH

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:



MULTITAP	IcepickC
DEBUGTAP	1
DAPTAP	4
ETBTAB	5

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.

©1989-2020 Lauterbach GmbH

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).

©1989-2020 Lauterbach GmbH

A Debug Access Port (DAP) is a CoreSight module from ARM which provides access via its debugport (JTAG, cJTAG, SWD) to:

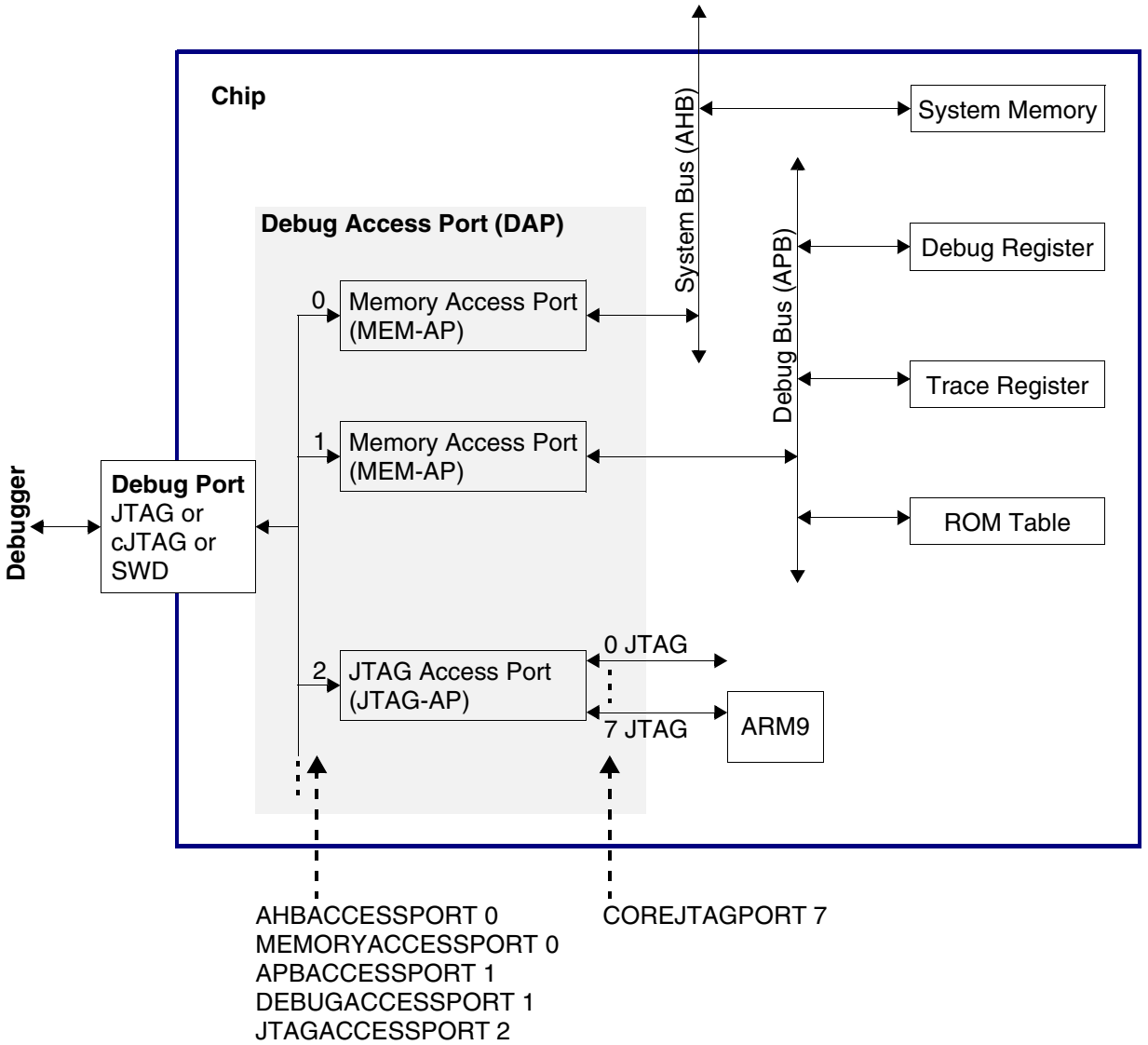
1. Different memory busses (AHB, APB, AXI). This is especially important if the on-chip debug register needs to be accessed this way. You can access the memory buses by using certain access classes with the debugger commands: “AHB:”, “APB:”, “AXI:, “DAP”, “E:”. The interface to these buses is called 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. At emulation or simulation system with using bus transactors the access to the busses must be specified by using the transactor identification name instead using the access port commands. For emulations/simulations with a DAP transactor the individual bus transactor name don’t need to be configured. Instead of this the DAP transactor name need to be passed and the regular access ports to the busses.

©1989-2020 Lauterbach GmbH

Example:



AHBACCESSPORT *<port>*	DAP access port number (0-255) which shall be used for “AHB:” access class. Default: *<port>*=0.
APBACCESSPORT *<port>*	DAP access port number (0-255) which shall be used for “APB:” access class. Default: *<port>*=1.
AXIACCESSPORT *<port>*	DAP access port number (0-255) which shall be used for “AXI:” access class. Default: port not available
COREJTAGPORT *<port>*	JTAG-AP port number (0-7) connected to the core which shall be debugged.

©1989-2020 Lauterbach GmbH

DAP2AHBACCESSPORT *<port>*
DAP2 access port number (0-255) which shall be used for "AHB2:" access class. Default: *<port>*=0.

DAP2APBACCESSPORT *<port>*
DAP2 access port number (0-255) which shall be used for "APB2:" access class. Default: *<port>*=1.

DAP2AXIACCESSPORT *<port>*
DAP2 access port number (0-255) which shall be used for "AXI2:" access class. Default: port not available

DAP2DEBUGACCESS-PORT *<port>*
DAP2 access port number (0-255) where the debug register can be found (typically on APB). Used for "DAP2:" access class. Default: *<port>*=1.

DAP2COREJTAGPORT *<port>*
JTAG-AP port number (0-7) connected to the core which shall be debugged. The JTAG-AP can be found on another DAP (DAP2).

DAP2JTAGPORT *<port>*
JTAG-AP port number (0-7) for an (other) DAP which is connected to a JTAG-AP.

DAP2MEMORYACCESS-PORT *<port>*
DAP2 access port number where system memory can be accessed even during runtime (typically on AHB). Used for "E:" access class while running, assuming "SYStem.MemoryAccess DAP2". Default: *<port>*=0.

DEBUGACCESSPORT *<port>*
DAP access port number (0-255) where the debug register can be found (typically on APB). Used for "DAP:" access class. Default: *<port>*=1.

JTAGACCESSPORT *<port>*
DAP access port number (0-255) of the JTAG Access Port.

MEMORYACCESSPORT *<port>*
DAP access port number where system memory can be accessed even during runtime (typically on AHB). Used for "E:" access class while running, assuming "SYStem.MemoryAccess DAP". Default: *<port>*=0.

AHBNAME *<name>*
AHB bus transactor name that shall be used for "AHB:" access class.

APBNAME *<name>*
APB bus transactor name that shall be used for "APB:" access class.

AXINAME *<name>*
AXI bus transactor name that shall be used for "AXI:" access class.

DAP2AHBNAME *<name>*
AHB bus transactor name that shall be used for "AHB2:" access class.

©1989-2020 Lauterbach GmbH

DAP2APBNAME *<name>*
APB bus transactor name that shall be used for “APB2:” access class.

DAP2AXINAME *<name>*
AXI bus transactor name that shall be used for “AXI2:” access class.

DAP2DEBUGBUSNAME *<name>*
APB bus transactor name identifying the bus where the debug register can be found. Used for “DAP2:” access class.

DAP2MEMORYBUSNAME *<name>*
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.MemoryAccess DAP2”.

DEBUGBUSNAME *<name>*
APB bus transactor name identifying the bus where the debug register can be found. Used for “DAP:” access class.

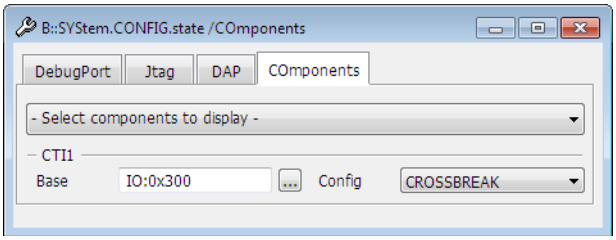
MEMORYBUSNAME *<name>*
AHB bus transactor name identifying the bus where system memory can be accessed even during runtime. Used for “E:” access class while running, assuming “SYStem.MemoryAccess DAP”.

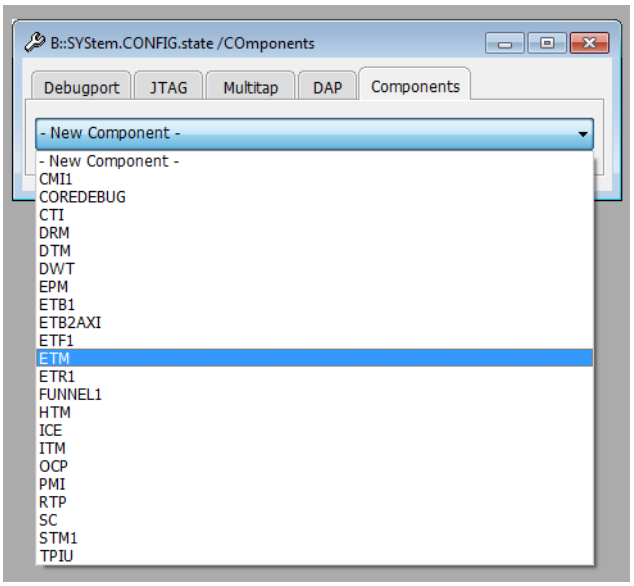
DAPNAME *<name>*
DAP transactor name that shall be used for DAP access ports.

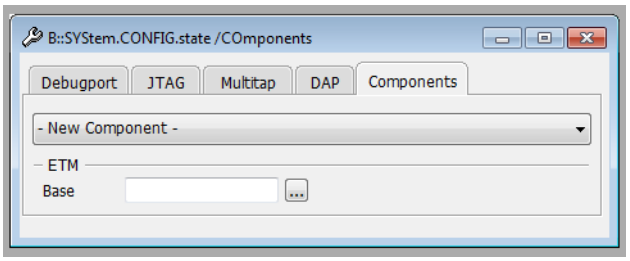
DAP2NAME *<name>*
DAP transactor name that shall be used for DAP access ports of 2nd order.

©1989-2020 Lauterbach GmbH

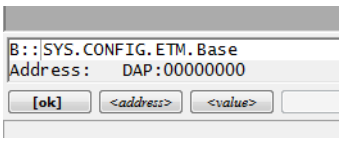
On the **Components** tab in the **SYStem.CONFIG.state** window, you can comfortably add the debug and trace components your chip includes and which you intend to use with the debugger’s help.







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.

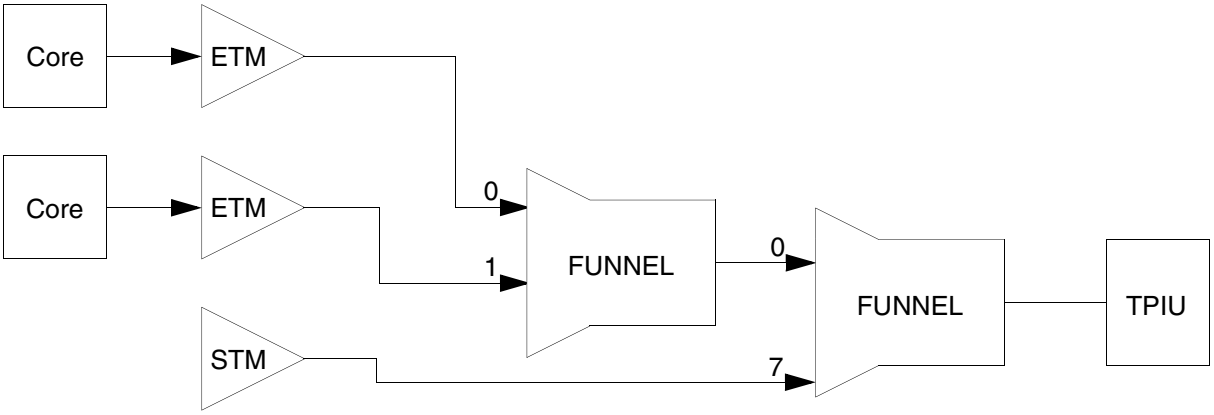


©1989-2020 Lauterbach GmbH

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

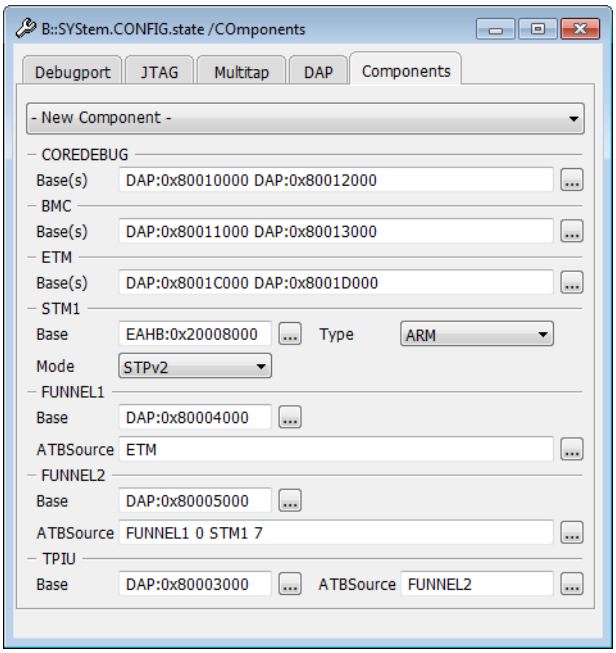
The *<address>* parameter can be just an address (e.g. 0x80001000) or you can add the access class in front (e.g. AHB:0x80001000). Without access class it gets the command specific default access class which is “EDAP:” in most cases.

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
```

©1989-2020 Lauterbach GmbH



… **.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:

©1989-2020 Lauterbach GmbH

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 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, 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**.

… **.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**.

©1989-2020 Lauterbach GmbH

… **.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.

ARMV8V1: Channel 0 and 1 of the CTM are used to distribute start/stop events from and to the CTIs. ARMv8 only.
ARMV8V2: Channel 2 and 3 of the CTM are used to distribute start/stop events from and to the CTIs. ARMv8 only.
ARMV8V3: Channel 0, 1 and 2 of the CTM are used to distribute start/stop events. Implemented on request. ARMv8 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.

©1989-2020 Lauterbach GmbH

ETB.NoFlush [**ON** | **OFF**]

Deactivates an ETB 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.

ETB.Size *<size>*

Specifies the size of the Embedded Trace Buffer. The ETB size can normally be read out by the debugger. Therefore this command is only needed if this can not be done for any reason.

ETM.StackMode [**NotAvail-bale** | **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.

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**]

IIn case the funnel can not or may not be programmed by the debugger, this option needs to be OFF. Default is ON.

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.

©1989-2020 Lauterbach GmbH

RTP.RamBase *<address>*	RAMBASE is the start address of RAM which accesses shall be traced. RAMBASE is needed for the RAM Trace Port (RTP) which is available on some derivatives from Texas Instruments. The trace packages include only relative addresses to PERBASE and RAMBASE.
STM.Mode [**NONE** | **XTIv2** | **SDTI** | **STP** | **STP64** | **STPv2**]	Selects the protocol type used by the System Trace Module (STM).
STM.Type [**None** | **Generic** | **ARM** | **SDTI** | **TI**]	Selects the type of the System Trace Module (STM). Some types allow to work with different protocols (see STM.Mode).
TPIU.Type [**CoreSight** | **Generic**]	Selects the type of the Trace Port Interface Unit (TPIU). CoreSight: Default. CoreSight TPIU. TPIU control register located at TPIU.Base *<address>* will be handled by the debugger. Generic: Proprietary TPIU. TPIU control register will not be handled by the debugger.

Components and Available Commands

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

ADTF.Base *<address>*
ADTF.RESET
ADTF.Type [None | ADTF | ADTF2 | GEM]
AMBA trace bus DSP Trace Formatter (ADTF) - Texas Instruments
Module of a TMS320C5x or TMS320C6x core converting program and data trace information in ARM CoreSight compliant format.

AET.Base *<address>*
AET.RESET
Advanced Event Triggering unit (AET) - Texas Instruments
Trace source module of a TMS320C5x or TMS320C6x core delivering program and data trace information.

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.

©1989-2020 Lauterbach GmbH

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 [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**]
CTI.RESET
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.

©1989-2020 Lauterbach GmbH

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.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.

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)

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.

©1989-2020 Lauterbach GmbH

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.

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).

©1989-2020 Lauterbach GmbH

In the last years the chips and its debug and trace architecture became much more complex. Especially the CoreSight trace components and their interconnection on a common trace bus required a reform of our commands. The new commands can deal even with 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**.

©1989-2020 Lauterbach GmbH

… **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.

TRACE... stands for the ADTF trace source module.

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.

©1989-2020 Lauterbach GmbH

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.

Deprecated and 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>*

©1989-2020 Lauterbach GmbH

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)

©1989-2020 Lauterbach GmbH

TIADTFBASE *<address>*	**ADTF.Base** *<address>*
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>*
TRACEETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource ADTF** *<port>* (1)
TRACEFUNNELPORT *<port>*	**FUNNEL1.ATBSource ADTF** *<port>* (1)
TRACETPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource ADTF** *<port>* (1)
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>:	**High** **Low** **HIGHWHENSTOPPED** **LOWWHENSTOPPED**

Default: High.

Controls the WDTDIS pin of the Automotive Debug Cable. This option is only available if an Automotive Debug Cable is connected to the PowerDebug module.

High	The WDTDIS pin is permanently driven high.
Low	The WDTDIS pin is permanently driven low.
HIGHWHEN-STOPPED	The WDTDIS pin is driven high when program is stopped.
LOWWHEN-STOPPED	The WDTDIS pin is driven low when program is stopped.

©1989-2020 Lauterbach GmbH

[Examples]

Format:	**SYStem.CONFIG GICD** *<sub_cmd>*
<sub_cmd>:	**Base** *<address>* **Type GIC400** | **GIC500** | **GICv3** | **GICv4** **RESet**

This command group makes TRACE32-internal configuration settings for the peripheral Generic Interrupt Controller Distributor (GICD) complying with the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*.

After you have selected a SoC with a *known configuration* using the **SYStem.CPU** *<soc>* command, TRACE32 makes its own internal settings for the peripheral *automatically*. You can view the result in the **SYStem.CONFIG.state** window on the **COmponents** tab.

For custom SoCs, you have to make these settings *manually*. To do this, use the **SYStem.CONFIG GICD** command group or the **SYStem.CONFIG.state /COmponents** window.

As soon as the subcommands **Base** and **Type** are configured, the following command groups are activated:

- **SYStem.CONFIG GICR** for the Generic Interrupt Controller Redistributor (GICR)
- **SYStem.CONFIG GICC** for the Generic Interrupt Controller physical CPU interface (GICC)
- **SYStem.CONFIG GICH** for the Generic Interrupt Controller virtual interface control (GICH)
- **SYStem.CONFIG GICV** for the Generic Interrupt Controller virtual CPU interface (GICV)

For information about whether the use of a newly-activated command group is mandatory or optional, see **here**.

Base	Informs the debugger about the physical base address of the peripheral, access class **AD:**
Type	Selects the type or version of the ARM Generic Interrupt Controller (GIC) used in the SoC. • **GIC400** and **GIC500** are implementations from ARM. • **GICv3** and **GICv4** are controllers complying with the GICv3/GICv4 specification from ARM. If the type is configured, the correct GIC register block is shown in the **PER.view** window.
RESet	Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.

©1989-2020 Lauterbach GmbH

Then the TRACE32-internal settings with:	If the type* or version of the Generic Interrupt Controller (GIC) is:			
	GIC400	GIC500	GICv3	GICv4
SYStem.CONFIG GICD are	mandatory	mandatory	mandatory	mandatory
SYStem.CONFIG GICR are	N/A	mandatory	mandatory	mandatory
SYStem.CONFIG GICC are	mandatory	optional	optional	optional
SYStem.CONFIG GICH are	mandatory	optional	optional	optional
SYStem.CONFIG GICV are	mandatory	optional	optional	optional
(*) Type or version are set with **SYStem.CONFIG GICD.Type**.				

Examples

Example 1:

```
;a known SoC is selected from the CPU list
SYStem.CPU IMX8MQ

;the TRACE32-internal settings for the GIC are preset => nothing to do
```

Example 2:

```
;configure custom SoC:
;e.g. Cortex-A53 Quadcore with ARM GIC-500 without legacy interface.
SYStem.CPU CortexA53
SYStem.CONFIG CORENUMBER 4.

SYStem.CONFIG ...

;example GIC Distributor base address: 0x10000000
SYStem.CONFIG GICD Base AD:0x10000000
SYStem.CONFIG GICD Type GIC500

;example GIC Redistributor base addresses: 0x11000000 0x11020000
;0x11040000 0x11060000
SYStem.CONFIG GICR Base AD:0x11000000 AD:0x11020000 AD:0x11040000 \
                                                        AD:0x11060000
```

©1989-2020 Lauterbach GmbH

Example 3:

```
;configure a custom SoC:
;e.g. Cortex-A53 Quadcore with ARM GIC-400
SYStem.CPU CortexA53
SYStem.CONFIG CORENUMBER 4.

SYStem.CONFIG ...

;example GIC Distributor base address: 0x10001000
SYStem.CONFIG GICD Base AD:0x10001000
SYStem.CONFIG GICD Type GIC400

;example GIC CPU interface base addresses (banked): 0x10002000
SYStem.CONFIG GICC Base AD:0x10002000

;example GIC virtual interface control base addresses (banked):
;0x10004000
SYStem.CONFIG GICH Base AD:0x10004000

;example GIC virtual CPU interface base addresses (banked): 0x10006000
SYStem.CONFIG GICV Base AD:0x10006000
```

©1989-2020 Lauterbach GmbH

Format:	**SYStem.CONFIG GICR** *<sub_cmd>*
<sub_cmd>:	**Base** *<address>*… **RESet**

This command group makes TRACE32-internal configuration settings for the peripheral Generic Interrupt Controller Redistributor (GICR) complying with the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*.

- **SYStem.CONFIG GICR** is activated as soon as **Base** and **Type** of the **SYStem.CONFIG GICD** command group have been set.
- The GICR configuration is only valid and mandatory if the **SYStem.CONFIG GICD.Type** is **GICv3**, **GICv4**, or **GIC500**.

For SoCs in the **SYStem.CPU** list that have a *known configuration*, TRACE32 makes its own internal settings for the peripheral *automatically.* You can view the result in the **SYStem.CONFIG.state** window on the **COmponents** tab.

For custom SoCs, you have to make these settings *manually.* To do this, use the **SYStem.CONFIG GICR** command group or the **SYStem.CONFIG.state /COmponents** window.

Base *<address>*…	Informs the debugger about the physical base address of the redistributor for each core, access class **AD:**
RESet	Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.CONFIG GICC** *<sub_cmd>*
<sub_cmd>:	**Base** *<address>*… **RESet**

This command group makes TRACE32-internal configuration settings for the peripheral Generic Interrupt Controller physical CPU interface (GICC) complying with the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*.

- **SYStem.CONFIG GICC** is activated as soon as **Base** and **Type** of the **SYStem.CONFIG GICD** command group have been set.
- The GICC configuration is mandatory if the **SYStem.CONFIG GICD.Type** is **GIC400**.
- The GICC configuration is optional if the **SYStem.CONFIG GICD.Type** is **GICv3**, **GICv4**, or **GIC500**.

For SoCs in the **SYStem.CPU** list that have a *known configuration*, TRACE32 makes its own internal settings for the peripheral *automatically*. You can view the result in the **SYStem.CONFIG.state** window on the **COmponents** tab.

For custom SoCs, you have to make these settings *manually*. To do this, use the **SYStem.CONFIG GICC** command group or the **SYStem.CONFIG.state /COmponents** window.

Base *<address>*	Informs the debugger about the physical base address of the CPU interface for each core, access class **`AD:`** If the address is banked in hardware, it is sufficient to specify only a single address.
RESet	Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.CONFIG GICH** *<sub_cmd>*
<sub_cmd>:	**Base** *<address>*… **RESet**

This command group makes TRACE32-internal configuration settings for the peripheral Generic Interrupt Controller virtual interface control (GICH) complying with the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*.

- **SYStem.CONFIG GICH** is activated as soon as **Base** and **Type** of the **SYStem.CONFIG GICD** command group have been set.
- The GICH configuration is mandatory if the **SYStem.CONFIG GICD.Type** is **GIC400**.
- The GICH configuration is optional if the **SYStem.CONFIG GICD.Type** is **GICv3**, **GICv4**, or **GIC500**.

For SoCs in the **SYStem.CPU** list that have a *known configuration*, TRACE32 makes its own internal settings for the peripheral *automatically*. You can view the result in the **SYStem.CONFIG.state** window on the **COmponents** tab.

For custom SoCs, you have to make these settings *manually*. To do this, use the **SYStem.CONFIG GICH** command group or the **SYStem.CONFIG.state /COmponents** window.

Base	Informs the debugger about the physical base address of the virtual interface control for each core, access class **`AD:`** If the address is banked in hardware, it is sufficient to specify only a single base address.
RESet	Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.CONFIG GICV** *<sub_cmd>*
<sub_cmd>:	**Base** *<address>*… **RESet**

This command group makes TRACE32-internal configuration settings for the peripheral Generic Interrupt Controller virtual CPU interface (GICV) complying with the *ARM® Generic Interrupt Controller Architecture Specification, GIC architecture version 3.0 and version 4.0*.

- **SYStem.CONFIG GICV** is activated as soon as **Base** and **Type** of the **SYStem.CONFIG GICD** command group have been set.
- The GICV configuration is mandatory if the **SYStem.CONFIG GICD.Type** is **GIC400**.
- The GICV configuration is optional if the **SYStem.CONFIG GICD.Type** is **GICv3**, **GICv4**, or **GIC500**.

For SoCs in the **SYStem.CPU** list that have a *known configuration*, TRACE32 makes its own internal settings for the peripheral *automatically*. You can view the result in the **SYStem.CONFIG.state** window on the **COmponents** tab.

For custom SoCs, you have to make these settings *manually*. To do this, use the **SYStem.CONFIG GICV** command group or the **SYStem.CONFIG.state /COmponents** window.

Base	Informs the debugger about the physical base address of the virtual CPU interface for each core, access class **AD:** If the address is banked in hardware, it is sufficient to specify only a single base address.
RESet	Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.

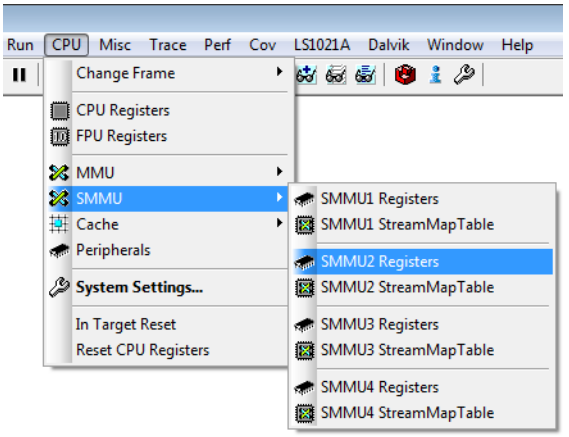
©1989-2020 Lauterbach GmbH

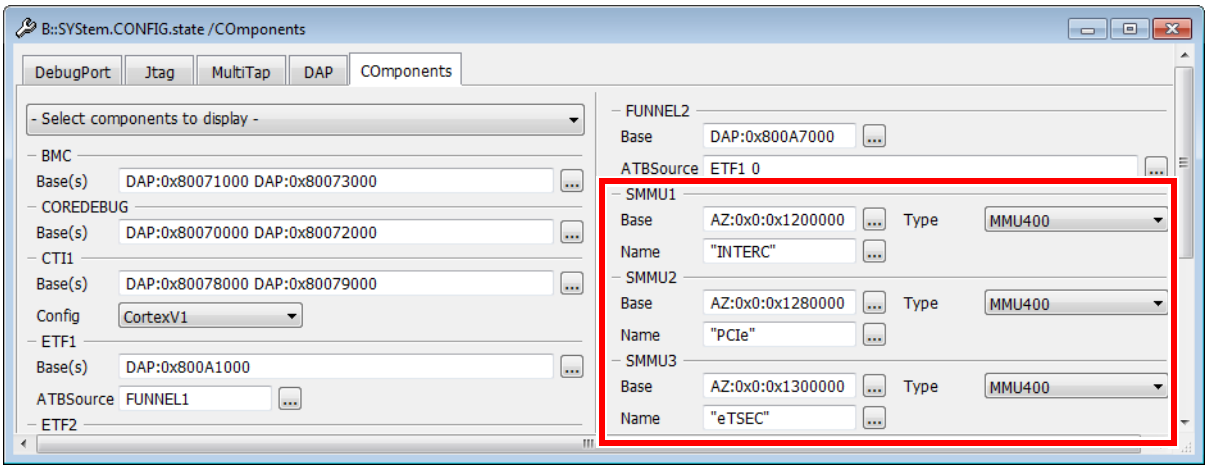
Format:	**SYStem.CONFIG SMMU** *<x> <sub_cmd>*
<x>:	**1** … **20**
<sub_cmd>:	**Base** *<base_address>* **Type MMU400** | **MMU401** | **MMU500** **Name** "*<name>*" **RESET**

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.





©1989-2020 Lauterbach GmbH

<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>*.

©1989-2020 Lauterbach GmbH

SYStem.CPU — Select the used CPU

[Go to figure]

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**CortexA32** | **CortexA34** | **CortexA35** | **CortexA53** | **CortexA55** | **CortexA57** | **CortexA65** | **CortexA72** | **CortexA73** | **CortexA75** | **CortexA76** | **CortexA76AE** | **CortexA77** | **CortexA57A53** | **CortexA72A53** | **CortexA73A53** | **CortexR52** | **NeoverseE1** | **NeoverseN1**

Selects the processor type. If your CPU is not listed, use **SYStem.CPU ARMV8-A**.

CORTEXA53	Is the default selection if the Debugger for Cortex-A/R (ARMv8, 32/64-bit) is used.

NOTE:	Only Cortex-A/R CPUs are shown in the example selection. There are other CPUs in the selection as well. Use `SYStem.CPU *` to get a selection list.

SYStem.JtagClock — Define the frequency of the debug port

[Go to figure]

Format:	**SYStem.JtagClock** [*<frequency>* | **CTCK** *<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.

©1989-2020 Lauterbach GmbH

The clock frequency affects e.g. the download speed. It could be required to reduce the frequency if there are buffers, additional loads or high capacities on the debug 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 this 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.
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 (Compensation by TCK). This feature can be used with a debug cable version 4b 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).

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

[Go to figure]

Format:	**SYStem.MemAccess** *<mode>*
<mode>:	**AHB** | **AXI** | … (CoreSight v3) **DAP** (CoreSight v2) **Enable** **Denied** **StopAndGo**

Default: Denied.

Allows to select a method for memory access while the CPU is running. 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**, …
Depending on which memory access ports are available on the chip, the memory access is done through the specified bus.

DAP
For CoreSight v3, **DAP** must not be used anymore.
A run-time memory access is done via the ARM CoreSight v2 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.

Denied
No memory access is possible while the CPU is executing the program.

Enable
CPU (deprecated)
Used to activate the memory access while the CPU is running on the TRACE32 Instruction Set Simulator and on debuggers which do not have a fixed name for the memory access method.

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”.

©1989-2020 Lauterbach GmbH

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

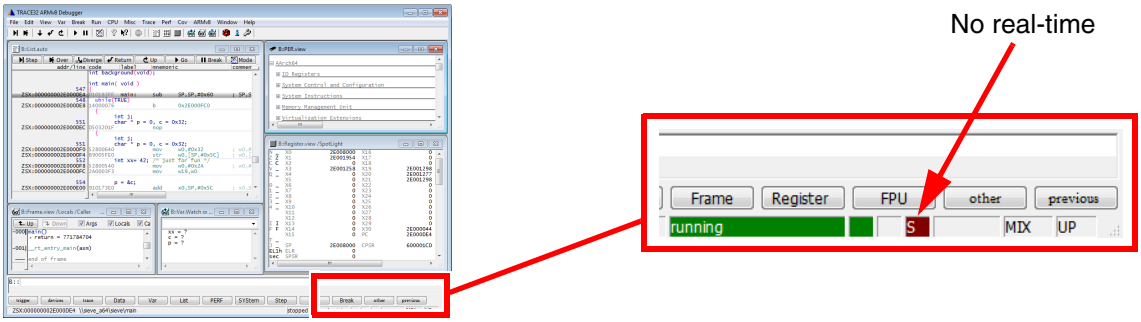
SYStem.CpuSpot Enabled				
SYS.MA. Access class	**Denied**	**DAP (SoC-400 only)**	**[AHB | AXI] (SoC-600 only)**	**StopAndGo**
EA	no access	AHB/AXI	AHB/AXI	CPU* (spotted)
A	no access	no access	no access	no access
AHB or AXI	no access	no access	no access	no access
EAHB or EAXI	AHB/AXI	AHB/AXI	AHB/AXI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

©1989-2020 Lauterbach GmbH

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



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

```
Data.dump E:0x100

Var.View %E first
```

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

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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

SYStem.Option Address32 — Define address format display

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

Default: AUTO.

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

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

SYStem.Option AHBHPROT — Select AHB-AP HPROT bits

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

Default: 0

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

©1989-2020 Lauterbach GmbH

SYStem.Option AXI32 — Use 32-bit atomic AXI accesses instead of 64-bit

Format:	**SYStem.Option AXI32** [**ON** | **OFF**]

Default: OFF

If set to ON, 64-bit atomic AXI accesses will be converted to 32-bit atomic AXI accesses. The option can be set when the AXI of a SoC does not support 64-bit atomic accesses.

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

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

Default: OFF.

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

SYStem.Option AXICACHEFLAGS — Select AXI-AP CACHE bits

Format:	**SYStem.Option AXICACHEFLAGS** *<value>*
<value>:	**DEVICENONSHAREABLE** **DEVICEINNERSHAREABLE** **DEVICEOUTERSHAREABLE** **DeviceSYStem** **NonCacheableNonShareable** **NonCacheableInnerShareable** **NonCacheableOuterShareable** **NonCacheableSYStem** **WriteThroughNonShareable** **WriteThroughInnerShareable** **WriteBackOuterShareable** **WRITETHROUGHSYSTEM**

Default: 0

©1989-2020 Lauterbach GmbH

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

DEVICENONSHAREABLE	CSW.CACHE = 0x0, CSW.DOMAIN = 0x0
DEVICEINNERSHAREABLE	CSW.CACHE = 0x1, CSW.DOMAIN = 0x1
DEVICEOUTERSHAREABLE	CSW.CACHE = 0x1, CSW.DOMAIN = 0x2
DeviceSYStem	CSW.CACHE = 0x1, CSW.DOMAIN = 0x3
NonCacheableNonShareable	CSW.CACHE = 0x2, CSW.DOMAIN = 0x0
NonCacheableInnerShareable	CSW.CACHE = 0x3, CSW.DOMAIN = 0x1
NonCacheableOuterShareable	CSW.CACHE = 0x3, CSW.DOMAIN = 0x2
NonCacheableSYStem	CSW.CACHE = 0x3, CSW.DOMAIN = 0x3
WriteThroughNonShareable	CSW.CACHE = 0x6, CSW.DOMAIN = 0x0
WriteThroughInnerShareable	CSW.CACHE = 0xA, CSW.DOMAIN = 0x1
WriteBackOuterShareable	CSW.CACHE = 0xE, CSW.DOMAIN = 0x2
WRITETHROUGHSYSTEM	CSW.CACHE = 0xE, CSW.DOMAIN = 0x3

SYStem.Option AXIHPROT — Select AXI-AP HPROT bits

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

Default: 0

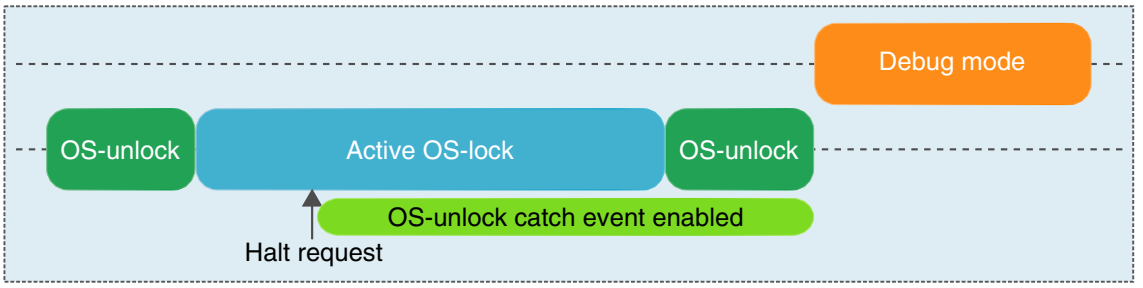
This option selects the value used for the HPROT bits in the Control Status Word (CSW) of an AXI Access Port of a DAP, when using the AXI: memory class.

©1989-2020 Lauterbach GmbH

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

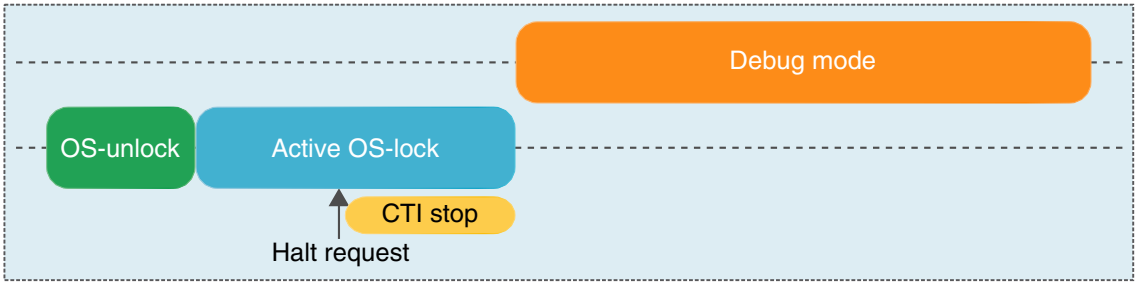
Default: OFF.

A CPU that is in OS-lock mode shall usually not stop when **Break.direct** is executed. This is the case when **SYStem.Option BreakOS** is **OFF**. The debugger will make sure that the CPU remains running and will enable an OS-unlock catch event to stop the CPU in debug mode as soon as it gets unlocked.



Break with **SYStem.Option BreakOS OFF** (default)

This behavior might not always be applicable. Some use cases require the CPU to stop while it is locked. Set **SYStem.Option BreakOS** to **ON**. The debugger will use a CTI stop event to halt the CPU in locked mode. The locked state is cached by the debugger and restored on a **Go** or **Step**. The user can change the cached lock state by writing to OSLAR manually (e.g. via the **PER** file).



Break with **SYStem.Option BreakOS ON**

To control the OS-unlock catch event for all cores, see also **TrOnchip.state**.

©1989-2020 Lauterbach GmbH

SYStem.Option CacheStatusCheck — Check status bits during cache access

Format:	**SYStem.Option CacheStatusCheck** [**ON** | **OFF**] (deprecated) **Replaced by auto-detection.**

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

[SYStem.state window > CFLUSH]

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

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 CLTAPKEY — Set key values for CLTAP operation

Format:	**SYStem.Option CLTAPKEY** *<key0> <key1> <key2> <key3>*

Default: *<key[n]>* = 0x00000000

Defines key values that are shifted by the debugger during an implementation defined CTTAP operation. All key values are shifted in a row during a single 128-bit DR-shift starting with *<key0>* and ending with *<key3>*.

<key[n]>	The key value that is shifted. The shift if a single key value starts with its LSB (little endian representation) and ends with the MSB.

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

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

Default: OFF.

©1989-2020 Lauterbach GmbH

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

SYStem.Option CTITimerStop — Stop system timer when CPU stops

Format:	**SYStem.Option CTITimerStop** [**OFF** | **ZYNQULTRASCALE** | **SR6P7**]

Default: OFF.

If set, the CTI of the device will be configured to start / stop the timer(s) when the CPU starts / stops.

OFF	CTI will not be configured for timer stop.
ZYNQULTRASCALE	The system timestamp generator (IOU_SCNTRS) is stopped. Other timers are not stopped. Requires that the stop event is also enabled in the system timer control register. **Note**: There is a delay between CPU and timer (re)start.
SR6P7	Every timer that is configured to stop in debug mode will be stopped when the CPU stops and restart when the CPU restarts. Configuration is required in corresponding timer control register.

SYStem.Option DAPDBGPWRUPREQ — Force debug power in DAP

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

Default: ON.

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

©1989-2020 Lauterbach GmbH

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

Use case:

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

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

SYStem.Option DAP2DBGPWRUPREQ — Force debug power in DAP2

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

Default: ON.

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

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

©1989-2020 Lauterbach GmbH

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 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.

©1989-2020 Lauterbach GmbH

SYStem.Option DAPSYSPWRUPREQ — Force system power in DAP

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

Default: ON.

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

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

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

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

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

SYStem.Option DAP2SYSPWRUPREQ — Force system power in DAP2

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

Default: ON.

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

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

©1989-2020 Lauterbach GmbH

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 DBGSPR — Use debugger view for SPR access

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

Default: OFF.

SPR register are always accessed with debugger privileges. The current CPU mode is ignored.

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.

SYStem.Option DCacheMaintenance — Data cache maintenance strategy

Format: **SYStem.Option DCacheMaintenance** [**SetWay** | **SetWayVA** | **VA**]

Default: VA (Virtual Address).

©1989-2020 Lauterbach GmbH

Determines which kind of cache maintenance instructions are used to clean or invalidate the data cache, e.g. during a physical access. It is recommended to either use **VA** or **SetWay**.

SetWay	Iterate over all cache levels and set/way tuples.
SetWayVA	Handle separate caches by VA to PoC and unified caches by Set/Way.
VA	Handle all caches by VA to PoC (Point of Coherency).

NOTE:	The **CACHE.FLUSH**, **CACHE.CLEAN** and **CACHE.INVALIDATE** commands are not affected by this option. They always operate on the entire cache.

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.

©1989-2020 Lauterbach GmbH

DormantToSwd	Switching procedure which is required if the device starts up in dormant state. The device has a dormant state but does not support JTAG.
JtagToDormantToSwd	Switching procedure as it is required on SWJ-DP with a dormant state. The device is in JTAG mode after power-up.

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

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

SYStem.Option DIAG — Activate more log messages

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

Default: OFF.

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

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

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

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

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

©1989-2020 Lauterbach GmbH

[Go to figure]

Format:	**SYStem.Option DisMode** *<option>*
<option>:	**AUTO** **ACCESS** **ARM** **THUMB** **AARCH64**

Default: AUTO.

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 and RW bit in the CPSR or on the selected access class. (e.g. `List SR:0` for ARM mode or `List ST:0` for THUMB mode).
ARM	Only the ARM disassembler is used (highest priority*, AArch32 only).
THUMB	Only the THUMB disassembler is used (highest priority*, AArch32 only).
AARCH64	Only the AARCH64 disassembler is used (highest priority*).
THUMBEE	Only the THUMB disassembler is used which supports the Thumb-2 Execution Environment extension (highest priority).

NOTE: Highest priority in this context means that this setting will overwrite all other symbol information that refer to the kind of opcode used. Which instruction set is used by the CPU can also be contained in the debug information of the file loaded. Or it can be encoded in the access class, like **List.auto X**:*<address>*, **List.auto R**:*<address>* or **List.auto T**:*<address>*, see also **Access Classes**.

©1989-2020 Lauterbach GmbH

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 eXclusiveMONitor — Support for exclusive monitors

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

Default: OFF.

Some CPU architectures might clear the exclusive monitor state when the CPU returns from debug mode to running mode. As a result, loops containing exclusive store operations might behave incorrectly during debug. When this option is set to on, the debugger tries to evaluate the result of the exclusive store and to correct the state of the exclusive monitor if possible.

This option was implemented on a customers request and should be **OFF** per default.

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.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.Option ICacheMaintenance** [**IALLU** | **IALLUIS** | **OFF**] **SYStem.Option CFLUSH** [**ON** | **OFF**] (deprecated)

Default: IALLU

Determines which kind of cache maintenance instructions are used to invalidate the instruction cache before the CPU restarts execution, e.g. before a **Go** or **Step**.

IALLU	Invalidate all instruction caches to Point of Unification (PoU).
IALLUIS	Invalidate all instruction caches in the Inner Shareable domain to Point of Unification (PoU).
IVAU	Invalidate instruction cache upon a program write, e.g. **Data.Assemble** *<address>* or **Break.Set** *<address>* **/SOFT**, by *<address>*. Instruction cache invalidation before a **Go** or **Step** is not performed.
OFF	Perform no instruction cache invalidation before CPU restart or during program write (e.g. **Data.Assemble**)

NOTE:	The **CACHE.FLUSH IC** and **CACHE.INVALIDATE IC** commands are not affected by this option. They always use IALLU.

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.

©1989-2020 Lauterbach GmbH

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

[SYStem.state window > INTDIS]

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

Default: OFF.

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

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

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

Default: OFF.

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

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

©1989-2020 Lauterbach GmbH

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

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.

If **SYStem.Option.MACHINESPACES** is set to **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.

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.

SYStem.Option MEMORYHPROT Select memory-AP HPROT bits

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

Default: 0

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

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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 Glossary"** (glossary.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
```

©1989-2020 Lauterbach GmbH

Format:	**SYStem.Option NOMA** [**OFF** | **Read** | **Write** | **ReadWrite** | **Quad**] **SYStem.Option NOMA** [**ON** | **OFF**] (deprecated)

Default: OFF.

Per default the debugger will try to access larger memory blocks using an optimized memory access mode of the CPU. If this is not desired, the Debugger may use an alternative memory access when this option is not **OFF**. This can be used as a workaround e.g. when the default memory access does not work with a target CPU. The memory access will become significantly slower. It is recommended not to use this option during normal debugger operation.

OFF	Use optimized memory access mode of the CPU.
Read	Use alternative memory access only for read accesses.
Write	Use alternative memory access only for write accesses.
ReadWrite	Use alternative memory access only for read and write accesses.
Quad	Perform an atomic 64-bit read/write access even when accessing larger blocks and don't use the optimized memory access mode of the CPU.

NOTE:	If the memory access width is byte or word the optimized memory access mode will never be used.

©1989-2020 Lauterbach GmbH

SYStem.Option NoPRCRReset — Disable warm reset via PRCR

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

Default: ON.

The debugger tries to do a warm reset, (e.g. **SYStem.Up**) when this option is set to **OFF**.

NOTE:	It is SoC specific whether the warm reset resets the core only or additional modules. Please consider: Peripherals might not be reset. Code execution from the reset vector might therefore happen in a different context compared to a power cycle of the SoC or a system wide reset, e.g. by asserting the reset line.

SYStem.Option OSUnlockCatch — Use the "OS Unlock Catch" debug event

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

Default: ON.

The debugger uses the OS Unlock Catch debug event in order to halt the target during **SYStem.Up** when this option is set to **ON**. This option has no effect on the settings of **TrOnchip.Set**.

©1989-2020 Lauterbach GmbH

SYStem.Option OVERLAY — Enable overlay support

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

Default: OFF.

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

OFF Disables support for code overlays.

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

Example:

```
SYStem.Option OVERLAY ON
Data.List 0x2:0x11c4                    ; Data.List <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**.

©1989-2020 Lauterbach GmbH

SYStem.Option PWRDWN — Allow power-down mode

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

Default: OFF.

Controls the CORENPDRQ bit in device power-down and reset control register (PRCR). This forces the system power controller in emulate mode.

SYStem.Option PAN — Overwrite CPSR.PAN setting

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

Default: OFF.

This option is effective since ARMv8.1. It is not effective for ARMv8.0. It controls how the debugger handles the Privileged Access Never (PAN) bit in CPSR while the CPU is in debug (stopped) mode. The debugger will restore the PAN setting, if modified, when the CPU returns to running state.

ON	The debugger will clear the CPSR.PAN bit temporarily in debug mode. Memory access are handled in the same fashion as for ARMv8.0
OFF	The debugger ignores the CPSR.PAN bit. All memory accesses are handled according to the PAN setting introduced with ARMv8.1

SYStem.Option PWRREQ — Request core power

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

Default: OFF.

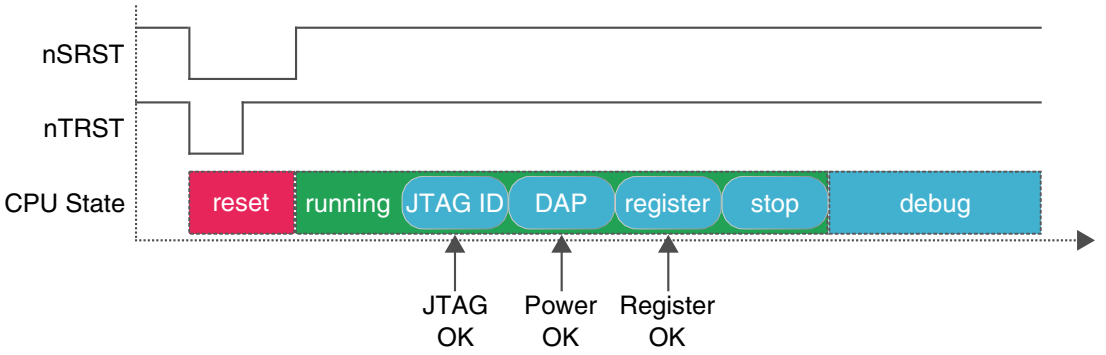
Controls the COREPURQ bit in device power-down and reset control register (PRCR). When this option is set to **OFF** the debugger will set this bit to 0. The debugger requests core power when this option is set to **ON**. Implemented on customers demand. Usually this option is not needed.

©1989-2020 Lauterbach GmbH

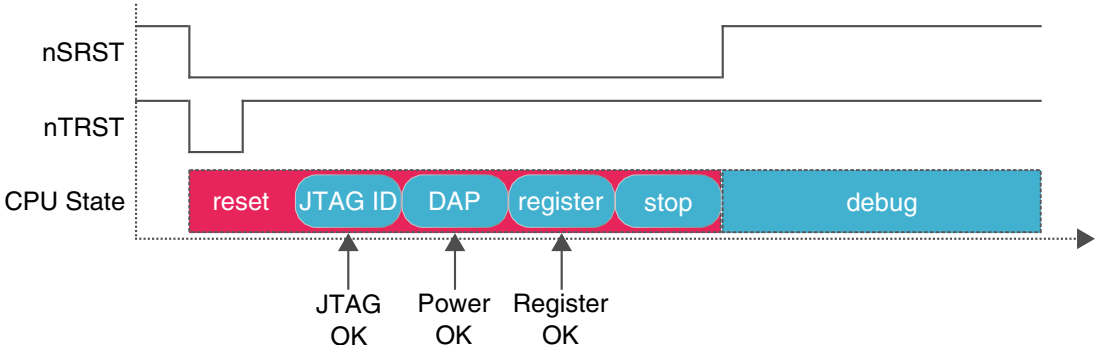
Format:	**SYStem.Option ResBreak** [**ON** | **OFF**]

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.



©1989-2020 Lauterbach GmbH

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.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.

©1989-2020 Lauterbach GmbH

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).

NOTE: The debugger will not perform the default warm reset via the PRCR if this option is set.

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.

©1989-2020 Lauterbach GmbH

SYStem.Option SLaVeSOFTRESet — Allow soft reset of slave cores

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

Default: OFF.

Allow the debugger to do a soft reset of a slave core during **SYStem.Up**. Only set to **ON** when reset event on a slave core is not distributed to other cores, e.g. by a reset controller. However, the reset will not be done if **SYStem.Option RESetREGister** is set up, i.e. the option is ignored.

SYStem.Option SMPMultipleCall — Send start event to each SMP core

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

Default: OFF.

When set to **ON**, the debugger will send a start pulse to each core via the CTI in an SMP system. This is a bugfix for systems with a broken CTI event distribution. Implemented on customers' request.

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

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

Default: OFF.

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

SYStem.Option SOFTQUAD — Use 64-bit access to set breakpoint

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.

©1989-2020 Lauterbach GmbH

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 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 TURBO Disable cache maintenance during memory access

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

Default: OFF.

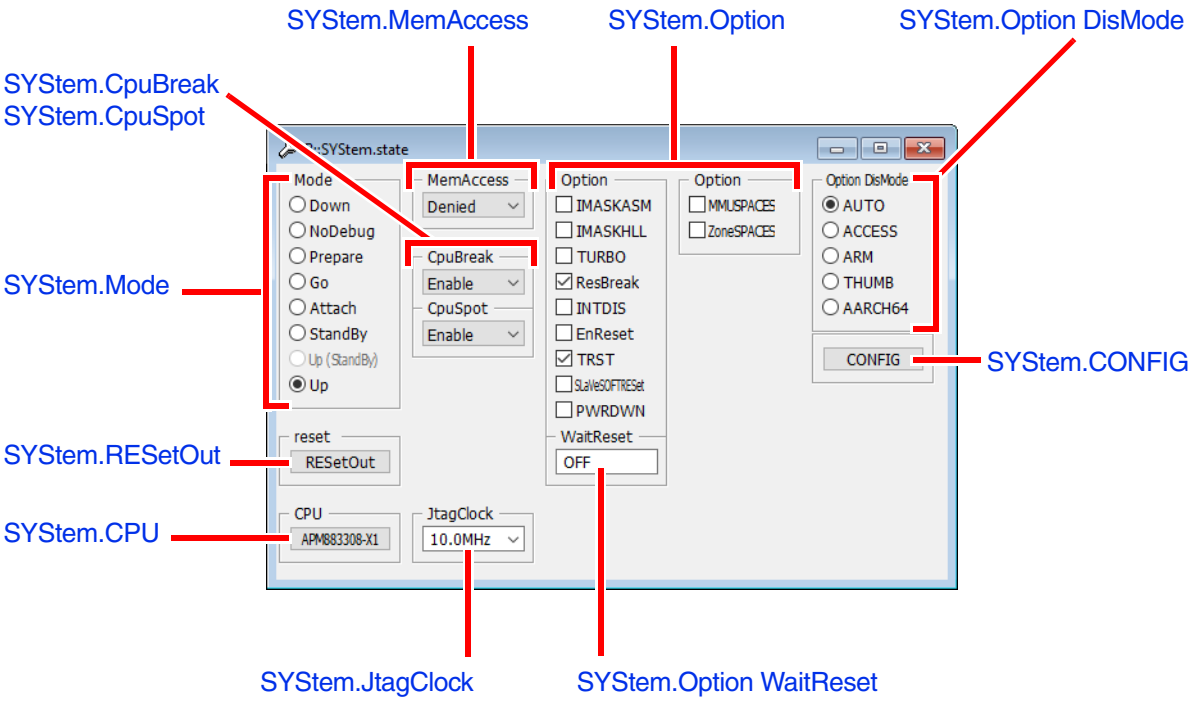
The debugger will not perform any cache maintenance during memory accesses. This will speed up memory accesses. However, the IC invalidation before a **Step** or **Go** may still be done. See **SYStem.Option ICacheMaintenance** for controlling this kind of cache maintenance.

©1989-2020 Lauterbach GmbH

Format:	**SYStem.state**

Displays the **SYStem.state** window for ARM. Offers a GUI for system settings that configure debugger and target behavior. All these settings are described in the sections of this chapter. Each link is a valid command that can be entered in the TRACE32 command line.

Click a link to navigate directly to the corresponding section.



SYStem.Option SYSPWRUPREQ Force system power

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

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).

©1989-2020 Lauterbach GmbH

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 WaitDAPPWR Wait for DAP power after DAP power request

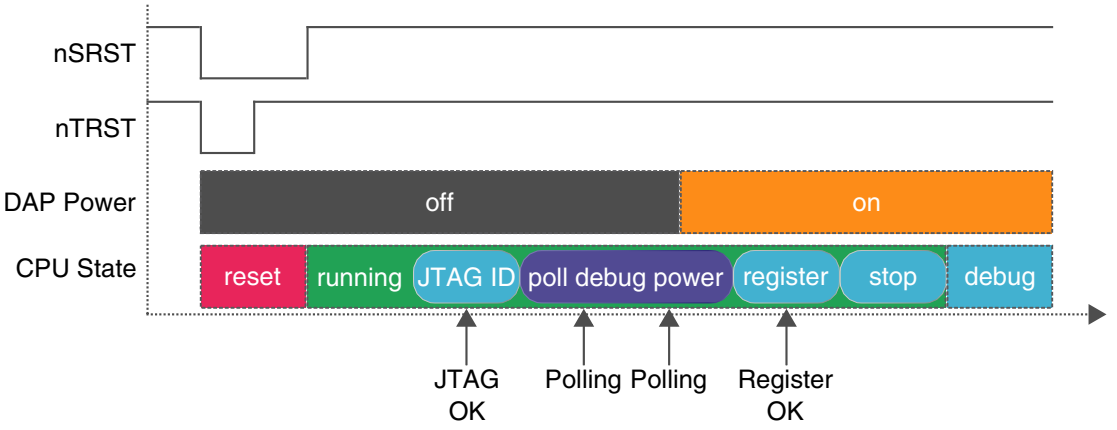
Format:	**SYStem.Option WaitDAPPWR** [**ON** | **OFF** | *<time>*]

Default: OFF = disabled.

Allows to add an additional wait time after a power up request of the ARM CoreSight subsystem (DAP).

ON	1 second default wait time
OFF	50 ms default wait time
<time>	Custom wait time, max. 30 sec, use ‘us’, ‘ms’, ‘s’ as units.

Example: The following figure shows a situation in which JTAG is available in reset, but the CoreSight subsystem (DAP) needs a certain time until it powers up due to the debuggers power request:



©1989-2020 Lauterbach GmbH

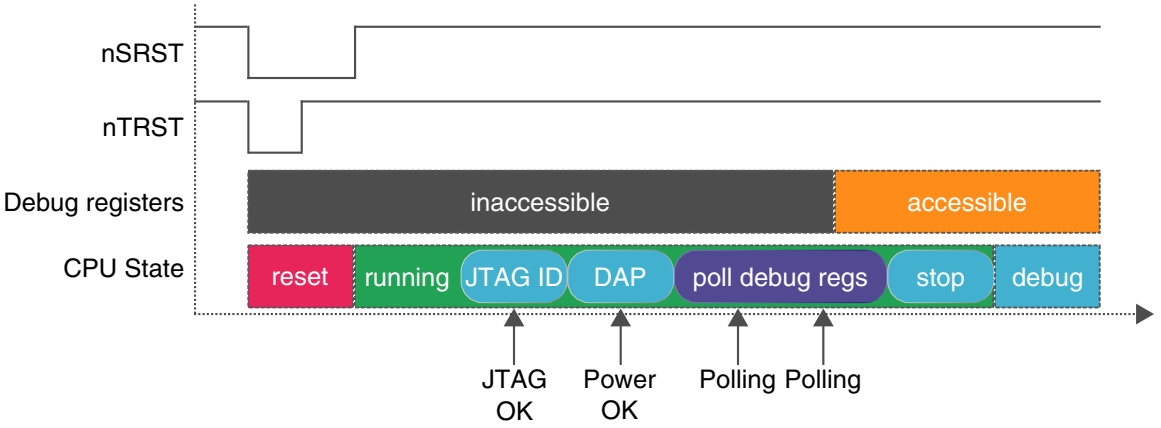
Format:	**SYStem.Option WaitDBGREG** [**ON** | **OFF** | *<time>*]

Default: OFF = disabled.

Adds an additional wait time after reset has been asserted to wait for core debug registers availability.

ON	1 second default wait time
OFF	50 ms default wait time
<time>	Custom wait time, max. 30 sec, use 'us', 'ms', 's' as units.

Example: The following figure shows a situation in which JTAG and the CoreSight subsystem (DAP) are available, but the SoC needs a certain time until the core debug registers become available:



In this example, the nSRST line is kept asserted while the debugger waits for the core debug registers to become available. However, it might be required to release the reset early for this to happen. In this case you should additionally use **SYStem.Option ResBreak OFF**.

©1989-2020 Lauterbach GmbH

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

Default: OFF = disabled.

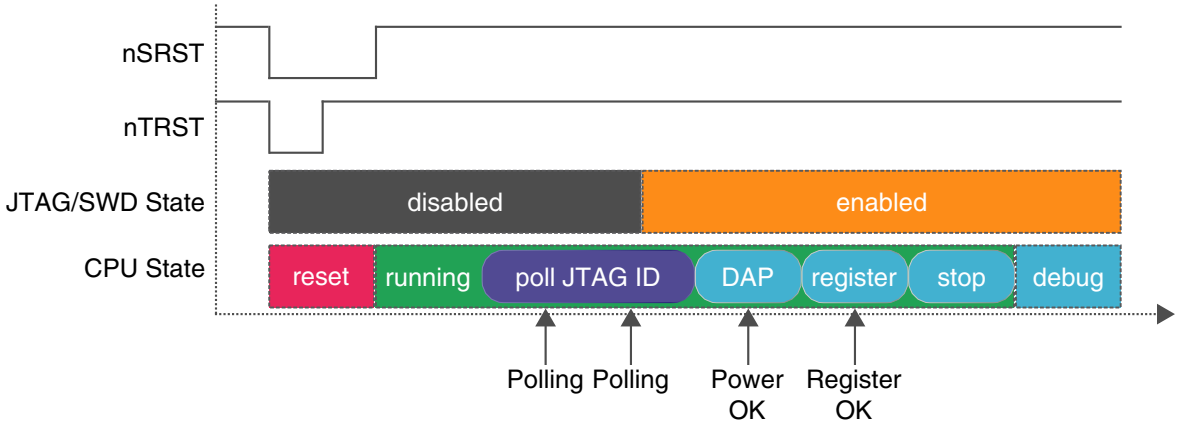
Allows to add additional busy time after reset. The command is limited to systems that use an ARM DAP.

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.



©1989-2020 Lauterbach GmbH

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**.

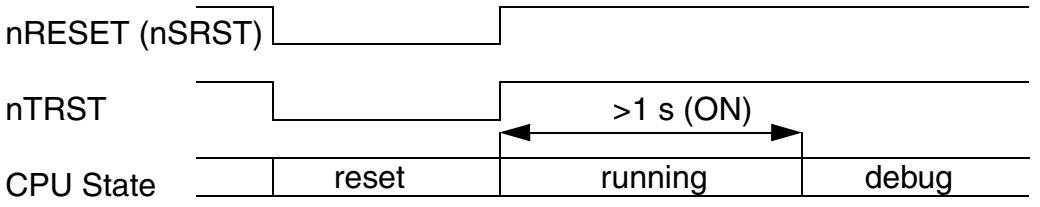
If **SYStem.Option ResBreak** is disabled, **SYStem.Option WaitReset** can be used to specify a waiting time between the deassertion of nSRST and nTRST and the first JTAG activity. During this time the core may execute some code, e.g to enable the JTAG port.

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** is *<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.



©1989-2020 Lauterbach GmbH

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 Glossary"** (glossary.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.

©1989-2020 Lauterbach GmbH

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**.

©1989-2020 Lauterbach GmbH

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:)
```

©1989-2020 Lauterbach GmbH

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
```

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

STATE.NOCTIACCESS()

[build 122395 - DVD 09/2020]

Syntax: **STATE.NOCTIACCESS()**

Returns whether the CTI related to the CPU can be accessed or not.

Return Value Type: Boolean.

Return Value and Description:

TRUE	The CPU is running and the CTI cannot be accessed
FALSE	The CPU is running or stopped and the CTI can be accessed

STATE.NOCPUACCESS()

[build 122395 - DVD 09/2020]

Syntax: **STATE.NOCPUACCESS()**

Returns whether the debug registers related to the CPU can be accessed or not.

Return Value Type: Boolean.

Return Value and Description:

TRUE	The CPU is running and the debug registers cannot be accessed
FALSE	The CPU is running or stopped and the debug registers can be accessed

©1989-2020 Lauterbach GmbH

The **BMC** (**B**ench**M**ark **C**ounter) commands provide control and usage of the on-chip benchmark and performance counters if available on the chip.

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.

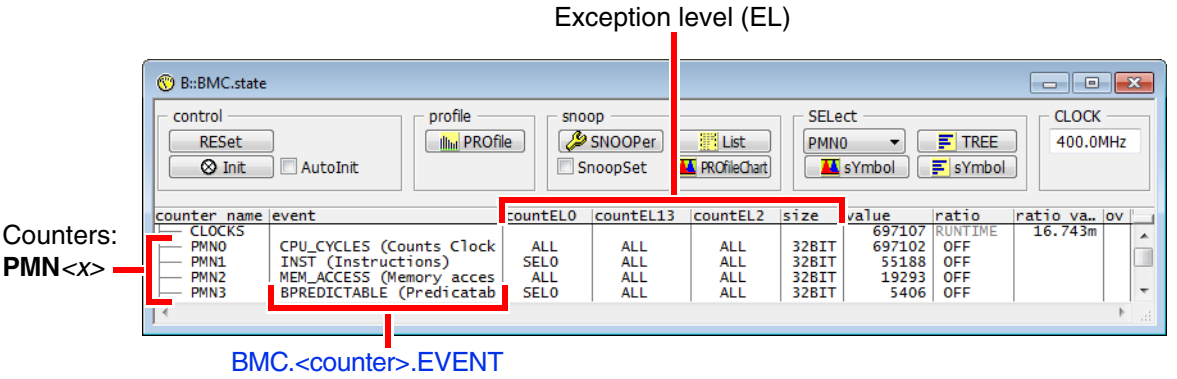
BMC.<counter>.CountEL<x> Select exception level events to be counted

Format: **BMC.***<counter>***.CountEL***<x>* *<option>*

<counter>: **PMN***<x>*

<option>:
DISable
ALL
ALLEL1NSEL3
NSEL0
NSEL1
NSEL2
NSEL1SEL3
SEL0
SEL1ALLEL3
SEL1NSEL3
SEL2
SEL3

Selects the exception level events to be counted.



©1989-2020 Lauterbach GmbH

Available counters:

PMN<*x*>	Counter to be configured. Number of counters is chip specific. Examples: PMN0, PMN1, PMN2 etc.

Select exception level (EL) that shall count certain events

Armv8-A (with TrustZone):

CountEL0	Configure secure/non-secure events to be counted for EL0.
CountEL13	Configure secure/non-secure events to be counted for EL1 and EL3.
CountEL2	Configure secure/non-secure events to be counted for EL2.

Select secure/non-secure events that shall be counted on a certain exception level. The availability of these options depends on the selected **CountEL**<*x*>.

DISable	Disable counting of all events for this <*counter*>.
ALL	Count all events on all exception levels with <*counter*>.
ALLEL1NSEL3	Count all EL1 and non-secure EL3 events with <*counter*>.
NSEL0	Count non-secure EL0 events with <*counter*>.
NSEL1	Count non-secure EL1 events with <*counter*>.
NSEL1SEL3	Counter non-secure EL1 events and secure EL3 events with <counter>
SEL0	Count secure EL0 events with <*counter*>.
SEL1ALLEL3	Count secure EL1 events and all EL3 events with <*counter*>.
SEL1NSEL3	Count secure EL1 events and non-secure EL3 events with <*counter*>.
SEL3	Count secure EL3 events with <*counter*>.

Example cores: Cortex-A3x, Cortex-A5x, Cortex-A7x

Armv8.4-A cores, or later (additional options)

NSEL2	Count non-secure events of EL2. Armv8.4 only.
SEL2	Count secure events of EL2. Armv8.4 only.

©1989-2020 Lauterbach GmbH

Armv8-R (without TrustZone):

CountEL0	Configure events to be counted for EL0.
CountEL1	Configure events to be counted for EL1.
CountEL2	Configure events to be counted for EL2.

Select whether events on a certain exception level shall be counted or not.

DISable	Disable counting of all events for this *<counter>*.
ALL	Count all events on all exception levels with *<counter>*.

Example core: Cortex-R52

Example:

```
BMC.PMN0.EVENT INST       ; Count instructions ...
BMC.PMN0.CountEL13 SEL3   ; ... in secure EL3 only with PMN0
```

Function

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

BMC.EXPORT Export benchmarking events from event bus

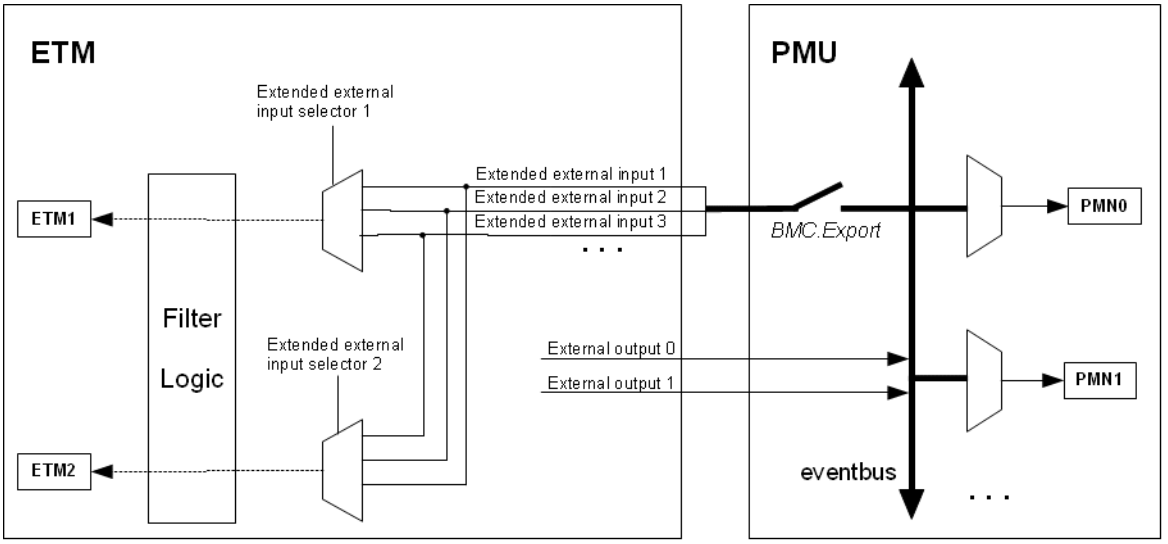
Format:	**BMC.EXPORT** [**ON** | **OFF**]

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

©1989-2020 Lauterbach GmbH

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.

BMC.LongCycle — Configure cycle counter width

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

Configures if the Cycle Counter (clocks) uses a 32-bit or 64-bit register width for counting.

If an Armv8 implementation is AArch64 only, the counter is always 64-bit.

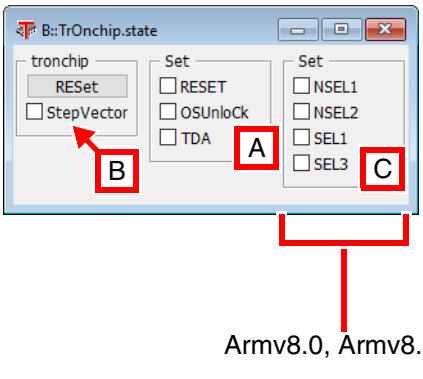
©1989-2020 Lauterbach GmbH

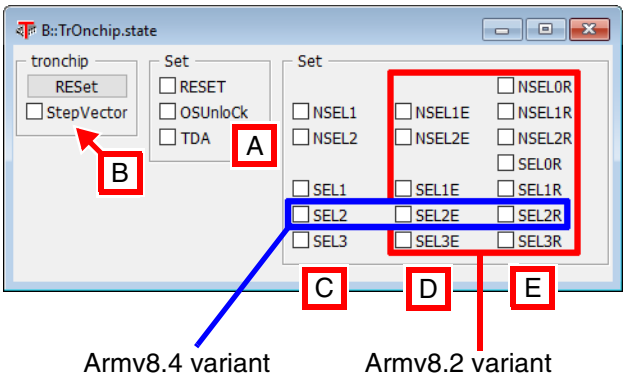
Format:	**BMC.PRESCALER** [**ON** | **OFF**]

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.

ARM specific TrOnchip Commands

The **TrOnchip** command group provides low-level access to the on-chip debug register. For configuring the low-level access, use the TRACE32 command line or the **TrOnchip.state** window:





A **TrOnchip.Set** If enabled, the program execution is stopped at the specified special event.

B **TrOnchip.Set** Enable/Disable stepping of exception handlers.

C **TrOnchip.Set** If enabled, the program execution stops on an exception level change.

D **TrOnchip.Set** If enabled, the program execution stops on an exception level entry from a lower exception level, Armv8.2 only, secure EL2 extension Armv8.4 only.

E **TrOnchip.Set** If enabled, the program execution stops on an exception level return from a higher execption level, Armv8.2 only, secure EL2 extension Armv8.4 only.

Special events are the reset, the removal of the OS-lock state or the access to debug registers (TDA).

Events that trigger the halt on an exception level change cannot distinguish the AArch32 modes abt, und, fiq, irq, svc, sys. Those modes are all covered by the option SEL1 or NSEL1. To stop on a specific AArch32 exception vector (e.g. fiq, irq) do not use the **TrOnchip.state** command. Set a single address breakpoint on the exception vector address instead.

©1989-2020 Lauterbach GmbH

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.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.

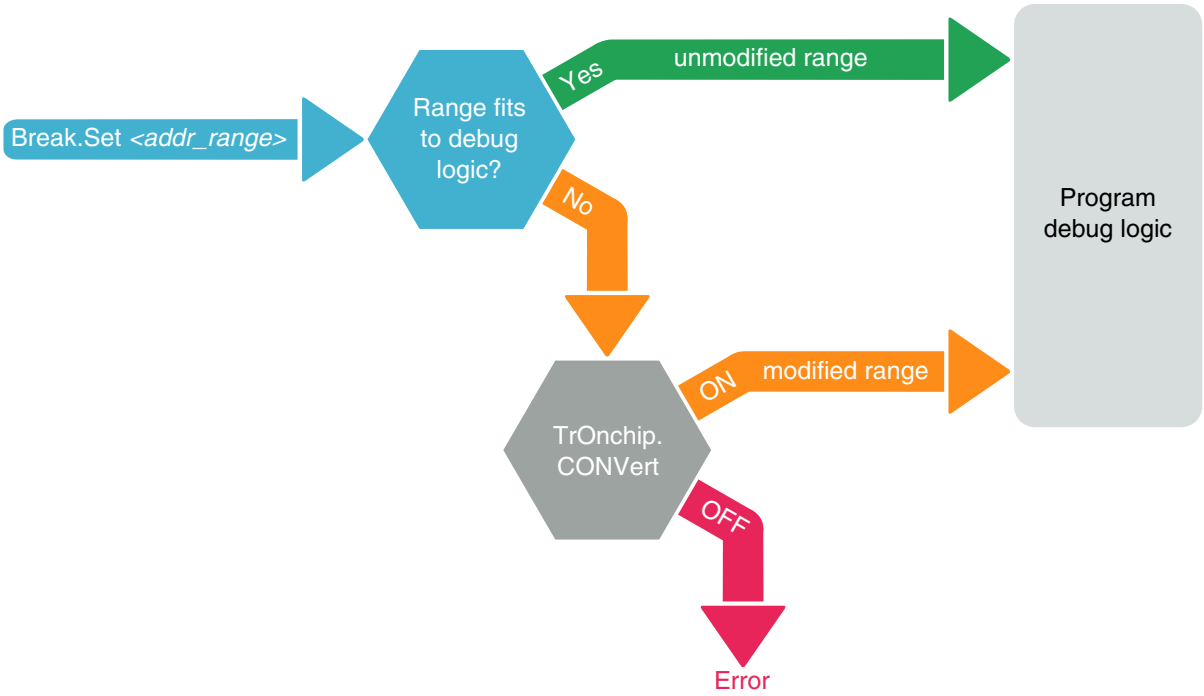
Example:

```
TrOnchip.ContextID ON

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

©1989-2020 Lauterbach GmbH

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).

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

TrOnchip.MatchASID Extend on-chip breakpoint/trace filter by ASID

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.

©1989-2020 Lauterbach GmbH

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
```

©1989-2020 Lauterbach GmbH

TrOnchip.RESERVE Exclude breakpoint or watchpoint from debugger usage

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

Default: OFF.

Selects breakpoints or watchpoints that will not be used by the debugger. This allows to do a custom configuration of breakpoint or watchpoint registers.

BP<*x*>	Selects whether breakpoint <*x*> can be used by the debugger. • **OFF**: Debugger can configure and use breakpoint <*x*>. • **ON**: Debugger shall not configure and overwrite breakpoint <*x*>.
WP<*x*>	Selects whether watchpoint <*x*> can be used by the debugger. • **OFF**: Debugger can configure and use watchpoint <*x*>. • **ON**: Debugger shall not configure and overwrite watchpoint <*x*>.

Example:

```
TrOnchip.RESERVE BP0 ON    ; Exclude breakpoint 0 from debugger usage
TrOnchip.RESERVE WP7 ON    ; Exclude watchpoint 7 from debugger usage
```

NOTE:	If all available breakpoints or watchpoints are excluded from debugger usage, the debugger cannot set onchip instruction or read/write breakpoints anymore. At least one breakpoint and one, preferably two watchpoints should be available to the debugger at any time.

TrOnchip.RESet Set on-chip trigger to default state

Format:	**TrOnchip.RESet**

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

©1989-2020 Lauterbach GmbH

Format:	**TrOnchip.Set** [**RESET** | **OSUnloCk** | **NSEL1** | **NSEL1E** | **NSEL1R** | **NSEL2** | **NSEL2E** | **NSEL2R** | **SEL1** | **SEL1E** | **SEL1R** | **SEL3** | **SEL3E** | **SEL3R**] [**ON** | **OFF**]

Default: RESET ON, all other options are OFF.

These options are available for all ARMv8 cores. Several exception catch events can be set simultaneously.

RESET [**ON** | **OFF**]	The core stops on a reset event.
OSUnloCk [**ON** | **OFF**]	The core stops on a OS unlock event.
TDA [**ON** | **OFF**]	The core is halted when it accesses debug registers and when the OS-lock is cleared.
NSEL1 [**ON** | **OFF**]	Non-secure exception level 1. The core stops when the exception level changes to non-secure EL1
NSEL2 [**ON** | **OFF**]	Non-secure exception level 2. The core stops when the exception level changes to non-secure EL2.
SEL1 [**ON** | **OFF**]	Secure exception level 1. The core stops when the exception level changes to secure EL1.
SEL3 [**ON** | **OFF**]	Secure exception level 3. The core stops when the exception level changes to secure EL3.
StepVector (deprecated)	Please see **TrOnchip.StepVector**.

Additional options for ARMv8.2-A cores (or later):

NSEL1E [**ON** | **OFF**]	Non-secure exception level 1 entry. The core stops on an exception entry to non-secure EL1, e.g. from a lower exception level.
NSEL1R [**ON** | **OFF**]	Non-secure exception level 1 return. The core stops on an exception return to non-secure EL1, e.g. from a higher exception level.
NSEL2E [**ON** | **OFF**]	Non-secure exception level 2 entry. The core stops on an exception entry to non-secure EL2, e.g. from a lower exception level.

©1989-2020 Lauterbach GmbH

NSEL2R [**ON** | **OFF**]	Non-secure exception level 2 return. The core stops on an exception return to non-secure EL2, e.g. from a higher exception level.
SEL1E [**ON** | **OFF**]	Secure exception level 1 entry. The core stops on an exception entry to secure EL1, e.g. from a lower exception level.
SEL1R [**ON** | **OFF**]	Secure exception level 1 return. The core stops on an exception return to secure EL1, e.g. from a higher exception level.
SEL3E [**ON** | **OFF**]	Secure exception level 3 entry. The core stops on an exception entry to secure EL3, e.g. from a lower exception level.
SEL3R [**ON** | **OFF**]	Secure exception level 3 return. The core stops on an exception return to secure EL3, e.g. from a higher exception level.

Additional options for ARMv8.4-A cores (or later):

SEL2 [**ON** | **OFF**]	Secure exception level 2. The core stops when the exception level changes to secure EL2.
SEL2E [**ON** | **OFF**]	Secure exception level 2 entry. The core stops on an exception entry to secure EL2, e.g. from a lower exception level.
SEL2R [**ON** | **OFF**]	Secure exception level 2 return. The core stops on an exception return to secure EL2, e.g. from a higher exception level.

Exception Catch into Current Exception Level of CPU (ARMv8.0 and ARMv8.1 only)

If the CPU shall resume operation upon a **Go.direct** or **Step.single**, this means the CPU has to leave debug mode. Leaving debug mode has the effect of an exception return from debug mode to the current exception level of the CPU. Such an exception return might already trigger an exception catch if a catch event was set for the current CPU mode. This only happens for ARMv8.0 or ARMv8.1 based devices. In this case, the debugger will issue a warning that the CPU cannot resume execution.

Example:

The CPU is currently in AArch64 EL2 and should be caught when it changes from non-secure EL1 to EL2.

The CPU has stopped in debug mode. **TrOnchip.Set NSEL2** has been set to **ON**. This means that an entry to EL2 will trigger a CPU stop. Now if a go is executed, the debugger will restore the CPU state and return the CPU from debug mode to EL2. This entry to EL2, however, will trigger the set EL2 catch event. The CPU will now enter debug mode again without having executed a single line of code.

©1989-2020 Lauterbach GmbH

Workaround:

If an exception trigger is used, it is likely to assume that the CPU will not stay in its exception level. If it would, such a trigger would be useless. Therefore try to proceed as follows:

- Disable the catch event for the current exception level of the CPU. If the CPU should resume from EL2 like in the example, disable NSEL2.
- Instead set a trigger to the EL the CPU will switch to and wait for it to get stopped there. In the example this would mean to set NSEL1 to **ON**.

Disable the last set trigger and set your initial catch event. In the example this would mean to set NSEL2 to **ON** again.

TrOnchip.StepVector — Step into exception handler

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

Step into exception handler if **ON**. Step over exception handler if **OFF**. In EDECR, this changes the bit EDECR.SS (Halting step enable).

TrOnchip.StepVectorResume — Catch exceptions and resume single step

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

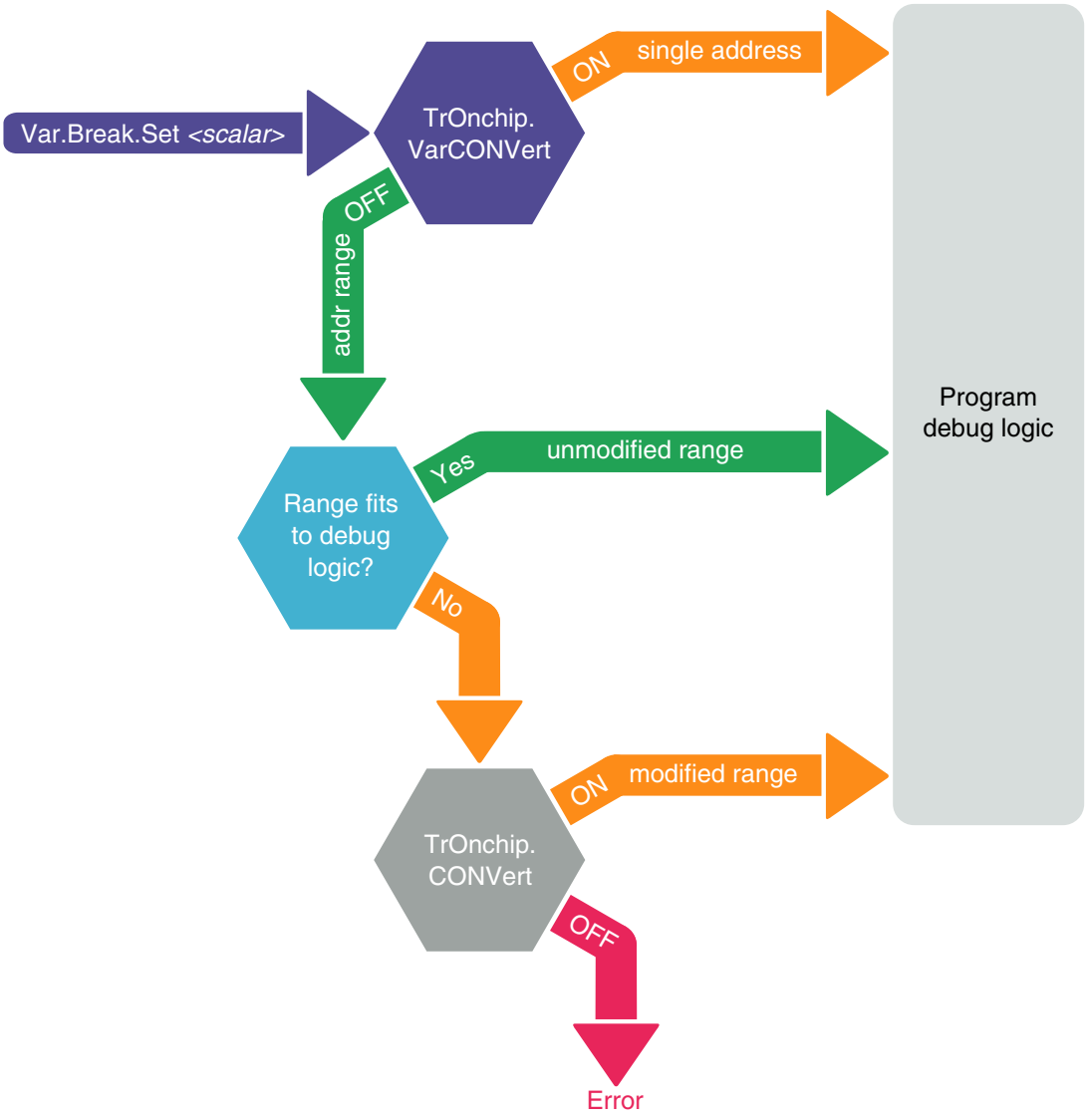
Default: OFF.

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

©1989-2020 Lauterbach GmbH

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.



©1989-2020 Lauterbach GmbH

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

[Go to figure]

Format:	**TrOnchip.state**

Opens the **TrOnchip.state** window.

©1989-2020 Lauterbach GmbH

Cache Analysis and Maintenance

ARMv8 cores feature a hierarchical memory system with multiple levels of cache. Using the **CACHE** command group, you can analyze and alter all cache levels that are accessible by external debuggers. The TRACE32 cache support visualizes the essential information about stored cache entries, including full decoding of the cache tag information. To perform basic cache maintenance tasks, please have a look at these recommended commands:

CACHE.view	Display control registers for configuration of all cache levels.
CACHE.INVALIDATE	Invalidate all entries stored in L1 or L2 instruction/data cache.
CACHE.FLUSH	Clean and invalidate all entries stored in L1 or L2 instruction/data cache.

The cache topology and capabilities of an ARMv8 cores are defined by its implementation and therefore may vary significantly for different types of cores. Especially commands for basic cache analysis tasks may be affected by this:

CACHE.DUMP	Display all entries located in L1 or L2 instruction/data cache.

For an overview of the TRACE32 cache support for various ARMv8 cores, please refer to **"TRACE32 Cache Support by CPU Type"**, page 202.

The list of referenced members of the **CACHE** command group in this section is far from complete. For a full list, please see **"General Commands Reference Guide C"** (general_ref_c.pdf).

©1989-2020 Lauterbach GmbH

TRACE32 Cache Support by CPU Type

ARMv8 cores with hierarchical memory system feature varying cache analysis capabilities. Each implementation can restrict the access the access of external debuggers to certain cache levels or sections. A short comparison for different types is shown here:

ARMv8 Core	L1 IC	L1 DC	L2	L3
Cortex-A32	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Cortex-A35	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Cortex-A53	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Cortex-A55	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Cortex-A57	Cache analysis and maintenance	Cache analysis and maintenance	Cache analysis and maintenance	-
Cortex-A65	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Cortex-A72	Cache analysis and maintenance	Cache analysis and maintenance	Cache analysis and maintenance	-
Cortex-A73	Cache analysis only when EL3 is in AArch64 state. No restrictions on cache maintenance.	Cache analysis only when EL3 is in AArch64 state. No restrictions on cache maintenance.	Cache maintenance	-
Cortex-A75	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Cortex-A76 **Cortex-A76AE**	Cache analysis and maintenance	Cache analysis and maintenance	Cache analysis and maintenance	-
Cortex-A77	Cache analysis and maintenance	Cache analysis and maintenance	Cache analysis and maintenance	-
Cortex-R52	Cache analysis and maintenance	Cache analysis and maintenance	-	-
Neoverse E1	Cache analysis and maintenance	Cache analysis and maintenance	Cache maintenance	-
Neoverse N1	Cache analysis and maintenance	Cache analysis and maintenance	Cache analysis and maintenance	-

©1989-2020 Lauterbach GmbH

NOTE:	Cache analysis refers to the readout of the cache content. See **CACHE.FLUSH** Cache maintenance refers to cache clean and invalidate operations. See**CACHE.FLUSH** and **CACHE.INVALIDATE**

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

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.

CPU specific Tables in MMU.DUMP <table>

ITLB	Displays the contents of the Instruction Translation Lookaside Buffer.
DTLB	Displays the contents of the Data Translation Lookaside Buffer.
TLB0	Displays the contents of the Translation Lookaside Buffer 0.
TLB1	Displays the contents of the Translation Lookaside Buffer 1.
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.

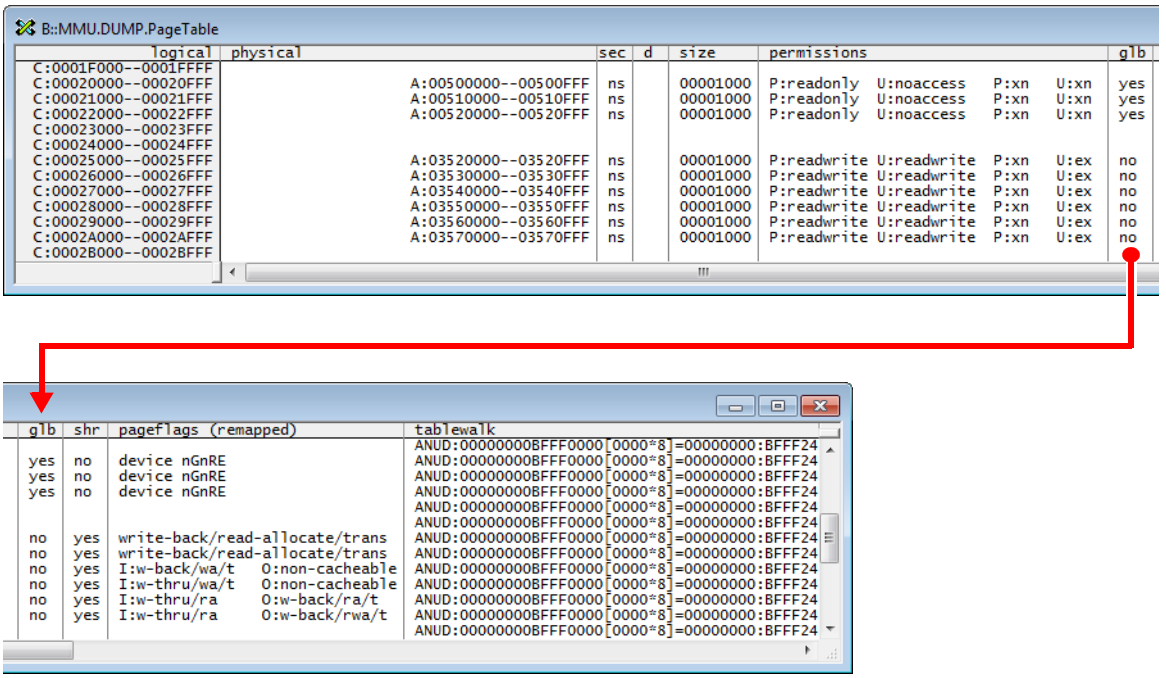
©1989-2020 Lauterbach GmbH

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. (i.e., 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.
EL1PageTable	Displays the translation table used if the CPU is in Exception level EL0 or EL1. This is the table pointed to by MMU registers TTBR0_EL1 and TTBR1_EL1. The option is enabled if Exception levels EL0 or EL1 use AArch64 mode.
EL2PageTable	Displays the translation table used if the CPU is in Exception level EL2. This is the table pointed to by MMU register TTBR_EL2. The option is available only if Exception level EL2 is implemented in the CPU.
EL3PageTable	Displays the translation table used if the CPU is in Exception level EL3. This is the table pointed to by MMU register TTBR_EL3. The option is available only if Exception level EL3 is implemented in the CPU and EL3 uses AArch64 mode.

Next:

- **Screenshot of the MMU.DUMP.PageTable Window**
- **Description of the Columns**
- **Examples**

©1989-2020 Lauterbach GmbH



Column Name	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	Data access permissions and instruction execution permissions. For details, see description of the access permissions: • **Data Access Permissions for non-Stage2 Page Tables** • **Data Access Permissions for Stage2 Page Tables** • **Instruction Execution Permissions for non-Stage2 Page Tables** • **Instruction Execution Permissions for Stage2 Page Tables**
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	Details of table walk for logical page address (one sub column for each table level, showing the access class, table base address, entry index, entry width in bytes and value of table entry)

©1989-2020 Lauterbach GmbH

Data Access Permissions for non-Stage2 Page Tables (AArch64 EL0/1/2/3 and AArch32 PL0/1/2)

[Back to Column Name: permissions]

This table describes the data access permissions shown in the **permissions** column of the following windows:

- **MMU.List.PageTable** window (AArch64 EL0/1/2/3 or AArch32 PL0/1/2)
- **MMU.DUMP.PageTable** window (AArch64 EL0/1/2/3 or AArch32 PL0/1/2)
- **SMMU.StreamMapRegGrp.Dump** window
- **SMMU.StreamMapRegGrp.list** window

P: *<data_access_permission>* data access permissions for privileged code (PL1 / EL1)
U: *<data_access_permission>* data access permissions for unprivileged code (PL0 / EL0)

Column "permissions" *<data_access_permission>* Keywords	**Description of the <data_access_permission> Keywords**
noaccess	No read access and no write access for this privilege level
readonly	Read access, no write access for this privilege level
readwrite	Read and write access for this privilege level
noaccess (DACR) (AArch32 only)	No read access and no write access for this privilege level as specified by register DACR bits associated with the domain.
readwrite (DACR) (AArch32 only)	Read and write access for this privilege level *(Managers)* as specified by register DACR bits associated with the domain.
reserved (DACR) (AArch32 only)	Reserved setting (unpredictable behavior) as specified by register DACR bits associated with the domain.
reserved	Reserved setting (unpredictable behavior)

©1989-2020 Lauterbach GmbH

Data Access Permissions for Stage2 Page Tables

[Back to Column Name: permissions]

This table describes the data access permissions shown in the **permissions** column of the following windows:

- **MMU.List.IntermediatePageTable** window
- **MMU.DUMP.IntermediatePageTable** window
- **SMMU.StreamMapRegGrp.Dump /IPT** window
- **SMMU.StreamMapRegGrp.list /IPT** window

Column "permissions" *<data_access_permission>* Keywords	**Description of the <data_access_permission> Keywords**
no access (AArch32 only)	No read access and no write access for non-secure PL1 and PL0
read access only (AArch32 only)	Read access and no write access for non-secure PL1 and PL0
write access only (AArch32 only)	Write access and no read access for non-secure PL1 and PL0
read/write access (AArch32 only)	Read and write access for non-secure PL1 and PL0
EL0,1: no access (AArch64 only)	No read access and no write access for non-secure EL1 and EL0
EL0,1: read only (AArch64 only)	Read access and no write access for non-secure EL1 and EL0
EL0,1: write only (AArch64 only)	Write access and no read access for non-secure EL1 and EL0
EL0,1: read/write (AArch64 only)	Read and write access for non-secure EL1 and EL0

©1989-2020 Lauterbach GmbH

Instruction Execution Permissions for non-Stage2 Page Tables (AArch64 EL0/1/2/3 and AArch32 PL0/1/2)

[Back to Column Name: permissions]

This table describes the access permissions for instruction execution shown in the **permissions** column of the following windows:

- **MMU.List.PageTable** window (AArch64 EL0/1/2/3 or AArch32 PL0/1/2)
- **MMU.DUMP.PageTable** window (AArch64 EL0/1/2/3 or AArch32 PL0/1/2)
- **SMMU.StreamMapRegGrp.Dump** window
- **SMMU.StreamMapRegGrp.list** window

P: *<execution_permission>* instruction execution permission for privileged code (PL1 / EL1)
U: *<execution_permission>* instruction execution permission for unprivileged code (PL0 / EL0)

Column "permissions" *<execution_permission>* Keywords	**Description of the <execution_permission> Keywords**
ex or **exec**	Code execution allowed
xn or **notexec**	Code execution not allowed
xnW	Code execution not allowed due to setting of bit WXN in CPU register SCTLR (AArch32) or SCTLR_ELx (AArch64) OR because the ARM AArch64 execution treats all regions writable at EL0 as being PXN
xnD	Code execution not allowed because dirty bit modifier (DBM) is set in page table entry
nopexec	Code execution not allowed for privileged level, code execution allowed for unprivileged level

©1989-2020 Lauterbach GmbH

Description of the Instruction Execution Permissions for Stage2 Page Tables

[Back to Column Name: permissions]

This table describes the access permissions for instruction execution shown in the **permissions** column of the following windows:

- **MMU.List.IntermediatePageTable** window
- **MMU.DUMP.IntermediatePageTable** window
- **SMMU.StreamMapRegGrp.Dump /IPT** window
- **SMMU.StreamMapRegGrp.list /IPT** window

Column "permissions" *<execution_permission>* Keywords	**Description of the <execution_permission> Keywords**
EL1:xn EL0:xn	No code execution allowed for EL1 and EL0
EL1:xn EL0:ex	No code execution allowed for EL1, code execution allowed for EL0
EL1:ex EL0:xn	Code execution allowed for EL1, no code execution allowed for EL0
EL1:ex EL0:ex	Code execution allowed for EL1 and EL0

©1989-2020 Lauterbach GmbH

Description of the Memory Attributes

[Back to Column Name: pageflags]

This table describes the memory attributes displayed in the **pageflags** column of the following windows:

- **MMU.List.PageTable** (any page table) and **MMU.List.IntermediatePageTable** window
- **MMU.DUMP.PageTable** (any page table) and **MMU.DUMP.IntermediatePageTable** window
- **SMMU.StreamMapRegGrp.Dump** window
- **SMMU.StreamMapRegGrp.list** window

Column “pageflags”	Description of the memory attributes
device nGnRnE	Device memory type non-Gathering, non-Reordering, no Early write acknowledgement
device nGnRE	Device memory type non-Gathering, non-Reordering, Early Write Acknowledgement
device nGRE	Device memory type non-Gathering, Reordering, Early Write Acknowledgement
device GRE	Device memory type Gathering, Reordering, Early Write Acknowledgement
write-thru/write-allo-cate/trans	Inner and Outer Normal Memory, Write-through transient, write allocate policy
write-thru/read-allo-cate/trans	Inner and Outer Normal Memory, Write-through transient, read allocate policy
write-thru/r-w-allo-cate/trans	Inner and Outer Normal Memory, Write-through transient, read and write allocate policy
non-cacheable	Inner and Outer Normal Memory, Non-Cacheable
write-back/write-allo-cate/trans	Inner and Outer Normal Memory, Write-back transient, write allocate policy
write-back/read-allo-cate/trans	Inner and Outer Normal Memory, Write-back transient, read allocate policy
write-back/r-w-allo-cate/trans	Inner and Outer Normal Memory, Write-back transient, read and write allocate policy
write-thru/write-allocate	Inner and Outer Normal Memory, Write-through non-transient, write allocate policy
write-thru/read-allocate	Inner and Outer Normal Memory, Write-through non-transient, read allocate policy
write-thru/read-write-allo-cate	Inner and Outer Normal Memory, Write-through non-transient, read and write allocate policy
write-back/write-allocate	Inner and Outer Normal Memory, Write-back non-transient, write allocate policy

©1989-2020 Lauterbach GmbH

Column “pageflags”	Description of the memory attributes
write-back/read-allocate	Inner and Outer Normal Memory, Write-back non-transient, read allocate policy
write-back/read-write-allocate	Inner and Outer Normal Memory, Write-back non-transient, read and write allocate policy
I:/O: w-thru/wa/t	Inner / Outer Normal Memory, Write-through transient, write allocate policy
I:/O: w-thru/ra/t	Inner / Outer Normal Memory, Write-through transient, read allocate policy
I:/O: w-thru/rwa/t	Inner / Outer Normal Memory, Write-through transient, read and write allocate policy
I:/O: non-cacheable	Inner / Outer Normal Memory, Non-Cacheable
I:/O: w-back/wa/t	Inner / Outer Normal Memory, Write-back transient, write allocate policy
I:/O: w-back/ra/t	Inner / Outer Normal Memory, Write-back transient, read allocate policy
I:/O: w-back/rwa/t	Inner / Outer Normal Memory, Write-back transient, read and write allocate policy
I:/O: w-thru/wa	Inner / Outer Normal Memory, Write-through non-transient, write allocate policy
I:/O: w-thru/ra	Inner / Outer Normal Memory, Write-through non-transient, read allocate policy
I:/O: w-thru/rwa	Inner / Outer Normal Memory, Write-through non-transient, read and write allocate policy
I:/O: w-back/wa	Inner / Outer Normal Memory, Write-back non-transient, write allocate policy
I:/O: w-back/ra	Inner / Outer Normal Memory, Write-back non-transient, read allocate policy
I:/O: w-back/rwa	Inner / Outer Normal Memory, Write-back non-transient, read and write allocate policy
P:*<permission>* **U:***<permission>*	Data access permissions for privileged and unprivileged execution level. *<permission>* may be readwrite, readonly or noaccess
exec	Code execution allowed for privileged and unprivileged level
notexec	Code execution allowed for privileged level but not for unprivileged level
nopexec	Code execution not allowed for privileged and unprivileged level

©1989-2020 Lauterbach GmbH

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:

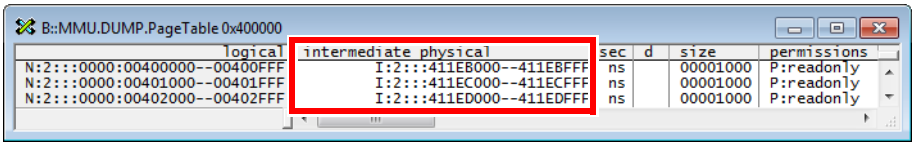
```
SYStem.Option MACHINESPACES ON

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

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

Example 3:

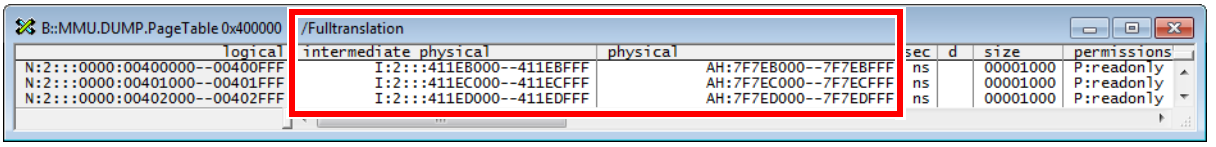
```
SYStem.Option MACHINESPACES ON

;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
```

©1989-2020 Lauterbach GmbH

Results for 3 a) and 3 b)



logical	intermediate physical	sec	d	size	permissions
N:2:::0000:00400000--00400FFF	I:2:::411EB000--411EBFFF	ns		00001000	P:readonly
N:2:::0000:00401000--00401FFF	I:2:::411EC000--411ECFFF	ns		00001000	P:readonly
N:2:::0000:00402000--00402FFF	I:2:::411ED000--411EDFFF	ns		00001000	P:readonly



logical	intermediate physical	physical	sec	d	size	permissions
N:2:::0000:00400000--00400FFF	I:2:::411EB000--411EBFFF	AH:7F7EB000--7F7EBFFF	ns		00001000	P:readonly
N:2:::0000:00401000--00401FFF	I:2:::411EC000--411ECFFF	AH:7F7EC000--7F7ECFFF	ns		00001000	P:readonly
N:2:::0000:00402000--00402FFF	I:2:::411ED000--411EDFFF	AH:7F7ED000--7F7EDFFF	ns		00001000	P:readonly

MMU.List — Compact display of MMU translation table

Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] [/*<option>*] **MMU.***<table>***.List** (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**

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

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

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.

©1989-2020 Lauterbach GmbH

<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>

TLB	Displays the contents of the Translation Lookaside Buffer.
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.

©1989-2020 Lauterbach GmbH

IntermedPageTable	Displays the translation table used by the MMU for the second stage translation of a guest machine. (i.e., 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.
EL1PageTable	Displays the translation table used if the CPU is in Exception level EL0 or EL1. This is the table pointed to by MMU registers TTBR0_EL1 and TTBR1_EL1. The option is enabled if Exception levels EL0 or EL1 use AArch64 mode.
EL2PageTable	Displays the translation table used if the CPU is in Exception level EL2. This is the table pointed to by MMU register TTBR_EL2. The option is available only if Exception level EL2 is implemented in the CPU.
EL3PageTable	Displays the translation table used if the CPU is in Exception level EL3. This is the table pointed to by MMU register TTBR_EL3. The option is available only if Exception level EL3 is implemented in the CPU and EL3 uses AArch64 mode.

MMU.SCAN Load MMU table from CPU

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 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.

©1989-2020 Lauterbach GmbH

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.

©1989-2020 Lauterbach GmbH

TRACE32 TLB Support by CPU Type

ARMv8 cores with virtual memory system may provide read access to internal TLB structures. The organization of the hierarchical TLB structures may vary strongly between core implementations and each may grant external debuggers different levels of access. A short comparison for different types is shown here:

ARMv8 Core	ITLB	DTLB	TLB0	TLB1
Cortex-A32	Reads main TLB		-	-
Cortex-A35	Reads main TLB		-	-
Cortex-A53	Reads main TLB		-	-
Cortex-A55	-	-	Reads L2 TLB	-
Cortex-A57	Reads L1 instruction TLB	Reads L1 data TLB	Reads L2 TLB	-
Cortex-A65	-	-	Reads L2 TLB	-
Cortex-A72	Reads L1 instruction TLB	Reads L1 data TLB	Reads L2 TLB	-
Cortex-A73	-	-	Reads L2 TLB RAM0	Reads L2 TLB Ram1
Cortex-A75	-	-	Reads L2 TLB RAM0	Reads L2 TLB Ram1
Cortex-A76 Cortex-A76AE	Reads L1 instruction TLB	Reads L1 data TLB	Reads L2 TLB	-
Cortex-A77	Reads L1 instruction TLB	Reads L1 data TLB	Reads L2 TLB	-
Neoverse E1	-	-	Reads main TLB	-
Neoverse N1	Reads L1 instruction TLB	Reads L1 data TLB	Reads L2 TLB	-

©1989-2020 Lauterbach GmbH

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 ARMv8   ;for example, the 'ARMv8' CPU is SMMU-capable

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

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

- The context type defined for each stream map register group (SMRG). This visualization shows the translation context associated with the SMRG such as:
  - The stage 1 context bank and the stage 1 page table type
  - The stage 2 context bank and the stage 2 page table type
  - The information whether the SMRG context is a HYPC and MONC context type
- The stream matching register settings, if supported by the SMMU
- The associated context bank index, the page table format and the MMU-enable/disable state for stage 1 and/or stage 2 address translation contexts
- Page table 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.
- A quick indication of the global SMMU fault status
- Peripheral register view:
  - Global Configuration Registers of the SMMU
  - Stream Matching and Mapping Registers
  - Context Bank Registers

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

- The **SMMU.ADD** command
- The **SMMU.StreamMapTable** command
- **Glossary - SMMU**
- **Arguments in SMMU Commands**

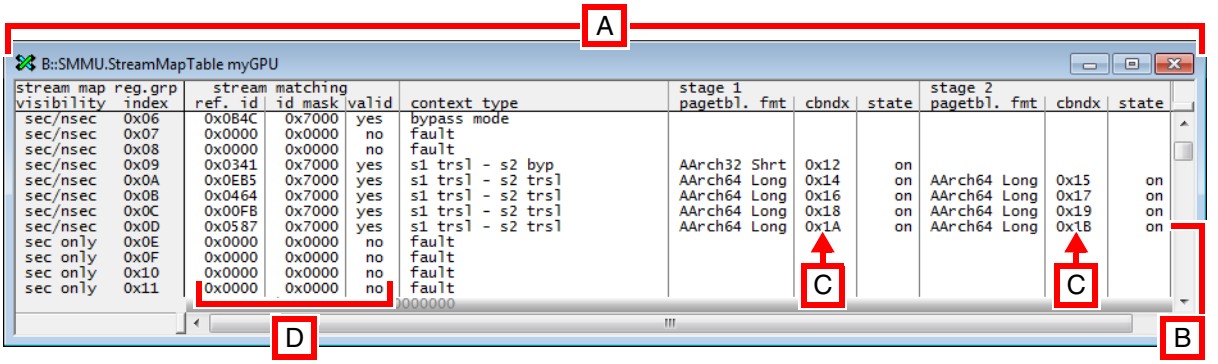
©1989-2020 Lauterbach GmbH

The **SMMU.StreamMapTable** 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.StreamMapTable** 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.

Glossary - SMMU

This figure illustrates a few SMMU terms. For explanations of the illustrated SMMU terms and other important SMMU terms not shown here, see below.



stream map visibility	reg.grp index	stream matching ref. id	id mask	valid	context type	stage 1 pagetbl. fmt	cbndx	state	stage 2 pagetbl. fmt	cbndx	state
sec/nsec	0x06	0x0B4C	0x7000	yes	bypass mode						
sec/nsec	0x07	0x0000	0x0000	no	fault						
sec/nsec	0x08	0x0000	0x0000	no	fault						
sec/nsec	0x09	0x0341	0x7000	yes	s1 trsl - s2 byp	AArch32 Shrt	0x12	on			
sec/nsec	0x0A	0x0EB5	0x7000	yes	s1 trsl - s2 trsl	AArch64 Long	0x14	on	AArch64 Long	0x15	on
sec/nsec	0x0B	0x0464	0x7000	yes	s1 trsl - s2 trsl	AArch64 Long	0x16	on	AArch64 Long	0x17	on
sec/nsec	0x0C	0x00FB	0x7000	yes	s1 trsl - s2 trsl	AArch64 Long	0x18	on	AArch64 Long	0x19	on
sec/nsec	0x0D	0x0587	0x7000	yes	s1 trsl - s2 trsl	AArch64 Long	0x1A	on	AArch64 Long	0x1B	on
sec only	0x0E	0x0000	0x0000	no	fault						
sec only	0x0F	0x0000	0x0000	no	fault						
sec only	0x10	0x0000	0x0000	no	fault						
sec only	0x11	0x0000	0x0000	no	fault						

A See stream mapping 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.

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.

Security State Determination Table (SSD Table)

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.

©1989-2020 Lauterbach GmbH

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.

Stream Map Register Group (SMRG)

A group of SMMU registers determining the translation context for a memory transaction stream, see **stream mapping table**.

Stream Mapping Table (short: Stream Map Table)

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.

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. The stream mapping table is the central table of the SMMU. See **SMMU.StreamMapTable**.

Stream Matching

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 mapping 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.StreamMapTable** window.

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 mapping table determines which translation context is applied to an incoming memory transaction stream.

Translation Context Bank (short: Context Bank)

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.

©1989-2020 Lauterbach GmbH

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.StreamMapTable**.
<cbndx>	Index of a translation context bank.
<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.

©1989-2020 Lauterbach GmbH

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

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:

<base_address>	Logical or physical base address of the memory-mapped SMMU register space. **NOTE**: 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 be able to see **all** SMMU information. • If you specify a **non-secure** address as the SMMU base address, you will only see the SMMU information which is visible from the non-secure domain. To specify a **secure** address, precede the base address with an access class such as AZ: or ZD: The **SMMU.ADD** command interprets access classes with an ambiguous security status as secure access classes: • Physical access class A: becomes AZ: • Logical access classes like D: or C: become ZSD: The **SMMU.ADD** command leaves access classes with a distinct security status unchanged, e.g. the access classes NSD:, NUD:, HD: etc.

©1989-2020 Lauterbach GmbH

<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: **MMU400**, **MMU401**, or **MMU500**.

Example:

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU500 AZ:0x50000000

;display the stream map table named myGPU
SMMU.StreamMapTable myGPU
```

SMMU.Clear — Delete an SMMU

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
```

©1989-2020 Lauterbach GmbH

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.ContextBank	Display the registers of a context bank
SMMU.Register.Global	Display the global registers of an SMMU
SMMU.Register.StreamMapRegGrp	Display the registers of an SMRG

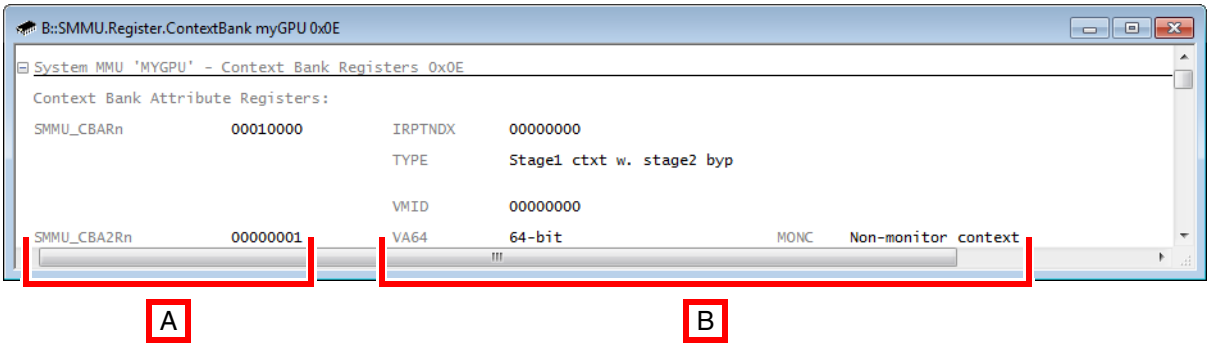
Example:

```
;open the SMMU.Register.StreamMapRegGrp window
SMMU.Register.StreamMapRegGrp   myGPU   0x0A

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

©1989-2020 Lauterbach GmbH

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.

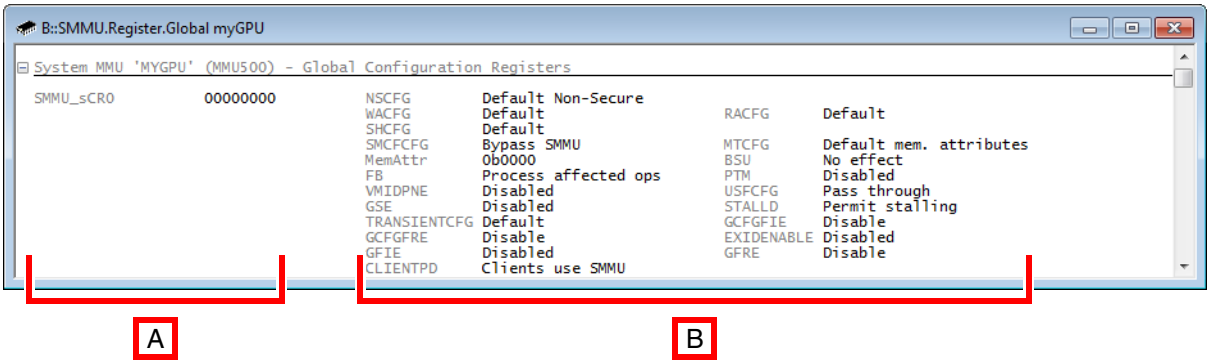
Example:

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

©1989-2020 Lauterbach GmbH

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
```

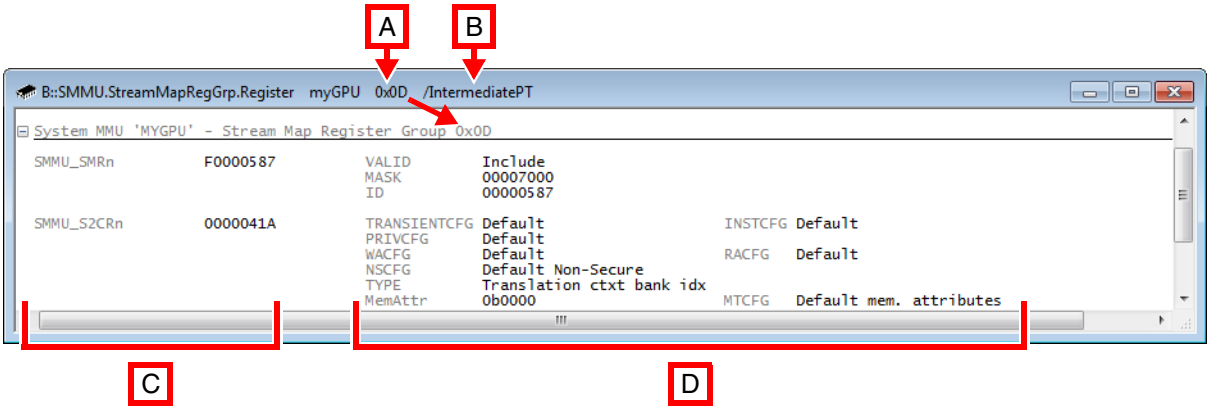
To display the global registers of an SMMU via the user interface TRACE32 PowerView:

- In the **SMMU.StreamMapTable** window, right-click an SMRG, and then select **Peripherals** > **Global Configuration Registers** from the popup menu.

©1989-2020 Lauterbach GmbH

Format:	**SMMU.Register.StreamMapRegGrp** *<args>* **SMMU.StreamMapRegGrp.Register** *<args>* (as an alias)
<args>:	*<name>* *<smrg_index>* [**/IntermediatePT**]

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 The option **IntermediatePT** is used to display the context bank registers of stage 2.

C Register name and content.

D 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.

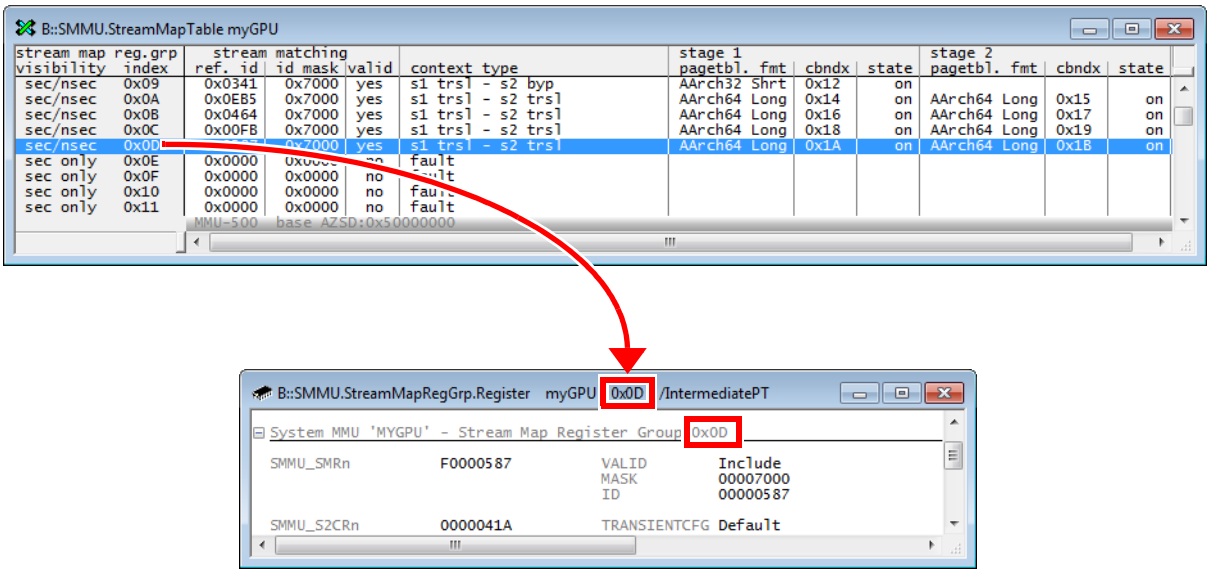
Example:

```
SMMU.StreamMapRegGrp.Register    myGPU    0x0D    /IntermediatePT
```

©1989-2020 Lauterbach GmbH

To view the registers of an SMRG via the user interface TRACE32 PowerView:

- In the **SMMU.StreamMapTable** 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**

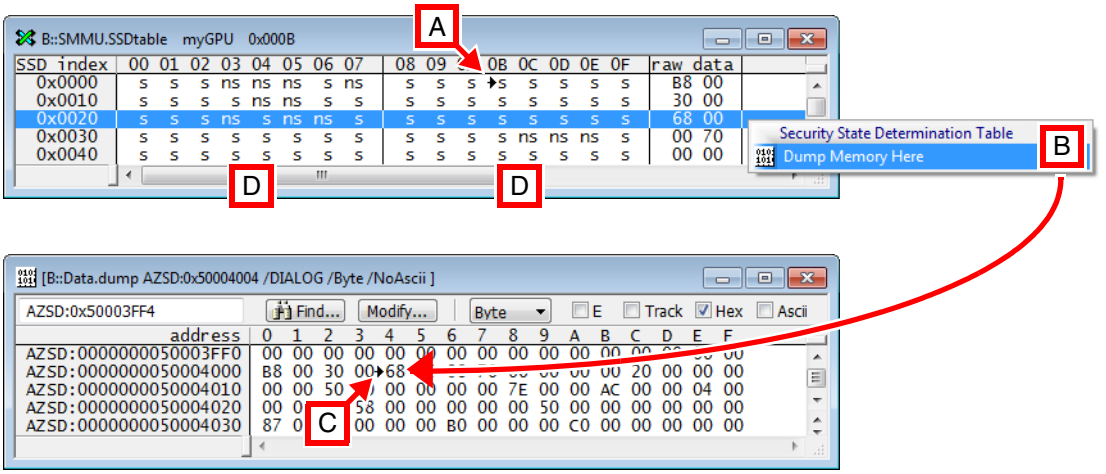
Deletes all SMMU definitions created with **SMMU.ADD** from TRACE32. The **SMMU.RESet** command does not affect your target SMMU.

To delete an individual SMMU created with **SMMU.ADD**, use **SMMU.Clear**.

©1989-2020 Lauterbach GmbH

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.

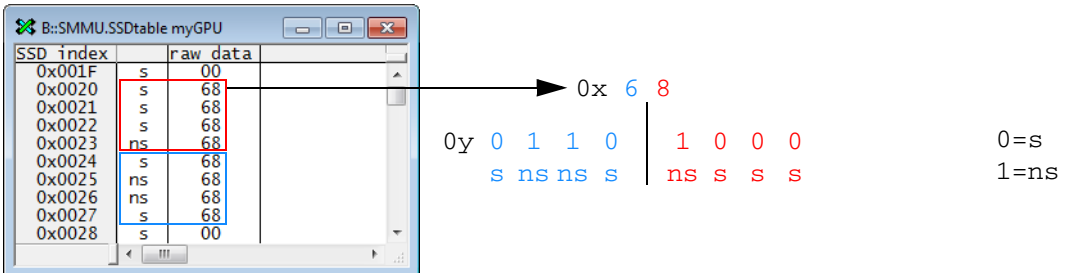


A In the SSD table, the black arrow indicates the *<start_index>*, here 0x00B

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.

©1989-2020 Lauterbach GmbH

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.StreamMapTable** 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).

©1989-2020 Lauterbach GmbH

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:

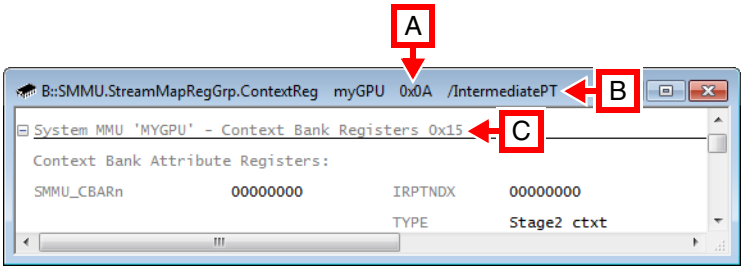
SMMU.StreamMapRegGrp.ContextReg	Shows the registers of the context bank associated with the stage 1 and/or stage 2 translation.
SMMU.StreamMapRegGrp.Dump	Dumps the page table associated with the stage 1 and/or stage 2 translation page wise.
SMMU.StreamMapRegGrp.list	Lists the page table entries associated with the stage 1 and/or stage 2 translation in a compact format.

©1989-2020 Lauterbach GmbH

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.StreamMapTable** 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>*.

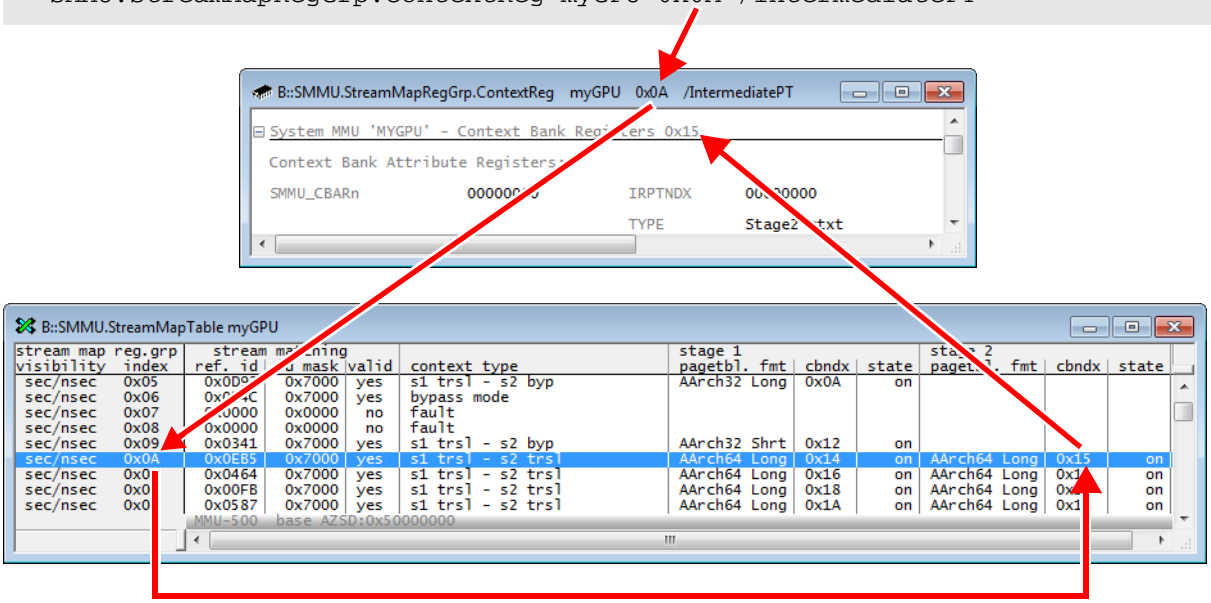
Arguments:

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

©1989-2020 Lauterbach GmbH

PRACTICE Script Example and Illustration of the Context Bank Look-up:

```
SMMU.StreamMapRegGrp.ContextReg myGPU 0x0A /IntermediatePT
```

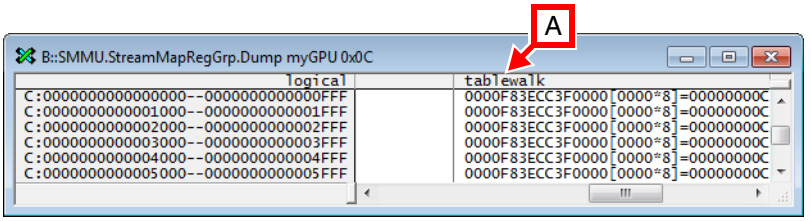


To display the context bank registers via the user interface TRACE32 PowerView:

- In the **SMMU.StreamMapTable** window, right-click an SMRG, and then select **Peripherals** > **Context Bank Registers of Stage 1** or **2** from the popup menu.

Format:	**SMMU.StreamMapRegGrp.Dump** *<args>*
<args>:	*<name> <smrg_index>* [*<address>* | *<range>*] [*/<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.
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:

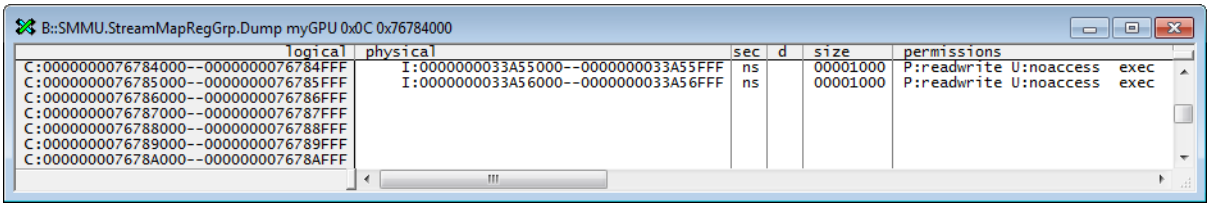
- In the **SMMU.StreamMapTable** window, right-click an SMRG, and then select from the popup menu:
  - **Stage 1 Page Table** > **Dump** or
  - **Stage 2 Page Table** > **Dump**

©1989-2020 Lauterbach GmbH

Description of Columns

This table describes the columns of the following windows:

- **SMMU.StreamMapRegGrp.list**
- **SMMU.StreamMapRegGrp.Dump**

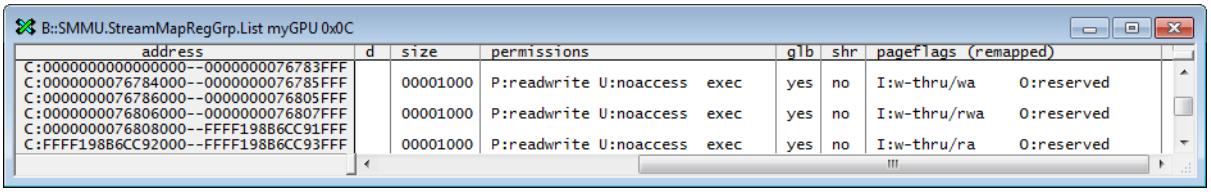


Column	Description
logical	Logical page address range
physical	Physical page address range
sec	Security state of entry (s=secure, ns=non-secure, sns=non-secure entry in secure page table)
d	Domain
size	Size of mapped page in bytes
permissions	Access permissions (P=privileged, U=unprivileged, exec=execution 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)

©1989-2020 Lauterbach GmbH

Format:	**SMMU.StreamMapRegGrp.list** *<args>*
<args>:	*<name> <smrg_index>* [*<address>* | *<range>*] [**/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.
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
```

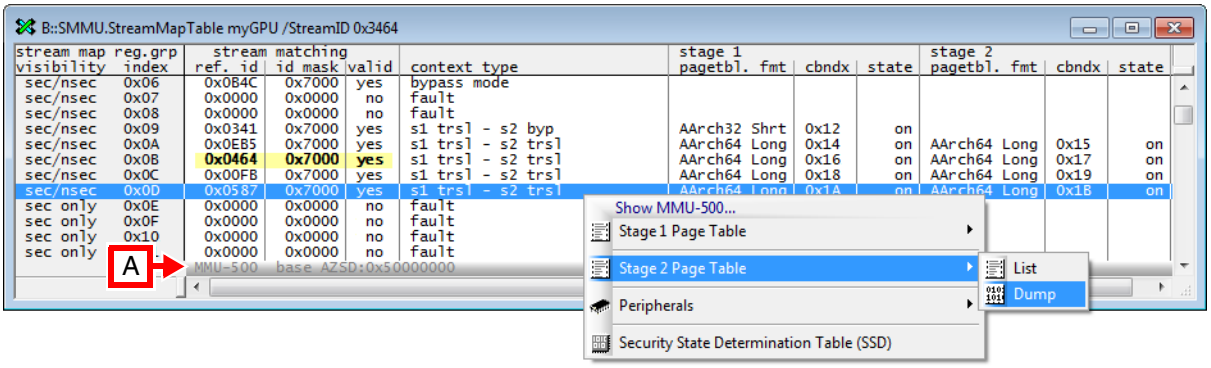
To list the page table entries via the user interface TRACE32 PowerView:

- In the **SMMU.StreamMapTable** window, right-click an SMRG, and then select from the popup menu:
  - **Stage 1 Page Table** > **List** or
  - **Stage 2 Page Table** > **List**

©1989-2020 Lauterbach GmbH

Format:	**SMMU.StreamMapTable** *<name>* [**/StreamID** *<value>*]

Opens the **SMMU.StreamMapTable** window, listing all stream map register groups of the SMMU that has the specified *<name>*. The window provides an overview of the 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.

Arguments

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

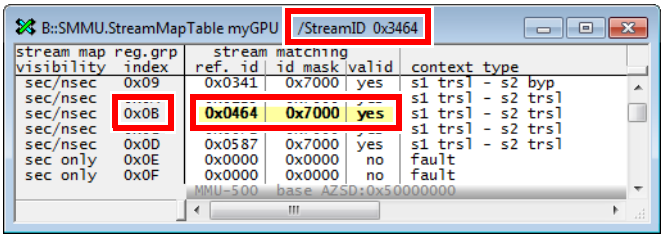
©1989-2020 Lauterbach GmbH

Example

[Back to Top]

This PRACTICE script example shows how to define an SMMU with the **SMMU.ADD** command. Then the script opens the SMMU in the **SMMU.StreamMapTable** window, searches for the *<stream_id>* 0x3463 and highlights the matching SMRG 0x0464 in yellow.

```
;define a new SMMU named "myGPU" for a graphics processing unit
SMMU.ADD "myGPU" MMU500 A:0x50000000

;open the window and highlight the matching SMRG in yellow
SMMU.StreamMapTable myGPU /StreamID 0x3464
```



NOTE:	At first glance, the **StreamID** 0x3464 does not seem to match the SMRG 0x0464. 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.StreamMapTable** window is a correct match for the **StreamID** 0x3464 we searched for.

See also function **SMMU.StreamID2SMRG()** in **"General Function Reference"** (general_func.pdf).

About the SMMU.StreamMapTable Window

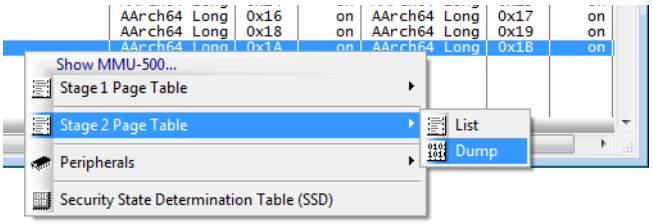
[Back to Top]

By right-clicking an SMRG or double-clicking certain cells of an SMRG, you can open additional windows to receive more information about the selected SMRG.

- Right-clicking opens the **Popup Menu**.
- Double-clicking an SMRG 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.

©1989-2020 Lauterbach GmbH

[Back to Top]



The popup menu in the **SMMU.StreamMapTable** window provides convenient shortcuts to the following commands:

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**

©1989-2020 Lauterbach GmbH

[Back to Top]

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**.

©1989-2020 Lauterbach GmbH

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.StreamMapTable** 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.

©1989-2020 Lauterbach GmbH

[Back to Top]

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)

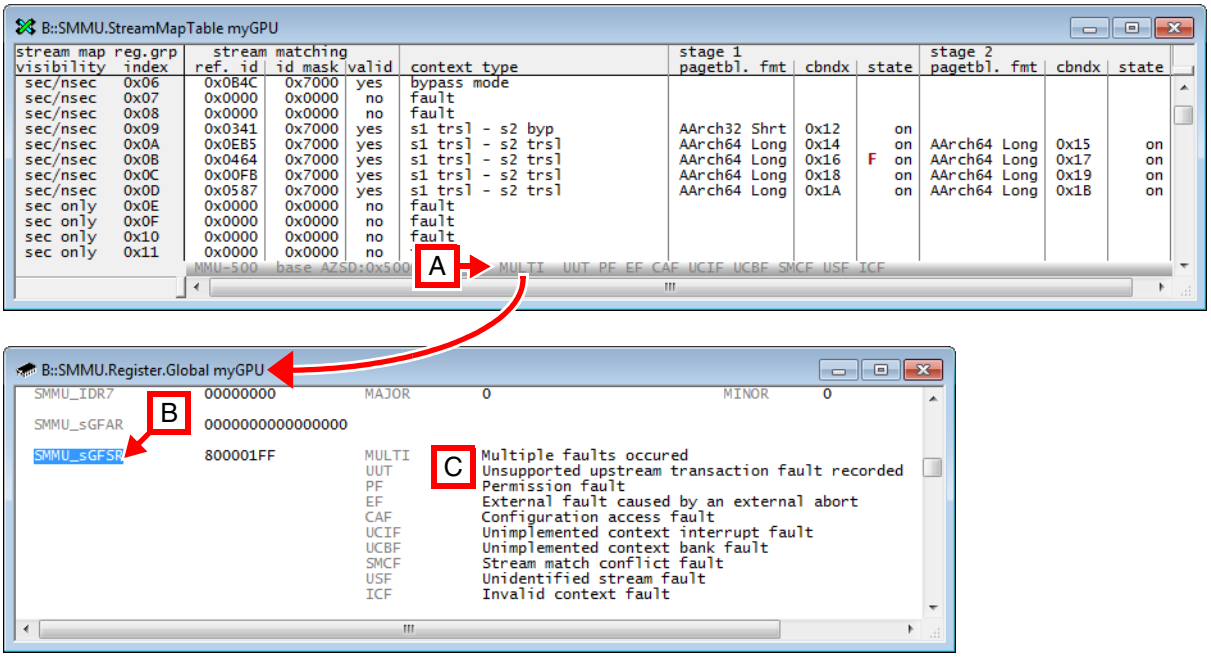
©1989-2020 Lauterbach GmbH

[Back to Top]

Codes in the gray window status bar at the bottom of the **SMMU.StreamMapTable** window indicate the current global fault status of the SMMU. These codes for the global faults are MULTI, UUT, PF, EF, CAF, UCIF, UCBF, SMCF, USF, ICF [**A**].

To view the descriptions of the global faults:

1. Double-click the gray window status bar to open the **SMMU.Register.Global** window [**A**].
2. Search for this register: SMMU_sGFSR [**B**]
   The global faults are described in the column on the right [**C**].



A Codes of global faults.

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.

NOTE:	A red letter in a **state** column of the **SMMU.StreamMapTable** window indicates a fault in a context bank. For descriptions of these faults, see **state** column.

©1989-2020 Lauterbach GmbH

Probe Cables

For debugging, two kinds of probe cables can be used to connect the debugger to the target: “Debug Cable” and “CombiProbe”.

The CombiProbe is mainly used in case a system trace port is available because it includes besides the debug interface a 4-bit wide trace port which is sufficient for program trace or for system trace.

For off-chip program 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 SWDP [ON | OFF]** alternatively 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).

©1989-2020 Lauterbach GmbH

Debug Cable

Adaptation for ARM Debug Cable: See **https://www.lauterbach.com/adarmdbg.html**.

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

For details on logical functionality, physical connector, alternative connectors, electrical characteristics, timing behavior and printing circuit design hints, refer to **"ARM JTAG Interface Specifications"** (app_arm_jtag.pdf).

CombiProbe

Adaptation for ARM CombiProbe: See **https://www.lauterbach.com/adarmcombi.html**.

The CombiProbe will always be delivered with 10-pin, 20-pin, 34-pin connectors. The CombiProbe cannot 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**.

©1989-2020 Lauterbach GmbH