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
History
Warning
Introduction
Brief Overview of Documents for New Users
Demo and Start-up Scripts
Quick Start of the JTAG Debugger
Troubleshooting
Communication between Debugger and Processor cannot be established
FAQ
Cortex-A/R (ARMv8, 32/64-bit)
Trace Extensions
Quick Start for Multicore Debugging
SMP Debugging - Quick Start
1. How to Debug a System with Multiple Identical Cores
2. Set up the SMP Debug Scenario
3. Enter Debug Mode
4. Switch Debug View between Cores
5. Write a Start-up Script Summary
AMP Debugging - Quick Start
1. How to Debug a System with Multiple Heterogenous Cores
2. Starting the TRACE32 GUIs
3. Master-Slave Concept
4. Setting up the Multicore Environment
5. Synchronized Go / Step / Break
6. Write a Start-up Script Summary
ARM Specific Implementations
AArch Mode Support

©1989-2019 Lauterbach GmbH
AArch64 and AArch32 Debugging 27
AArch64 and AArch32 Switching 28
TrustZone Technology 30
AArch64 Secure Model 30
AArch32 Secure Model 31
Debug Permission 31
Checking Debug Permission 31
Checking Secure State 32
Changing the Secure State from within TRACE32 32
AArch64 System Registers Access 33
AArch32 Coprocessor Registers Access 33
Accessing Cache and TLB Contents 33
Breakpoints and Vector Catch Register 33
Breakpoints and Secure Modes 33
big.LITTLE 34
Debugger Setup 34
Consequence for Debugging 35
Requirements for the Target Software 35
big.LITTLE MP 35
Breakpoints 36
Software Breakpoints 36
On-chip Breakpoints for Instructions 36
On-chip Breakpoints for Data 36
Example for Standard Breakpoints 37
Secure, Non-Secure, Hypervisor Breakpoints 38
Example for ETM Stopping Breakpoints 42
Access Classes 43
System Registers (AArch64 Mode) 51
Coprocessors (AArch32 Mode) 53
Accessing Memory at Run-time 56
Semihosting 60
AArch64 HLT Emulation Mode 61
AArch64 DCC Communication Mode (DCC = Debug Communication Channel) 62
AArch32 SVC (SWI) Emulation Mode 63
AArch32 DCC Communication Mode (DCC = Debug Communication Channel) 64
Virtual Terminal 66
Large Physical Address Extension (LPAE) 67
Consequence for Debugging 67
Virtualization Extension, Hypervisor 68
Consequence for Debugging 68
Debug Field 69
Run Mode 69
Run-time Measurements 70
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CLOCK</td>
<td>Inform debugger about core clock</td>
</tr>
<tr>
<td>SYStem.CONFIG.state</td>
<td>Display target configuration</td>
</tr>
<tr>
<td>SYStem.CONFIG</td>
<td>Configure debugger according to target topology</td>
</tr>
<tr>
<td>SYStem.CONFIG GICD</td>
<td>Generic Interrupt Controller Distributor (GIC)</td>
</tr>
<tr>
<td>SYStem.CONFIG GICR</td>
<td>Generic Interrupt Controller Redistributor</td>
</tr>
<tr>
<td>SYStem.CONFIG GICC</td>
<td>Generic Interrupt Controller physical CPU interface</td>
</tr>
<tr>
<td>SYStem.CONFIG GICH</td>
<td>Generic Interrupt Controller virtual interface control</td>
</tr>
<tr>
<td>SYStem.CONFIG GICV</td>
<td>Generic Interrupt Controller virtual CPU interface</td>
</tr>
<tr>
<td>SYStem.CONFIG SMMU</td>
<td>Internal use</td>
</tr>
<tr>
<td>SYStem.CPU</td>
<td>Select the used CPU</td>
</tr>
<tr>
<td>SYStem.CpuAccess</td>
<td>Run-time memory access (intrusive)</td>
</tr>
<tr>
<td>SYStem.JtagClock</td>
<td>Define the frequency of the debug port</td>
</tr>
<tr>
<td>SYStem.LOCK</td>
<td>Tristate the JTAG port</td>
</tr>
<tr>
<td>SYStem.MemAccess</td>
<td>Run-time memory access</td>
</tr>
<tr>
<td>SYStem.Mode</td>
<td>Establish the communication with the target</td>
</tr>
<tr>
<td>SYStem.Option</td>
<td>Special setup</td>
</tr>
<tr>
<td>SYStem.Option Address32</td>
<td>Define address format display</td>
</tr>
<tr>
<td>SYStem.Option AHBHPROT</td>
<td>Select AHB-AP HPROT bits</td>
</tr>
<tr>
<td>SYStem.Option AXIACEEnable</td>
<td>ACE enable flag of the AXI-AP</td>
</tr>
<tr>
<td>SYStem.Option AXICACHEFLAGS</td>
<td>Select AXI-AP CACHE bits</td>
</tr>
<tr>
<td>SYStem.Option AXIHPROT</td>
<td>Select AXI-AP HPROT bits</td>
</tr>
<tr>
<td>SYStem.Option BreakOS</td>
<td>Allow break during OS-unlock</td>
</tr>
<tr>
<td>SYStem.Option CacheStatusCheck</td>
<td>Check status bits during cache access</td>
</tr>
<tr>
<td>SYStem.Option CFLUSH</td>
<td>FLUSH the cache before step/go</td>
</tr>
<tr>
<td>SYStem.Option CLTAPKEY</td>
<td>Set key values for CLTAP operation</td>
</tr>
<tr>
<td>SYStem.Option CoreSightRESET</td>
<td>Assert CPU reset via CTRL/STAT</td>
</tr>
<tr>
<td>SYStem.Option DAPDBGPWRUPREQ</td>
<td>Force debug power in DAP</td>
</tr>
<tr>
<td>SYStem.Option DAP2DBGPWRUPREQ</td>
<td>Force debug power in DAP2</td>
</tr>
<tr>
<td>SYStem.Option DAPNOIRCHECK</td>
<td>No DAP instruction register check</td>
</tr>
<tr>
<td>SYStem.Option DAPREMAP</td>
<td>Rearrange DAP memory map</td>
</tr>
<tr>
<td>SYStem.Option DAPSYSYPWRUPREQ</td>
<td>Force system power in DAP</td>
</tr>
<tr>
<td>SYStem.Option DAP2SYSYPWRUPREQ</td>
<td>Force system power in DAP2</td>
</tr>
<tr>
<td>SYStem.Option DBGSPR</td>
<td>Use debugger view for SPR access</td>
</tr>
<tr>
<td>SYStem.Option DBGUNLOCK</td>
<td>Unlock debug register via OSLAR</td>
</tr>
<tr>
<td>SYStem.Option DCacheMaintenance</td>
<td>Data cache maintenance strategy</td>
</tr>
</tbody>
</table>
Overview of Debugging with Zones

SYStem.Option DEBUGPORTOptions Options for debug port handling 134
SYStem.Option DIAG Activate more log messages 135
SYStem.Option DUALPORT Implicitly use run-time memory access 135
SYStem.Option DisMode Define disassembler mode 136
SYStem.Option EnReset Allow the debugger to drive nRESET (nSRST) 137
SYStem.Option eXclusiveMONitor Support for exclusive monitors 137
SYStem.Option HRCWOverRide Enable override mechanism 137
SYStem.Option ICa cheMaintenance I-Cache maintenance strategy 138
SYStem.Option IMASKASM Disable interrupts while single stepping 138
SYStem.Option IMASKHLL Disable interrupts while HLL single stepping 139
SYStem.Option INTDIS Disable all interrupts 139
SYStem.Option MACHINESPACES Address extension for guest OSes 140
SYStem.Option MEMORYHPROT Select memory-AP HPROT bits 140
SYStem.Option MemStatusCheck Check status bits during memory access 141
SYStem.Option MMUPhysLogMemaccess Memory access preferences 141
SYStem.Option MMUSPACES Separate address spaces by space IDs 142
SYStem.Option N OMA Use alternative memory access 143
SYStem.Option NoPRCRReset Disable warm reset via PRCR 143
SYStem.Option OSUnlockCatch Use the 'OS Unlock Catch' debug event 144
SYStem.Option OVERLAY Extend debugger timeout 144
SYStem.Option PALLADIUM Enable overlay support 144
SYStem.Option PWRDWN Allow power-down mode 145
SYStem.Option PAN Overwrite CPSR.PAN setting 145
SYStem.Option PWRREQ Request core power 146
SYStem.Option ResBreak Halt the core after reset 147
SYStem.Option ResetDetection Choose method to detect a target reset 148
SYStem.RESetOut Assert nRESET/nSRST on JTAG connector 148
SYStem.Option RESetREGister Generic software reset 149
SYStem.Option RisingTDO Target outputs TDO on rising edge 149
SYStem.Option SLaVeSOFTRESet Allow soft reset of slave cores 150
SYStem.Option SMPMultipleCall Send start event to each SMP core 150
SYStem.Option SOFTLONG Use 32-bit access to set breakpoint 150
SYStem.Option SOFTQUAD Use 64-bit access to set breakpoint 150
SYStem.Option STEPSOFT Use software breakpoints for ASM stepping 151
SYStem.Option SOFTWORD Use 16-bit access to set breakpoint 151
SYStem.Option TURBO Disable cache maintenance during memory access 151
SYStem.state Display SYStem window 152
SYStem.Option SYSPWRUPREQ Force system power 152
SYStem.Option TRST Allow debugger to drive TRST 153
SYStem.Option WaitIDCODE IDCODE polling after deasserting reset 153
SYStem.Option WaitReset Wait with JTAG activities after deasserting reset 154
SYStem.Option ZoneSPACES Enable symbol management for ARM zones 156

©1989-2019 Lauterbach GmbH
ARM specific Benchmarking Commands .......................................................... 163
BMC.<counter>.CountEL<x> Select exception level events to be counted 163

ARM specific TrOnchip Commands ............................................................... 165
TrOnchip.ContextID Enable context ID comparison 166
TrOnchip.CONVert Allow extension of address range of breakpoint 166
TrOnchip.MatchASID Extend on-chip breakpoint/trace filter by ASID 167
TrOnchip.MatchZone Extend on-chip breakpoint/trace filter by zone 168
TrOnchip.RESERVE Exclude breakpoint or watchpoint from debugger usage 169
TrOnchip.RESet Set on-chip trigger to default state 169
TrOnchip.Set Set bits in the vector catch register 170
TrOnchip.VarCONVert Convert breakpoints on scalar variables 173
TrOnchip.state Display on-chip trigger window 174

Cache Analysis and Maintenance ................................................................. 175
TRACE32 Cache Support by CPU Type 176

CPU specific MMU Commands ................................................................. 177
MMU.DUMP Page wise display of MMU translation table 177
MMU.List Compact display of MMU translation table 188
MMU.SCAN Load MMU table from CPU 190
TRACE32 TLB Support by CPU Type 192

CPU specific SMMU Commands ................................................................. 193
SMMU Hardware system MMU (SMMU) 193
SMMU.ADD Define a new hardware system MMU 197
SMMU.Clear Delete an SMMU 198
SMMU.Register Peripheral registers of an SMMU 199
SMMU.Register.ContextBank Display registers of context bank 200
SMMU.Register.Global Display global registers of SMMU 201
SMMU.Register.StreamMapRegGrp Display registers of an SMRG 202
SMMU.RESet Delete all SMMU definitions 203
SMMU.SSDtable Display security state determination table 204
SMMU.StreamMapRegGrp Access to stream map table entries 206
SMMU.StreamMapRegGrp.ContextReg Display context bank registers 207
SMMU.StreamMapRegGrp.Dump Page-wise display of SMMU page table 209
SMMU.StreamMapRegGrp.list List the page table entries 211
SMMU.StreamMapTable Display a stream map table 212

Target Adaption ......................................................................................... 219
Probe Cables 219
Interface Standards JTAG, Serial Wire Debug, cJTAG 219
Connector Type and Pinout 220
Debug Cable 220
Support ................................................................. 221
Available Tools .................................................. 221
32-Bit Compilers .................................................. 223
64-Bit Compilers .................................................. 225
Target Operating Systems .................................... 225
Boot Loaders ....................................................... 225
3rd-Party Tool Integrations .................................... 226

Products ................................................................. 227
Product Information ............................................. 227
Order Information ............................................... 228
History

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”.

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.
<table>
<thead>
<tr>
<th>WARNING:</th>
<th>To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Recommendation for the software start:</td>
</tr>
<tr>
<td></td>
<td>1. Disconnect the debug cable from the target while the target power is off.</td>
</tr>
<tr>
<td></td>
<td>2. Connect the host system, the TRACE32 hardware and the debug cable.</td>
</tr>
<tr>
<td></td>
<td>3. Power ON the TRACE32 hardware.</td>
</tr>
<tr>
<td></td>
<td>4. Start the TRACE32 software to load the debugger firmware.</td>
</tr>
<tr>
<td></td>
<td>5. Connect the debug cable to the target.</td>
</tr>
<tr>
<td></td>
<td>6. Switch the target power ON.</td>
</tr>
<tr>
<td></td>
<td>7. Configure your debugger e.g. via a start-up script.</td>
</tr>
<tr>
<td></td>
<td>Power down:</td>
</tr>
<tr>
<td></td>
<td>1. Switch off the target power.</td>
</tr>
<tr>
<td></td>
<td>2. Disconnect the debug cable from the target.</td>
</tr>
<tr>
<td></td>
<td>3. Close the TRACE32 software.</td>
</tr>
<tr>
<td></td>
<td>4. Power OFF the TRACE32 hardware.</td>
</tr>
</tbody>
</table>
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 instructs 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.view** dialog:
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:
You can also inspect the demo folder manually in the root installation directory of TRACE32. The `~/demo/arm64/` and the `~/demo/arm/` folders contain:

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardware/</td>
<td>Ready-to-run debugging and flash programming demos for evaluation boards. <strong>Recommended for getting started!</strong></td>
</tr>
<tr>
<td>bootloader</td>
<td>Examples for uboot, uefi and other bootloaders</td>
</tr>
<tr>
<td>compiler/</td>
<td>Hardware independent compiler examples.</td>
</tr>
<tr>
<td>etc/</td>
<td>Various examples, e.g. data trace, terminal application, …</td>
</tr>
<tr>
<td>fdx/</td>
<td>Example applications for the FDX feature.</td>
</tr>
<tr>
<td>flash/</td>
<td>Binaries for target based programming and example declarations for internal flash.</td>
</tr>
<tr>
<td>kernel/</td>
<td>Various OS Awareness examples.</td>
</tr>
</tbody>
</table>
Quick Start of the JTAG Debugger

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 specific settings.

   SYSTEM.CPU <cputype> ; Default <cputype>: 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 is selected, so the following SYSTEM.CONFIG commands would not be required. However, for generic CPU 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 were set. If not, you have to set them manually:

   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.
3. Enter debug mode.

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

This command resets the CPU 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 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
Break.direct
```

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. Refer to “Supported Compilers” to find the command that is necessary for your 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 value are only examples.

```
RESet
SYStem.CPU CortexA53 // Select your 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.
```
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.     // Open source code window *)
List.Mix                     // Show clearly arranged
// peripherals in 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
```
A debug session might look like this:
Troubleshooting

Communication between Debugger and Processor cannot be established

Typically the **SYStem.Mode Up** command is the first command of a debug session for which communication with the target (e.g. evaluation board) is required. If you receive error messages like “debug port fail” or “debug port time out” while executing this command, this may have the reasons below. “target processor in reset” is just a follow-up error message. Open the **AREA.view** window to 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 to the JTAG connector.
- There is the need to enable (jumper) the debug features on the target. It will e.g. not work if nTRST signal is directly connected to ground on target side.
- The target is in an unrecoverable state. Re-power your target and try again.
- The target cannot communicate with the debugger while in reset. Try **SYStem.Mode Attach** followed by “Break” instead of **SYStem.Up** or use **SYStem.Option EnReset OFF**.
- The default frequency of the JTAG/SWD/cJTAG debug port is too high, especially if you emulate your core or if you use an FPGA based target. In this case try **SYStem.JtagClock 50kHz** and optimize the speed when you got it working.
- The core has no power.
- The core has no clock.
- The core is kept in reset.
- 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 ResBreak** and/or **SYStem.Option WaitReset**. As an alternative, try to use **SYStem.Mode Attach**.
- 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**.
The debugger is accessed via Internet/VPN and the performance is very slow. What can be done to improve debug performance?

The main cause for bad debug performance via Internet or VPN are low data throughput and high latency. The ways to improve performance by the debugger are limited:

In PRACTICE scripts, use "SCREEN.OFF" at the beginning of the script and "SCREEN.ON" at the end. "SCREEN.OFF" will turn off screen updates. Please note that if your program stops (e.g. on error) without executing "SCREEN.OFF", some windows will not be updated.

"SYStem.POLLING SLOW" will set a lower frequency for target state checks (e.g. power, reset, jtag state). It will take longer for the debugger to recognize that the core stopped on a breakpoint.

"SETUP.URATE 1.s" will set the default update frequency of Data.List/Data.dump/Variable windows to 1 second (the slowest possible setting).

prevent unneeded memory accesses using "MAP.UPDATEONCE [address-range]" for RAM and "MAP.CONST [address--range]" for ROM/FLASH. Address ranged with "MAP.UPDATEONCE" will read the specified address range only once after the core stopped at a breakpoint or manual break. "MAP.CONST" will read the specified address range only once per SYStem.Mode command (e.g. SYStem.Up).
**What can be the reasons why setting a software breakpoint fails?**

Setting a software breakpoint can fail when the target HW is not able to implement the wanted breakpoint.

Possible reasons:

The wanted breakpoint needs special features that are only possible to realize by the trigger unit inside the controller. Example: Read, write and access (Read/Write) breakpoints ("type" in Break.Set window). Breakpoints with checking in real-time for data-values ("Data"). Breakpoints with special features ("action") like TriggerTrace, TraceEnable, TraceOn/TraceOFF.

TRACE32 can not change the memory. Example: ROM and Flash when no preparation with FLASH.Create, FLASH.TARGET and FLASH.AUTO was made. All type of memory if the memory device is missing the necessary control signals like WriteEnable or settings of registers and SpecialFunctionRegisters (SFR).

Contrary settings in TRACE32. Like: MAP.BOnchip for this memory range. Break.SELect.<breakpoint-type> Onchip (HARD is only available for ICE and FIRE).

RTOS and MMU: If the memory can be changed by Data.Set but the breakpoint doesn't work it might be a problem of using an MMU on target when setting the breakpoint to a symbolic address that is different than the writable and intended memory location.
The debugger is accessed via Internet/VPN and the performance is very slow. What can be done to improve debug performance?

The main cause for bad debug performance via Internet or VPN are low data throughput and high latency. The ways to improve performance by the debugger are limited:

In PRACTICE scripts, use "SCREEN.OFF" at the beginning of the script and "SCREEN.ON" at the end. "SCREEN.OFF" will turn off screen updates. Please note that if your program stops (e.g. on error) without executing "SCREEN.OFF", some windows will not be updated.

"SYStem.POLLING SLOW" will set a lower frequency for target state checks (e.g. power, reset, jtag state). It will take longer for the debugger to recognize that the core stopped on a breakpoint.

"SETUP.URATE 1.s" will set the default update frequency of Data.List/Data.dump/Variable windows to 1 second (the slowest possible setting).

prevent unneeded memory accesses using "MAP.UPDATEONCE [address-range]" for RAM and "MAP.CONST [address--range]" for ROM/FLASH. Address ranged with "MAP.UPDATEONCE" will read the specified address range only once after the core stopped at a breakpoint or manual break. "MAP.CONST" will read the specified address range only once per SYStem.Mode command (e.g. SYStem.Up).

Trace Extensions

There following types of trace extensions are available on the 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)
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.
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:

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

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

If you have no predefined SMP chip 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 CPUs 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:


3. Enter Debug Mode

This will bring all cores into debug mode that have been assigned to the SMP system using the \texttt{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, \texttt{SYSTEM.Mode Up} may fail with an error. In this case try to attach to the CPU and to stop it:

\begin{verbatim}
SYSTEM.Up
SYSTEM.Mode Attach ; Attach without reset
Break.direct ; Enter debug mode
\end{verbatim}

4. Switch Debug View between Cores

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

\begin{verbatim}
CORE.select 0. ; select core 0
Register.view ; view registers of current core 0
Register.view /CORE 1. ; view registers of core 1
\end{verbatim}

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:

\begin{verbatim}
CD.DO <smp_setup>.cmm
\end{verbatim}
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:

![Diagram of AMP system with two independent SMP systems and two single cores]

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)](image)

SMP System GUI (Linux SMP-2)

![SMP System GUI (Linux SMP-4)](image)

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 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
PORT=<portnumber>         ; enable InterCom
                         ; e.g. PORT=20000
```

©1989-2019 Lauterbach GmbH
3. Master-Slave Concept

An AMP system may share certain hardware like reset lines. 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.:

```plaintext
; 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 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.:

```plaintext
CORE0 SYStem.CONFIG CORE 1. 1. ; GUI0: 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.
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 50 (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
```
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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AArch64</td>
<td>Supports execution of 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).</td>
</tr>
<tr>
<td>AArch32</td>
<td>Supports execution of 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.</td>
</tr>
</tbody>
</table>

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 mode. It is recommended to use SYStem.Option DisMode AUTO and to let the debugger decide which disassembly mode is used. To choose on your own 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 the 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.
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 and 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 is 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
For an application there are two possibilities to switch the AArch mode. An application might configure the lower Exception Levels (ELx) and 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, e.g. 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 EL1 using an exception return (ERET) and transition back to AArch64 EL3 using a secure monitor call (SMC).

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 a.s.o.)
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:

![Diagram of TrustZone model]

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

![Diagram showing non-secure world and secure world modes](image)

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 **SYSstem.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.
Checking Secure State

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

In 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 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 to AREA.view
&secureMode=Register(NS) ; Read current secure mode 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.
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 or non-secure 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.
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):

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

```plaintext
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>
```

**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):

```plaintext
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>
```
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.

<table>
<thead>
<tr>
<th>Family</th>
<th>On-chip breakpoints</th>
<th>Instruction breakpoints</th>
<th>Read/Write breakpoint</th>
<th>Data Value breakpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-A3x</td>
<td>2 ... 16 instruction</td>
<td>2 ... 16 single address</td>
<td>2 ... 16 single address</td>
<td>—</td>
</tr>
<tr>
<td>Cortex-A5x</td>
<td>2 ... 16 read/write</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cortex-A7x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example for Standard Breakpoints

Assume you have a target with

- FLASH from 0x0--0xffffff
- RAM from 0x100000--0x11ffff

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

```
Map.BOnchip 0x0--0xffffff
```

**The following standard breakpoint combinations are possible.**

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

   ```
   Break.Set 0x100000 /Program ; Software BP on address
   Break.Set myFunc /Program ; Software BP on symbol
   Break.Set 0x100 /Program ; On-chip BP on 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
   ```
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 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
```

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

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

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

Now breakpoints can be 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: or H: access class is automatically set
Var.Break.Set struct1 ; Set Read/Write breakpoints to the whole structure struct1. The breakpoint is either a secure/non-secure or hypervisor type.
```

Example 1: Load secure application and set breakpoints

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

// Load demo application and tell TRACE32 that it is secure (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. An application is loaded and TRACE32 is advised by the access class “Z:” at the end of the `Data.LOAD.ELF` command that this application runs in secure mode.

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

![Secure breakpoint(s)]

**Set Breakpoints and Enforce Secure Mode**

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

```
Break.Set Z:main           ; Enforce secure breakpoint on main()
Break.Set N:main           ; Enforce non-secure breakpoint on main()
Break.Set H:main           ; Enforce hypervisor breakpoint on main()
```

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

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

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

<table>
<thead>
<tr>
<th>BP access class</th>
<th>EL3 sec</th>
<th>EL3 nsec</th>
<th>EL2</th>
<th>EL1/EL0 sec</th>
<th>EL1/EL0 nsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>M: (secure)</td>
<td>halt</td>
<td>halt</td>
<td>no halt</td>
<td>no halt</td>
<td>no halt</td>
</tr>
<tr>
<td>Z: (secure)</td>
<td>no halt</td>
<td>no halt</td>
<td>no halt</td>
<td>halt</td>
<td>no halt</td>
</tr>
<tr>
<td>H: (hypervisor)</td>
<td>no halt</td>
<td>no halt</td>
<td>halt</td>
<td>no halt</td>
<td>no halt</td>
</tr>
<tr>
<td>N: (non-secure)</td>
<td>no halt</td>
<td>no halt</td>
<td>no halt</td>
<td>no halt</td>
<td>halt</td>
</tr>
</tbody>
</table>
For 32-bit modes "Z," "N:" or "H:" determine in which mode the CPU shall stop:

<table>
<thead>
<tr>
<th>BP access class</th>
<th>mon sec</th>
<th>mon nsec</th>
<th>hyp</th>
<th>svc/usr sec</th>
<th>svc/usr nsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z: (secure)</td>
<td>halt</td>
<td>halt</td>
<td>no halt</td>
<td>halt</td>
<td>no halt</td>
</tr>
<tr>
<td>H: (hypervisor)</td>
<td>no halt</td>
<td>no halt</td>
<td>halt</td>
<td>no halt</td>
<td>no halt</td>
</tr>
<tr>
<td>N: (non-secure)</td>
<td>no halt</td>
<td>no halt</td>
<td>no halt</td>
<td>no halt</td>
<td>halt</td>
</tr>
</tbody>
</table>

**Example 2:** Load secure application and set hypervisor breakpoint.

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

// 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 it a secure breakpoint. Two breakpoints are defined as hypervisor breakpoints with “H:”, i.e. the symbol information is explicitly overwritten. The `Break.List` now shows a mixed breakpoint setup:

![Breakpoint List](image)

**NOTE:**

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

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

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

Configuration of the Target CPU

The configuration of the onchip breakpoints will be placed in the breakpoint/watchpoint registers of the 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 mode.
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.

  ```
  SYSystem.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:

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

It’s recommended to add both configuration commands to your PRACTICE start-up script (*.cmm).

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).
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

<table>
<thead>
<tr>
<th>Access Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Absolute addressing (physical address)</td>
</tr>
<tr>
<td>AHB, AHB2</td>
<td>See DAP.</td>
</tr>
<tr>
<td>APB, APB2</td>
<td>See DAP.</td>
</tr>
<tr>
<td>AXI, AXI2</td>
<td>See DAP.</td>
</tr>
<tr>
<td>C14</td>
<td>Access to C14-Coprocessor register. Its recommended to only use this in AArch32 mode.</td>
</tr>
<tr>
<td>C15</td>
<td>Access to C15-Coprocessor register. Its recommended to only use this in AArch32 mode.</td>
</tr>
<tr>
<td>D</td>
<td>Data Memory</td>
</tr>
<tr>
<td>Access Class</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>DAP, DAP2, AHB, AHB2, APB, APB2, AXI, AXI2</td>
<td>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 &lt;mem_ap#&gt; -&gt; “DAP” SYStem.CONFIG AHBACCESSPORT &lt;mem_ap#&gt; -&gt; “AHB” SYStem.CONFIG APBACCESSPORT &lt;mem_ap#&gt; -&gt; “APB” SYStem.CONFIG AXIACCESSPORT &lt;mem_ap#&gt; -&gt; “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 &lt;mem_ap#&gt;) available in case there are two DAPs which needs to be controlled by the debugger.</td>
</tr>
<tr>
<td>E</td>
<td>Run-time memory access (see SYStem.CpuAccess and SYStem.MemAccess)</td>
</tr>
<tr>
<td>M ARmv8-A only</td>
<td>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.</td>
</tr>
<tr>
<td>H</td>
<td>EL2/Hypervisor Mode (devices having Virtualization Extension)</td>
</tr>
<tr>
<td>J</td>
<td>Java Code (8-bit)</td>
</tr>
<tr>
<td>N</td>
<td>EL0/1 Non-Secure Mode (TrustZone devices)</td>
</tr>
<tr>
<td>P</td>
<td>Program Memory</td>
</tr>
<tr>
<td>R</td>
<td>AArch32 ARM Code (A32, 32-bit instr. length)</td>
</tr>
<tr>
<td>S</td>
<td>Supervisor Memory (privileged access)</td>
</tr>
<tr>
<td>SPR ARmv8-A only</td>
<td>Access to System Register, Special Purpose Registers and System Instructions. Its recommended to only use this in AArch64 mode.</td>
</tr>
<tr>
<td>T</td>
<td>AArch32 Thumb Code (T32, 16-bit instr. length)</td>
</tr>
<tr>
<td>U</td>
<td>User Memory (non-privileged access) not yet implemented; privileged access will be performed.</td>
</tr>
<tr>
<td>USR</td>
<td>Access to Special Memory via User-Defined Access Routines</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Access Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM</td>
<td>Virtual Memory (memory on the debug system)</td>
</tr>
<tr>
<td>X ARMv8-A only</td>
<td>AArch64 ARM64 Code (A64, 32-bit instr. length)</td>
</tr>
<tr>
<td>Z</td>
<td>Secure Mode (TrustZone devices)</td>
</tr>
</tbody>
</table>
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.
How to Create Valid Access Class Combinations

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

Rules to create a valid access class combination:

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

Memory Access through CPU (CPU View)

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

Example combinations:

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

ZHR, NHR  The ArmV8.4-A extension does not include a secure AArch32 hypervisor. Therefore any 32-bit access class specifiers (R, T, TE) are illegal.
ZHT, NHT  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.
ZHTE, NHTE

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
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
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.

<table>
<thead>
<tr>
<th>User input at the command line</th>
<th>Expansion by TRACE32</th>
<th>These access classes are added because...</th>
</tr>
</thead>
<tbody>
<tr>
<td>List.Mix</td>
<td>NSR:</td>
<td>N: … the CPU is in non-secure mode.</td>
</tr>
<tr>
<td>(see also illustration below)</td>
<td></td>
<td>S: … the CPU is in supervisor mode.</td>
</tr>
<tr>
<td>Data.dump A:0x0</td>
<td>ANSD:0x0</td>
<td>N: … the CPU is in non-secure mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S: … the CPU is in supervisor mode.</td>
</tr>
<tr>
<td>Data.dump Z:0x0</td>
<td>ZSD:0x0</td>
<td>S: … the CPU is in supervisor mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>D: … data is viewed (not code).</td>
</tr>
</tbody>
</table>

**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.
System Registers (AArch64 Mode)

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>_<CRn>_<CRm>_<op2> | MSR <S>_<op1>_<CRn>_<CRm>_<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 (4bit) in the SPR address. The parameter arrangement is:

<table>
<thead>
<tr>
<th>SPR:</th>
<th>op0</th>
<th>op1</th>
<th>CRn</th>
<th>CRm</th>
<th>op2</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRS</td>
<td>&lt;Xt&gt;, &lt;S&gt;<em>&lt;op1&gt;</em>&lt;CRn&gt;<em>&lt;CRm&gt;</em>&lt;op2&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR</td>
<td>&lt;S&gt;<em>&lt;op1&gt;</em>&lt;CRn&gt;<em>&lt;CRm&gt;</em>&lt;op2&gt;, &lt;Xt&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```
Data.Set SPR:0x34210 %Quad 0x200800000000 // 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
```
Coproceessors (AArch32 Mode)

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.

<\textbf{MCR}|\textbf{MRC} p14, <opc1>, <Rt>, <CRn>, <CRm>, <opc2>

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

\begin{center}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{C14:} & \textbf{op1} & \textbf{op2} & \textbf{CRm} & \textbf{CRn} \\
\hline
14 & 12 & 10 & 8 & 7 & 4 & 3 & 0 \\
\hline
\end{tabular}
\end{center}

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

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

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

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

<table>
<thead>
<tr>
<th>C15:</th>
<th>op1</th>
<th>op2</th>
<th>CRm</th>
<th>CRn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

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.
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:

Fixed to 1, 64-bit access
Fixed to 0, not used for 64-bit access

<table>
<thead>
<tr>
<th>C15:</th>
<th>1</th>
<th>op1</th>
<th>0</th>
<th>CRm</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>14</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

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 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.
Accessing Memory at Run-time

This section 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.
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:<addr>” can be extended to “Data.dump ENSD:<addr>”.

Intrusive run-time access

To activate intrusive memory accesses, use the command SYStem.CpuAccess Enable.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CpuAccess Enable</td>
<td>Intrusive run-time memory access, CPU is periodically stopped / restarted</td>
</tr>
<tr>
<td>Data.dump E:0x100</td>
<td>Intrusive access via CPU. Prefix “E”</td>
</tr>
<tr>
<td>Var.view %E myVar</td>
<td>is required to read 0x100 or myVar</td>
</tr>
</tbody>
</table>
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:

```
SYSystem.CONFIG MEMORYACCESSPORT 1. ; Define memory access port and AXI
SYSystem.CONFIG AXIACCESSPORT 1. ; access port (e.g. port number 1)
Data.dump EAXI:<addr> ; Run-time access via AXI. Prefix "E"
Data.dump EAXI:myVar ; is required to read <addr> or myVar
```

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 `SYSystem.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.

```
SYSystem.CONFIG MEMORYACCESSPORT 1. ; Define memory access port and AHB
// SYSystem.CONFIG AHBAACCESSPORT 1. ; or AXI access port
SYSystem.CONFIG AXIACCESSPORT 1.
SYSystem.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 `SYSystem.Option AXIACEEnable ON`

```
SYSystem.CONFIG MEMORYACCESSPORT 1. ; Define memory access port and AXI
SYSystem.CONFIG AXIACCESSPORT 1. ; access port (e.g. port number 1)
SYSystem.Option AXIACEEnable ON ; Enable cache coherent transactions
SYSystem.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
```
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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSlation.Create</td>
<td>If the CPU has never stopped, set the translation manually.</td>
</tr>
<tr>
<td>MMU.SCAN</td>
<td>Scan static page tables into the debugger while the CPU is stopped.</td>
</tr>
<tr>
<td>TRANSlation.TableWalk</td>
<td>Use if CPU stops and page tables are modified frequently (e.g. by OS).</td>
</tr>
</tbody>
</table>
Semihosting is a technique for application program running on an ARM processor to communicate with the host computer of the debugger. This way the application can use the I/O facilities of the host computer like keyboard input, screen output, and file I/O. This is especially useful if the target platform does not yet provide these I/O facilities or in order to output additional debug information in printf() style.

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.
AArch64 HLT Emulation Mode

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
```bash
~/demo/arm64/etc/semihosting_arm_emulation/aarch64/halt_armv8.cmm
```
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`
AArch32 SVC (SWI) Emulation Mode

A breakpoint placed on the SVC exception entry stops the application. The debugger handles the request while the application is stopped, provides the required communication with the host, and restarts the application at the address which was stored in the link register 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
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.
For examples and source codes, see ~/demo/arm64/etc/semihosting_trace32_dcc/aarch32

DCC Mode (TRACE32)

Application (termdemo.c)
TRACE32 Semihosting Functions (t32term.c, t32term.h)
Core Specific DCC Driver (t32term.c, ..., c)
Software MRC/MCR
DCC Register
Core 0
Target

Monitor
Keyboard

Filesystem

Debugger ICD

USB ETH

TRACE32 PowerView

PC or Workstation
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:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DCC</strong></td>
<td>Use the DCC port of the JTAG interface to transfer 1 byte at once.</td>
</tr>
<tr>
<td><strong>DCC3</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>DCC4A</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>DCC4B</strong></td>
<td>Four byte binary mode. Used to transfer non-ASCII 32-bit data (e.g. to or from a file).</td>
</tr>
</tbody>
</table>

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**
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.
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.

![Diagram of MMU stages](image)

**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 physical address, use the ‘I:’ access class.
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:

<table>
<thead>
<tr>
<th>Run state</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>running (power down)</td>
<td>The core is not powered.</td>
</tr>
<tr>
<td>running (secured)</td>
<td>Core is secured. Access by debugger not possible.</td>
</tr>
<tr>
<td>running (sticky reset)</td>
<td>Core is running with sticky reset bit set.</td>
</tr>
<tr>
<td>running (OS lock)</td>
<td>Core is running with OS-lock bit set.</td>
</tr>
<tr>
<td>running (OS lock/catch)</td>
<td>Core is running with OS-lock bit set and OS catch event is enabled for this core when SYStem.Option BreakOS is OFF.</td>
</tr>
<tr>
<td>no access to cpu</td>
<td>The core seems to be running or powered but the debug registers are not accessible.</td>
</tr>
<tr>
<td>disabled</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.
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 | COnponents |

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.

<table>
<thead>
<tr>
<th>&lt;tab&gt;</th>
<th>Opens the SYStem.CONFIG.state window on the specified tab. For tab descriptions, see below.</th>
</tr>
</thead>
</table>
| DebugPort  (default) | The DebugPort tab informs the debugger about the debug connector type and the communication protocol it shall use.  
  For descriptions of the commands on the DebugPort tab, see DebugPort. |
### Jtag

The **Jtag** tab informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip.

For descriptions of the commands on the **Jtag** tab, see [Jtag](#).

### MultiTap

Informs the debugger about the existence and type of a System/Chip Level Test Access Port. The debugger might need to control it in order to reconfigure the JTAG chain or to control power, clock, reset, and security of different chip components.

For descriptions of the commands on the **MultiTap** tab, see [Multitap](#).

### 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>:**

```
(C)JTAGFLAGS <flags>
CJTAGTCA <value>
CONNECTOR [MIPI34 | MIPI20T]
CORE <core> <chip>
CoreNumber <number>
DEBUGPORT [DebugCable0 | DebugCableA | DebugCableB]
DEBUGPORTTYPE [JTAG | SWD | CJTAG | CJTAGSWD]
NIDNTTRSTTORST [ON | OFF]
```
<parameter>
(DebugPort cont.)
NIDNTPSRISINGEDGE [ON | OFF]
NIDNTRSTPOLARITY [High | Low]
PortSHaRing [ON | OFF | Auto]
Slave [ON | OFF]
SWDP [ON | OFF]
SWDPIDLEHIGH [ON | OFF]
SWDPTargetSel <value>
TriState [ON | OFF]

<parameter>
(JTAG)
CHIPDRLENGTH <bits>
CHIPDRPATTERN [Standard | Alternate <pattern>]
CHIPDRPRES <bits>
CHIPIRLENGTH <bits>
CHIPIRPATTERN [Standard | Alternate <pattern>]
CHIPIRPOST <bits>
CHIPIRPRE <bits>

<parameter>
(JTAG cont.)
DAP2DRPOST <bits>
DAP2DRPRES <bits>
DAP2IRPOST <bits>
DAP2IRPRES <bits>
DAPDRPOST <bits>
DAPDRPRES <bits>
DAPIRPOST <bits>
DAPIRPRES <bits>

<parameter>
(JTAG cont.)
DRPOST <bits>
DRPRES <bits>
ETBDRPOST <bits>
ETBDRPRES <bits>
ETBIRPOST <bits>
ETBIRPRES <bits>
IRPOST <bits>
IRPRES <bits>

<parameter>
(JTAG cont.)
NEXTDRPOST <bits>
NEXTDRPRES <bits>
NEXTIRPOST <bits>
NEXTIRPRES <bits>
RTPDRPOST <bits>
RTPDRPRES <bits>
RTPIRPOST <bits>
RTPIRPRES <bits>

<parameter>
(JTAG cont.)
Slave [ON | OFF]
TAPState <state>
TCKLevel <level>
TriState [ON | OFF]
<parameter>: CFGCONNECT <code>
    DAP2TAP <tap>
    DAPTPAP <tap>
    DEBUGTAP <tap>
    ETBTPAP <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>: AHBACCESSPORT <port>
    APBACCESSPORT <port>
    AXIACCESSPORT <port>
    COREJTAGPORT <port>
    DAP2AHBACCESSPORT <port>
    DAP2APBACCESSPORT <port>
    DAP2AXIACCESSPORT <port>
    DAP2COREJTAGPORT <port>

<parameter>: DAP2DEBUGACCESSPORT <port>
    DAP2JTAGPORT <port>
    DAP2AHBACCESSPORT <port>
    DEBUGACCESSPORT <port>
    JTAGACCESSPORT <port>
    MEMORYACCESSPORT <port>

<parameter>: ADTF.Base <address>
    ADTF.RESET
    AET.Base <address>
    AET.RESET
    BMC.Base <address>
    BMC.RESET
    CMI.Base <address>
    CMI.RESET

<parameter>: CMI.TraceID <id>
    COREDEBUG.Base <address>
    COREDEBUG.RESET
    CTI.Base <address>
    CTI.Config [NONE | ARMV1 | ARMPPostInit | OMAP3 | TMS570 | CortexV1 | QV1]
    CTI.RESET
    DRM.Base <address>
    DRM.RESET
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTM</td>
<td>RESET, Type [None</td>
</tr>
<tr>
<td>DWT</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>EPM</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ETB2AXI</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>EPM</td>
<td>RESET</td>
</tr>
<tr>
<td>EPM</td>
<td>RESET</td>
</tr>
<tr>
<td>ETB</td>
<td>ATBSource &lt;source&gt;</td>
</tr>
<tr>
<td>ETB</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ETB</td>
<td>NoFlush [ON</td>
</tr>
<tr>
<td>ETB</td>
<td>RESET</td>
</tr>
<tr>
<td>ETB</td>
<td>Size &lt;size&gt;</td>
</tr>
<tr>
<td>ETF</td>
<td>ATBSource &lt;source&gt;</td>
</tr>
<tr>
<td>ETF</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ETF</td>
<td>RESET</td>
</tr>
<tr>
<td>ETM</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ETM</td>
<td>RESET</td>
</tr>
<tr>
<td>ETR</td>
<td>ATBSource &lt;source&gt;</td>
</tr>
<tr>
<td>ETR</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ETR</td>
<td>RESET</td>
</tr>
<tr>
<td>FUNNEL</td>
<td>ATBSource &lt;sourcelist&gt;</td>
</tr>
<tr>
<td>FUNNEL</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>FUNNEL</td>
<td>Name &lt;string&gt;</td>
</tr>
<tr>
<td>FUNNEL</td>
<td>PROGrammable [ON</td>
</tr>
<tr>
<td>HSM</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>HSM</td>
<td>RESET</td>
</tr>
<tr>
<td>HTM</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>HTM</td>
<td>RESET</td>
</tr>
<tr>
<td>HTM</td>
<td>Type [CoreSight</td>
</tr>
<tr>
<td>ICE</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ICE</td>
<td>RESET</td>
</tr>
<tr>
<td>ITM</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>ITM</td>
<td>RESET</td>
</tr>
<tr>
<td>L2CACHE</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>L2CACHE</td>
<td>RESET</td>
</tr>
<tr>
<td>L2CACHE</td>
<td>Type [NONE</td>
</tr>
<tr>
<td>OCP</td>
<td>Base &lt;address&gt;</td>
</tr>
<tr>
<td>OCP</td>
<td>RESET</td>
</tr>
<tr>
<td>OCP</td>
<td>TraceID &lt;id&gt;</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>OCP.Type</td>
<td>&lt;type&gt;</td>
</tr>
<tr>
<td>PMI.Base</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>PMI.RESET</td>
<td></td>
</tr>
<tr>
<td>PMI.TraceID</td>
<td>&lt;id&gt;</td>
</tr>
<tr>
<td>RTP.Base</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>RTP.PerBase</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>RTP.RamBase</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>RTP.RESET</td>
<td></td>
</tr>
<tr>
<td>SC.Base</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>SC.RESET</td>
<td></td>
</tr>
<tr>
<td>SC.TraceID</td>
<td>&lt;id&gt;</td>
</tr>
<tr>
<td>STM.Base</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>STM.Mode</td>
<td>[NONE</td>
</tr>
<tr>
<td>STM.RESET</td>
<td></td>
</tr>
<tr>
<td>STM.Type</td>
<td>[None</td>
</tr>
<tr>
<td>TPIU.ATBSource</td>
<td>&lt;source&gt;</td>
</tr>
<tr>
<td>TPIU.Base</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>TPIU.RESET</td>
<td></td>
</tr>
<tr>
<td>TPIU.Type</td>
<td>[CoreSight</td>
</tr>
<tr>
<td>BMCBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>BYPASS</td>
<td>&lt;seq&gt;</td>
</tr>
<tr>
<td>COREBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>CTIBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>CTICONFIG</td>
<td>[NONE</td>
</tr>
<tr>
<td>DEBUGBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>DTMCONFIG</td>
<td>[ON</td>
</tr>
<tr>
<td>DTMETBFUNNELPORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>DTMFUNNEL2PORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>DTMFUNNELPORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>DTMTPIU FUNNELPORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>DWTBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>ETB2AXIBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>ETBBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>ETBFUNNELBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>ETFBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>ETMBASE</td>
<td>&lt;address&gt;</td>
</tr>
<tr>
<td>ETMETBFUNNELPORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>ETMFUNNEL2PORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>ETMFUNNELPORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>ETMTPIU FUNNELPORT</td>
<td>&lt;port&gt;</td>
</tr>
<tr>
<td>FILLDRZERO</td>
<td>[ON</td>
</tr>
</tbody>
</table>

<parameter>:
(Components cont.)

©1989-2019 Lauterbach GmbH
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;parameter&gt;:</strong></td>
<td>FUNNEL2BASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>FUNNELBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>HSMBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>HTMBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>HTMETBFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>HTMFUNNEL2PORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>HTMFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>HTMTPIUFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td>(Deprecated cont.)</td>
<td>ITMBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>ITMETBFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>ITMFUNNEL2PORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>ITMFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>ITMTPIUFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>PERBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>RAMBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>RTPBASE &lt;address&gt;</td>
</tr>
<tr>
<td>(Deprecated cont.)</td>
<td>SDTIBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>STMBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>STMETBFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>STMFUNNEL2PORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>STMFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>STMTPIUFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>TIADTFBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TIDRMBASE &lt;address&gt;</td>
</tr>
<tr>
<td>(Deprecated cont.)</td>
<td>TIEPMBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TIICEBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TIOCPBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TIOCP_TYPE &lt;type&gt;</td>
</tr>
<tr>
<td></td>
<td>TIPMIBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TISCBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TISTMBASE &lt;address&gt;</td>
</tr>
<tr>
<td>(Deprecated cont.)</td>
<td>TPIUBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TPIUFUNNELBASE &lt;address&gt;</td>
</tr>
<tr>
<td></td>
<td>TRACEETBFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>TRACEFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td></td>
<td>TRACETPIUFUNNELPORT &lt;port&gt;</td>
</tr>
<tr>
<td>view</td>
<td></td>
</tr>
</tbody>
</table>
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”.
<parameters> describing the “DebugPort”

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 GUIs) to simultaneously debug
different cores on the same target system.

Because of the default setting of this command

detector#1: <core>=1 <chip>=1
detector#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 will control the (same) trace
port. Sometimes it does not hurt if such a module will be controlled
twice. So even then it might work. But the correct specification
which might be a must is to tell the debugger that these cores
sharing resources are on the same <chip>. Whereby the “chip”
does not need to be identical with the device on your target board:

detector#1: <core>=1 <chip>=1
detector#2: <core>=2 <chip>=1
CORE <core> <chip>  
For cores on the same <chip> the debugger assumes they share the same resource if the control registers of the resource has the same address.

(cont.)

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:
**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 SYSstem.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 SYSstem.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 SYSstem.CONFIG NIDNTRSTPOLARITY High

This option must be used together with SYSstem.CONFIG NIDNTRSTTORST ON because you also have to use the TRST signal of an ARM debug cable as reset signal for NIDnT in this case.

**NIDNTRSTTORST**

[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.
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PortSHaRing</strong> [ON</td>
<td>OFF</td>
<td>Auto]</td>
</tr>
<tr>
<td><strong>Slave</strong> [ON</td>
<td>OFF]</td>
<td>If several debuggers share the same debug port, all except one must have this option active. <strong>JTAG</strong>: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting <strong>Slave ON</strong>.</td>
</tr>
<tr>
<td><strong>SWDP</strong> [ON</td>
<td>OFF]</td>
<td>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.</td>
</tr>
<tr>
<td><strong>SWDPIdleHigh</strong> [ON</td>
<td>OFF]</td>
<td>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</td>
</tr>
</tbody>
</table>
**SWDPTargetSel** `<value>`  
Device address in case of a multidrop serial wire debug port.  
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.
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.

**CHIPDRLENGTH** <bits>
Number of Data Register (DR) bits which needs to get a certain BYPASS pattern.

©1989-2019 Lauterbach GmbH
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHIPDRPATTERN</td>
<td>Data Register (DR) pattern which shall be used for BYPASS instead of the standard (1...1) pattern.</td>
</tr>
<tr>
<td>[Standard</td>
<td>Alternate</td>
</tr>
<tr>
<td>CHIPIRLENGTH</td>
<td>Number of Instruction Register (IR) bits which needs to get a certain BYPASS pattern.</td>
</tr>
<tr>
<td>&lt;bits&gt;</td>
<td></td>
</tr>
<tr>
<td>CHIPIRPATTERN</td>
<td>Instruction Register (IR) pattern which shall be used for BYPASS instead of the standard pattern.</td>
</tr>
<tr>
<td>[Standard</td>
<td>Alternate</td>
</tr>
<tr>
<td>Slave [ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>TAPState &lt;state&gt;</td>
<td>This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.</td>
</tr>
<tr>
<td></td>
<td>0 Exit2-DR</td>
</tr>
<tr>
<td></td>
<td>1 Exit1-DR</td>
</tr>
<tr>
<td></td>
<td>2 Shift-DR</td>
</tr>
<tr>
<td></td>
<td>3 Pause-DR</td>
</tr>
<tr>
<td></td>
<td>4 Select-IR-Scan</td>
</tr>
<tr>
<td></td>
<td>5 Update-DR</td>
</tr>
<tr>
<td></td>
<td>6 Capture-DR</td>
</tr>
<tr>
<td></td>
<td>7 Select-DR-Scan</td>
</tr>
<tr>
<td></td>
<td>8 Exit2-IR</td>
</tr>
<tr>
<td></td>
<td>9 Exit1-IR</td>
</tr>
<tr>
<td></td>
<td>10 Shift-IR</td>
</tr>
<tr>
<td></td>
<td>11 Pause-IR</td>
</tr>
<tr>
<td></td>
<td>12 Run-Test/Idle</td>
</tr>
<tr>
<td></td>
<td>13 Update-IR</td>
</tr>
<tr>
<td></td>
<td>14 Capture-IR</td>
</tr>
<tr>
<td></td>
<td>15 Test-Logic-Reset</td>
</tr>
<tr>
<td></td>
<td>Default: 7 = Select-DR-Scan.</td>
</tr>
</tbody>
</table>

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 SYSstem.Option DISSableSOFTRES [ON | OFF]
**TCKLevel `<level>`**  Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target.

Default: 0.

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

Please note:
- nTRST must have a pull-up resistor on the target.
- TCK can have a pull-up or pull-down resistor.
- Other trigger inputs need to be kept in inactive state.

Default: OFF.

---

**TAP types:**

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

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

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

- **ETB (Embedded Trace Buffer)** TAP if the ETB has its own TAP to access its control register (typical with ARM11 cores).
  - ETBDRPOST, ETBDRPRE, ETBIRPOST, ETBIRPRE.

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

- NEXTDRPOST, NEXTDRPRE, NEXTIRPOST, NEXTIRPRE.

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

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

```
SYStem.CONFIG IRPRE 15.
SYStem.CONFIG DRPRE 3.
SYStem.CONFIG DAPIRPOST 16.
SYStem.CONFIG DAPDRPOST 3.
SYStem.CONFIG ETBIRPOST 5.
SYStem.CONFIG ETBDRPOST 1.
SYStem.CONFIG ETBIRPRE 11.
SYStem.CONFIG ETBDRPRE 2.
```
A “Multitap” is a system level or chip level test access port (TAP) in a JTAG scan chain. It can for example provide functions to re-configure the JTAG chain or view and control power, clock, reset and security of different chip components.

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

Example:

```
CFGCONNECT <code>
The <code> is a hexadecimal number which defines the JTAG scan chain configuration. You need the chip documentation to figure out the suitable code. In most cases the chip specific default value can be used for the debug session.

Used if MULTITAP=STCLTAPx.

DAPTAP <tap>
Specifies the TAP number which needs to be activated to get the DAP TAP in the JTAG chain.

Used if MULTITAP=Icepickx.

DAP2TAP <tap>
Specifies the TAP number which needs to be activated to get a 2nd DAP TAP in the JTAG chain.

Used if MULTITAP=Icepickx.
```
DEBUGTAP <tap> Specifies the TAP number which needs to be activated to get the core TAP in the JTAG chain. E.g. ARM11 TAP if you intend to debug an ARM11.

Used if MULTITAP=Icepickx.

ETBTAP <tap> Specifies the TAP number which needs to be activated to get the ETB TAP in the JTAG chain.

Used if MULTITAP=Icepickx. ETB = Embedded Trace Buffer.

MULTITAP [NONE | IcepickA | IcepickB
| IcepickC | IcepickD | 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).
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 busses 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.
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.
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAP2AHBACCESSPORT &lt;port&gt;</td>
<td>DAP2 access port number (0-255) which shall be used for “AHB2:” access class. Default: &lt;port&gt;=0.</td>
</tr>
<tr>
<td>DAP2APBACCESSPORT &lt;port&gt;</td>
<td>DAP2 access port number (0-255) which shall be used for “APB2:” access class. Default: &lt;port&gt;=1.</td>
</tr>
<tr>
<td>DAP2AXIACCESSPORT &lt;port&gt;</td>
<td>DAP2 access port number (0-255) which shall be used for “AXI2:” access class. Default: port not available</td>
</tr>
<tr>
<td>DAP2DEBUGACCESSPORT &lt;port&gt;</td>
<td>DAP2 access port number (0-255) where the debug register can be found (typically on APB). Used for “DAP2:” access class. Default: &lt;port&gt;=1.</td>
</tr>
<tr>
<td>DAP2COREJTAGPORT &lt;port&gt;</td>
<td>JTAG-AP port number (0-7) connected to the core which shall be debugged. The JTAG-AP can be found on another DAP (DAP2).</td>
</tr>
<tr>
<td>DAP2JTAGPORT &lt;port&gt;</td>
<td>JTAG-AP port number (0-7) for an (other) DAP which is connected to a JTAG-AP.</td>
</tr>
<tr>
<td>DAP2MEMORYACCESSPORT &lt;port&gt;</td>
<td>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: &lt;port&gt;=0.</td>
</tr>
<tr>
<td>DEBUGACCESSPORT &lt;port&gt;</td>
<td>DAP access port number (0-255) where the debug register can be found (typically on APB). Used for “DAP:” access class. Default: &lt;port&gt;=1.</td>
</tr>
<tr>
<td>JTAGACCESSPORT &lt;port&gt;</td>
<td>DAP access port number (0-255) of the JTAG Access Port.</td>
</tr>
<tr>
<td>MEMORYACCESSPORT &lt;port&gt;</td>
<td>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: &lt;port&gt;=0.</td>
</tr>
<tr>
<td>AHBNAME &lt;name&gt;</td>
<td>AHB bus transactor name that shall be used for “AHB:” access class.</td>
</tr>
<tr>
<td>APBNAME &lt;name&gt;</td>
<td>APB bus transactor name that shall be used for “APB:” access class.</td>
</tr>
<tr>
<td>AXINAME &lt;name&gt;</td>
<td>AXI bus transactor name that shall be used for “AXI:” access class.</td>
</tr>
<tr>
<td>DAP2AHBNAME &lt;name&gt;</td>
<td>AHB bus transactor name that shall be used for “AHB2:” access class.</td>
</tr>
<tr>
<td><strong>DAP2APBNAME</strong> &lt;name&gt;</td>
<td>APB bus transactor name that shall be used for “APB2:” access class.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DAP2AXINAME</strong> &lt;name&gt;</td>
<td>AXI bus transactor name that shall be used for “AXI2:” access class.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DAP2DEBUGBUSNAME</strong> &lt;name&gt;</td>
<td>APB bus transactor name identifying the bus where the debug register can be found. Used for “DAP2:” access class.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DAP2MEMORYBUSNAME</strong> &lt;name&gt;</td>
<td>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”.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DEBUGBUSNAME</strong> &lt;name&gt;</td>
<td>APB bus transactor name identifying the bus where the debug register can be found. Used for “DAP:” access class.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>MEMORYBUSNAME</strong> &lt;name&gt;</td>
<td>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”.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DAPNAME</strong> &lt;name&gt;</td>
<td>DAP transactor name that shall be used for DAP access ports.</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>DAP2NAME</strong> &lt;name&gt;</td>
<td>DAP transactor name that shall be used for DAP access ports of 2nd order.</td>
</tr>
</tbody>
</table>
<parameters> describing debug and trace “Components”

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

Each configuration can be done by a command in a script file as well. Then you do not need to enter everything again on the next debug session. If you press the button with the three dots you get the corresponding command in the command line where you can view and maybe copy it into a script file.
You can have several of the following components: CMI, ETB, ETF, ETR, FUNNEL, STM.

**Example:** FUNNEL1, FUNNEL2, FUNNEL3,...

The `<address>` parameter can be just an address (e.g. 0x80001000) or you can 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.ATBSrc ETM.0 0 ETM.1 1
SYStem.CONFIG.FUNNEL2.ATBSrc FUNNEL1 0 STM1 7
SYStem.CONFIG.TPIU.ATBSrc FUNNEL2
```
Specifying 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 “... .ATBS<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.ATBS<source> ETM 0 HTM 1 STM 7

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

In an SMP (Symmetric MultiProcessing) debug session where you used a list of base addresses to specify one component per core you need to indicate which component in the list is meant:
Example: Four cores with ETM modules.
SYStem.CONFIG ETM.Base 0x1000 0x2000 0x3000 0x4000
SYStem.CONFIG FUNNEL1.ATBSource ETM.0 0 ETM.1 1
ETM.2 2 ETM.3 3
"...2" of "ETM.2" indicates it is the third ETM module which has
the base address 0x3000. The indices of a list are 0, 1, 2, 3,...
If the numbering is accelerating, starting from 0, without gaps,
like the example above then you can shorten it to
SYStem.CONFIG FUNNEL1.ATBSource ETM

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

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

....BASE <address>
This command informs the debugger 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, COREDEBUG, 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.

©1989-2019 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.
### ARMv8-A/-R Debugger

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETB.NoFlush [ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>ETB.Size &lt;size&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>ETM.StackMode [NotAvail-bale</td>
<td>TRGETM</td>
</tr>
<tr>
<td></td>
<td>NotAvailable: stack mode is not available for this on-chip trace. TRGETM: the trigger delay counter of the on-chip-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.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>FUNNEL.Name &lt;string&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>FUNNEL.PROGrammable [ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>HTM.Type [CoreSight</td>
<td>WPT]</td>
</tr>
<tr>
<td>L2CACHE.Type [NONE</td>
<td>Generic</td>
</tr>
<tr>
<td>OCP.Type &lt;type&gt;</td>
<td>Specifies the type of the OCP module. The &lt;type&gt; is just a number which you need to figure out in the chip documentation.</td>
</tr>
<tr>
<td>RTP.PerBase &lt;address&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Components and Available Commands

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

#### 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.

---

**ADTF.Base <address>**

ADTF.RESET

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.
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.
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.

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.

CoreSight Trace Funnel (CSTF) - ARM CoreSight module
Combines multiple trace sources onto a single trace bus (ATB = AMBA Trace Bus)

Hardware Security Module (HSM) - Infineon

AMBA AHB Trace Macrocell (HTM) - ARM CoreSight module
Trace source delivering trace data of access to an AHB bus.

Instrumentation Trace Macrocell (ITM) - ARM CoreSight module
Trace source delivering system trace information e.g. sent by software in printf() style.

Level 2 Cache Controller
The debugger might need to handle the controller to ensure cache coherency for debugger operation.

Open Core Protocol watchpoint unit (OCP) - Texas Instruments
Trace source module delivering bus trace information to a system trace module.
PMI.Base <address>
PMI.RESET
PMI.TraceID <id>
Power Management Instrumentation (PMI) - Texas Instruments
Trace source reporting power management events to a system trace module.

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

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

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

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

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

**Example**: SYStem.CONFIG ETMBASE APB:0x8011c000

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

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

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

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

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

For a list of possible components including a short description see Components and Available Commands.
... PORT <port>

Informs the debugger about which trace source is connected to which input port of which funnel. A CoreSight trace funnel provides 8 input ports (port 0-7) to combine the data of various trace sources to a common trace stream.

Example: SYStem.CONFIG STMFUNNEL2PORT 3

Meaning: The System Trace Module (STM) is connected to input port #3 on FUNNEL2.

On an SMP debug session some of these commands can have a list of <port> parameter.

In case there are dedicated funnels for the ETB and the TPIU their base addresses are specified by ETBFUNNELBASE, TPIUFUNNELBASE respectively. And the funnel port number for the ETM are declared by ETMETBFUNNELPORT, ETMTPIUFUNNELPORT respectively.

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.
**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>`

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMCBASE <code>&lt;address&gt;</code></td>
<td>Indicates the base address of the BMC.</td>
</tr>
<tr>
<td>BYPASS <code>&lt;seq&gt;</code></td>
<td>Specifies the source port for bypass.</td>
</tr>
<tr>
<td>COREBASE <code>&lt;address&gt;</code></td>
<td>Represents the base address of the core debug module.</td>
</tr>
<tr>
<td>CTIBASE <code>&lt;address&gt;</code></td>
<td>Specifies the base address of the CTI.</td>
</tr>
<tr>
<td>CTICONFIG <code>&lt;type&gt;</code></td>
<td>Defines the configuration type for the CTI.</td>
</tr>
<tr>
<td>DEBUGBASE <code>&lt;address&gt;</code></td>
<td>Indicates the base address of the debug module.</td>
</tr>
<tr>
<td>DTMCONFIG [ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>DTMETBFUNNELPORT <code>&lt;port&gt;</code></td>
<td>Specifies the DTM source port for funneling.</td>
</tr>
<tr>
<td>DTMFUNNEL2PORT <code>&lt;port&gt;</code></td>
<td>Indicates the DTM source port for funneling.</td>
</tr>
<tr>
<td>DTMFUNNELPORT <code>&lt;port&gt;</code></td>
<td>Specifies the DTM source port for funneling.</td>
</tr>
<tr>
<td>DTMTPIUFUNNELPORT <code>&lt;port&gt;</code></td>
<td>Defines the DTM source port for funneling.</td>
</tr>
<tr>
<td>DWTBASE <code>&lt;address&gt;</code></td>
<td>Represents the base address of the DWT.</td>
</tr>
<tr>
<td>ETB2AXIBASE <code>&lt;address&gt;</code></td>
<td>Indicates the base address of the ETB2AXI.</td>
</tr>
<tr>
<td>TIOCPTYPE <code>&lt;type&gt;</code></td>
<td>Specifies the type of the OCP module from Texas Instruments (TI).</td>
</tr>
</tbody>
</table>

`view` Opens a window showing most of the SYStem.CONFIG settings and allows to modify them.
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.
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 **COnponents** 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 /COnponents** 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](#).

<table>
<thead>
<tr>
<th><strong>Base</strong></th>
<th>Informs the debugger about the physical base address of the peripheral, access class <strong>AD</strong>:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>Selects the type or version of the ARM Generic Interrupt Controller (GIC) used in the SoC.</td>
</tr>
<tr>
<td></td>
<td>- <strong>GIC400</strong> and <strong>GIC500</strong> are implementations from ARM.</td>
</tr>
<tr>
<td></td>
<td>- <strong>GICv3</strong> and <strong>GICv4</strong> are controllers complying with the GICv3/GICv4 specification from ARM.</td>
</tr>
<tr>
<td></td>
<td>If the type is configured, the correct GIC register block is shown in the <strong>PER.view</strong> window.</td>
</tr>
<tr>
<td><strong>RESet</strong></td>
<td>Removes the component from the TRACE32 configuration.</td>
</tr>
<tr>
<td></td>
<td>This does not cause a physical reset for the peripheral on the chip.</td>
</tr>
</tbody>
</table>
### Overview of Mandatory and Optional TRACE32-internal Settings

<table>
<thead>
<tr>
<th>Then the TRACE32-internal settings with:</th>
<th>If the type* or version of the Generic Interrupt Controller (GIC) is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CONFIG GICD are</td>
<td>GIC400</td>
</tr>
<tr>
<td></td>
<td>mandatory</td>
</tr>
<tr>
<td>SYStem.CONFIG GICR are</td>
<td>N/A</td>
</tr>
<tr>
<td>SYStem.CONFIG GICC are</td>
<td>mandatory</td>
</tr>
<tr>
<td>SYStem.CONFIG GICH are</td>
<td>mandatory</td>
</tr>
<tr>
<td>SYStem.CONFIG GICV are</td>
<td>mandatory</td>
</tr>
<tr>
<td>(*) Type or version are set with SYStem.CONFIG GICD.Type.</td>
<td></td>
</tr>
</tbody>
</table>

#### 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
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
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.

<table>
<thead>
<tr>
<th><strong>&lt;sub_cmd&gt;</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Base &lt;address&gt;...</strong></td>
<td>Informs the debugger about the physical base address of the redistributor for each core, access class <strong>AD</strong>:</td>
</tr>
<tr>
<td><strong>RESet</strong></td>
<td>Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th><strong>Base &lt;address&gt;</strong></th>
<th>Informs the debugger about the physical base address of the CPU interface for each core, access class <strong>AD</strong>: If the address is banked in hardware, it is sufficient to specify only a single address.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESet</strong></td>
<td>Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.</td>
</tr>
</tbody>
</table>
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 GI400.
- The GICH configuration is optional if the SYStem.CONFIG GICD.Type is GIv3, GIv4, or GI500.

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.

<table>
<thead>
<tr>
<th><strong>Base</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESet</strong></td>
<td>Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.</td>
</tr>
</tbody>
</table>
SYStem.CONFIG GICV

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.

<table>
<thead>
<tr>
<th>Base</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESet</td>
<td>Removes the component from the TRACE32 configuration. This does not cause a physical reset for the peripheral on the chip.</td>
</tr>
</tbody>
</table>
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.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x&gt;</td>
<td>Serial number of the SMMU.</td>
</tr>
<tr>
<td>Base</td>
<td>Logical or physical base address of the memory-mapped SMMU register space.</td>
</tr>
<tr>
<td>Type</td>
<td>Defines the type of the ARM system MMU IP block: MMU400, MMU401, or MMU500.</td>
</tr>
<tr>
<td>Name</td>
<td>Assigns a user-defined name to an SMMU.</td>
</tr>
<tr>
<td>RESET</td>
<td>Resets the configuration of an SMMU specified with &lt;x&gt;.</td>
</tr>
</tbody>
</table>
SYStem.CPU

Select the used CPU

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.CPU &lt;cpu&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cpu&gt;:</td>
<td>CortexA32</td>
</tr>
</tbody>
</table>

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

Default selection:
- CORTEXA53 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.CpuAccess

Run-time memory access (intrusive)

SYStem.state window > CpuAccess

Format: SYStem.CpuAccess Enable | Denied | Nonstop

Default: Denied.

Configures how memory access is handled during runtime.

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

If SYStem.CpuAccess Enable 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 E: or the format option %E.

```
Data.dump E:0x100
Var.View %E first
```
Define the frequency of the debug port

**SYStem.JtagClock**

[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.

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.

<table>
<thead>
<tr>
<th>&lt;frequency&gt;</th>
<th>• The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays this value in the SYStem.state window.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• 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.</td>
</tr>
</tbody>
</table>

**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).
SYStem.LOCK

Tristate the JTAG port

Format: SYStem.LOCK [ON | OFF]

Default: OFF.

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

It must be ensured that the state of the ARM core JTAG state machine remains unchanged while the system is locked. To ensure correct hand-over, the options SYStem.CONFIG TAPState and SYStem.CONFIG TCKLevel must be set properly. They define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: nTRST must have a pull-up resistor on the target, EDBGRQ must have a pull-down resistor.
### Format:

<table>
<thead>
<tr>
<th>SYStem.MemAccess &lt;mode&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mode&gt;:</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>DAP</td>
</tr>
<tr>
<td>Denied</td>
</tr>
</tbody>
</table>

Default: Denied.

Allows to configure the memory access while the CPU is running.

**CPU**

A run-time memory access is made without CPU intervention while the program is running. This is only possible on the instruction set simulator.

**DAP**

A run-time memory access is done via the 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). Not possible on the instruction set simulator.

**Denied**

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

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.

To update 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
```
**SYStem.Mode**  
Establish the communication with the target

```
Format:  SYSystem.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 **SYSystem.Option RESetREGister**. Afterwards direct access to the CoreSight DAP interface is provided. For a reset, the reset line has to be connected to the debug connector.

  The debugger initializes the debug port (JTAG, SWD, cJTAG) and CoreSight DAP interface, but does not connect to the CPU. This debug mode is used if the CPU shall not be debugged or bypassed, i.e. the debugger can access the memory busses, such as AXI, AHB and APB, directly through the memory access ports of the CoreSight DAP.

  Typical use cases:
  - The debugger accesses (physical) memory and bypasses the CPU if a mapping exists. Memory might require initialization before it can be accessed.
  - The debugger accesses peripherals, e.g. for configuring registers prior to stopping the CPU in debug mode. Peripherals might need to be clocked and powered before they can be accessed.
  - Third-party software or proprietary debuggers use the TRACE32 API (application programming interface) to access the debug port and DAP via the TRACE32 debugger hardware.

- **Go**
  Resets the target via the reset line, initializes the debug port (JTAG, SWD, cJTAG), and starts the program execution. For a reset, the reset line has to be connected to the debug connector.
  Program execution can, for example, be stopped by the **Break** command.
**Attach**

No reset happens, the mode of the core (running or halted) does not change. The debug port (JTAG, SWD, cJTAG) will be initialized. After this command has been executed, the user program can, for example, be stopped with the **Break** command.

**StandBy**

Keeps the target in reset via the reset line and waits until power is detected. For a reset, the reset line has to be connected to the debug connector.

Once power has been detected, the debugger restores as many debug registers as possible (e.g. on-chip breakpoints, vector catch events, trace control) and releases the CPU from reset to start the program execution.

When a CPU power-down is detected, the debugger switches automatically back to the **StandBy** mode. This allows debugging of a power cycle because debug registers will be restored on power-up.

**NOTE:** Usually only on-chip breakpoints and vector catch events can be set while the CPU is running. To set a software breakpoint, the CPU has to be stopped.

**Up**

Resets the target via the reset line, initializes the debug port (JTAG, SWD, cJTAG), stops the CPU, and enters debug mode.

For a reset, the reset line has to be connected to the debug connector. The current state of all registers is read from the CPU.
The **SYStem.Option** commands are used to control special features of the debugger or emulator or to configure the target. It is recommended to execute the **SYStem.Option** commands *before* the emulation is activated by a **SYStem.Up** or **SYStem.Mode** command.

### SYStem.Option Address32

**Define address format display**

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

Default: 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.

### 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.
**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
  DeviceSYSystem
  NonCacheableNonShareable
  NonCacheableInnerShareable
  NonCacheableOuterShareable
  NonCacheableSYSystem
  WriteThroughNonShareable
  WriteThroughInnerShareable
  WriteBackOuterShareable
  WRITE THROUGH SYSTEM
```

Default: 0

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
- **DeviceSYSystem**: 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
- **NonCacheableSYSystem**: CSW.CACHE = 0x3, 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.

- WriteThroughNonShareable: CSW.CACHE = 0x6, CSW.DOMAIN = 0x0
- WriteThroughInnerShareable: CSW.CACHE = 0xA, CSW.DOMAIN = 0x1
- WriteBackOuterShareable: CSW.CACHE = 0xE, CSW.DOMAIN = 0x2
- WRITE THROUGH SYSTEM: CSW.CACHE = 0xE, CSW.DOMAIN = 0x3
SYStem.Option BreakOS

Allow break during OS-unlock

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.
**SYStem.Option CacheStatusCheck**  
Check status bits during cache access

| Format: | SYStem.Option.CacheStatusCheck [ON | OFF] |
|----------------|------------------------------------------|

Default: OFF

Enables status flags check during a cache access. The debugger checks if the CPU is ready to provide new cache data. This check should not be needed if the CPU runs fast enough. However, slow targets (like emulations systems) may need this status check.

This option is currently only applicable for Cortex-A57 and Cortex-A72 cores. The cache readout for all other cores will always behave as if this option was set to ‘ON’.

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

[SYStem.state window > CFLUSH]

|----------------|------------------------------------------------------------------|

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

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option CLTAPKEY &lt;key0&gt; &lt;key1&gt; &lt;key2&gt; &lt;key3&gt;</th>
</tr>
</thead>
</table>

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.

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 DAPDBGPWRUPREQ**  
Force debug power in DAP

| Format: | SYStem.Option DAPDBGPWRUPREQ [ON | AlwaysON] |
|---------|-----------------------------------------------|

Default: ON.

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

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

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

**OFF**
- 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 two TRACE32 PowerView GUIs, where one GUI is the master and all other GUIs are slaves. If the master GUI is closed first, it releases the debug power. As a result, a debug port fail error may be displayed in the remaining slave GUIs because they cannot access the debug interface anymore.

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

**Force debug power in DAP2**

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

Default: ON.

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

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

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

- **OFF**
  - Debug power is not requested and not checked by the debugger.
  - The control bit is set to 0.

**Use case:**

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

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

**SYStem.Option 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**

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option DAPREMAP (&lt;address_range&gt;\ &lt;address&gt;)</th>
</tr>
</thead>
</table>

The Debug Access Port (DAP) can be used for memory access during runtime. If the mapping on the DAP is different than the processor view, then this re-mapping command can be used.

| NOTE: | Up to 16 \(<address\_range>\ <address>\) pairs are possible. Each pair has to contain an address range followed by a single address. |

### SYStem.Option DAPSYSPWRUPREQ  
**Force system power in DAP**

| Format: | SYStem.Option DAPSYSPWRUPREQ \([AlwaysON | ON | OFF]\) |
|---|---|

Default: ON.

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

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

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

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

This option is for target processors having a Debug Access Port (DAP) e.g., Cortex-A or Cortex-R.
SYStem.Option DAP2SYSPWRUPREQ

Force system power in DAP2

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

Default: ON.

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

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

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

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

SYStem.Option 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).

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>: JTAGTOSWD.[auto | CORTEX | LUMINARY | None]
SWDDORMANT.[auto | WakeUp | NoWakeUp]

Default: auto.
This option is only required for expert users in case a non-standard SWD implementation is used.

**JTAGTOSWD** tells the debugger what to do when switching from JTAG to SWD mode:

<table>
<thead>
<tr>
<th>auto</th>
<th>Auto detection. The debugger tries to detect the correct SWD to JTAG shift sequence.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORTEX</td>
<td>Switching procedure like it should be done for Cortex cores.</td>
</tr>
<tr>
<td>LUMINARY</td>
<td>Switching procedure like it should be done for LuminaryMicro devices.</td>
</tr>
<tr>
<td>None</td>
<td>Do not send any SWD to JTAG sequence.</td>
</tr>
</tbody>
</table>

**SWDDORMANT** tells the debugger how to handle a dormant state when switching to SWD:

<table>
<thead>
<tr>
<th>auto</th>
<th>Auto detection.</th>
</tr>
</thead>
<tbody>
<tr>
<td>WakeUp</td>
<td>Wake up the debug port without prior checks by sending DORMANT to SWD sequence.</td>
</tr>
<tr>
<td>NoWakeUp</td>
<td>Do not wake up the debug port by suppressing the DORMANT to SWD sequence.</td>
</tr>
</tbody>
</table>

**SYStem.Option DIAG**  
*Activate more log messages*

Format:  

```plaintext
SYStem.Option DIAG [ON | OFF]
```

Default: OFF.

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

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

Format:  

```plaintext
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.Option.MemAccess** is disabled.
### SYStem.Option DisMode

Define disassembler mode

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option DisMode &lt;option&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;option&gt;:</td>
<td>AUTO</td>
</tr>
<tr>
<td></td>
<td>ACCESS</td>
</tr>
<tr>
<td></td>
<td>ARM</td>
</tr>
<tr>
<td></td>
<td>THUMB</td>
</tr>
<tr>
<td></td>
<td>AARCH64</td>
</tr>
</tbody>
</table>

Default: AUTO.

This command specifies the selected disassembler.

**AUTO**  
The information provided by the compiler output file is used for the disassembler selection. If no information is available it has the same behavior as the option ACCESS.

**ACCESS**  
The selected disassembler depends on the T 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:<addr>, List.auto R:<addr>** or **List.auto T:<addr>**, see also **Access Classes**.
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.
## SYStem.Option ICacheMaintenance

**I-Cache maintenance strategy**

| Format: | SYSystem.Option ICacheMaintenance [IALLU | IALLUIS | OFF] |
|---------|--------------------------------------------------|
|        | SYSystem.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 <addr>** or **Break.Set <addr> /SOFT, by <addr>**. 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.

## SYSystem.Option IMASKASM

**Disable interrupts while single stepping**

[Systerm.state window > IMASKASM]

| Format: | SYSystem.Option IMASKASM [ON | OFF] |

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.
**SYStem.Option IMASKHLL**  
**Disable interrupts while HLL single stepping**  

[SYStem.state window > IMASKHLL]

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

Default: OFF.

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

**SYStem.Option INTDIS**  
**Disable all interrupts**  

[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 MACHINESPACES

Address extension for guest OSes

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.
**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. |
SYStem.Option MMUSPACES

Separate address spaces by space IDs

Format:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.Option MMUSPACES [ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>SYStem.Option MMUspaces [ON</td>
<td>OFF] (deprecated)</td>
</tr>
<tr>
<td>SYStem.Option MMU [ON</td>
<td>OFF] (deprecated)</td>
</tr>
</tbody>
</table>

Default: OFF.

NOTE: SYStem.Option MMUSPACES should not be set to ON if only one translation table is used on the target.

If a debug session requires space IDs, you must observe the following sequence of steps:

1. Activate SYStem.Option MMUSPACES.
2. Load the symbols with Data.LOAD.

Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

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

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

Use alternative memory access

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.

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

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.

SYStem.Option NoPRCRRReset
Disable warm reset via PRCR

Default: OFF for most devices. ON for Cortex-A53 based devices.

The debugger tries to do a warm reset, (e.g. SYStem.Up). This is not done when this option is set to ON.
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.

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 both to the execution area (for active overlays) and to the storage area. In 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.

SYStem.Option OVERLAY ON
Data.List 0x2:0x11c4 ; Data.List <overlay_id>:<address>
**SYStem.Option PALLADIUM**

Extend debugger timeout

Format:

```plaintext
SYStem.Option PALLADIUM [ON | OFF] (deprecated)
Use SYStem.CONFIG DEBUGTIMESCALE instead.
```

Default: OFF.

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

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

---

**SYStem.Option PWRDWN**

Allow power-down mode

Format:

```plaintext
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:

```plaintext
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.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>The debugger will clear the CPSR.PAN bit temporarily in debug mode. Memory access are handled in the same fashion as for ARMv8.0</td>
</tr>
<tr>
<td>OFF</td>
<td>The debugger ignores the CPSR.PAN bit. All memory accesses are handled according to the PAN setting introduced with ARMv8.1</td>
</tr>
</tbody>
</table>
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.
SYStem.Option ResBreak

Halt the core after reset

Format: SYStem.Option ResBreak [ON | OFF]

Default: ON.

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

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

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option ResetDetection <code>&lt;method&gt;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;method&gt;</code>:</td>
<td>nSRST</td>
</tr>
</tbody>
</table>

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**

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.
**SYStem.Option RESetREGister**

Generic software reset

```plaintext
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.Prep 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.
**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.
### 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.
SYStem.state Display SYStem window

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).
SYStem.Option TRST  
Allow debugger to drive TRST

Format:  
SYStem.Option TRST [ON | OFF]

Default: ON.

If this option is disabled, the nTRST line is never driven by the debugger (permanent high). Instead five consecutive TCK pulses with TMS high are asserted to reset the TAP controller which have the same effect.

SYStem.Option WaitIDCODE  
IDCODE polling after deasserting reset

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

Default: OFF = disabled.

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

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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>1 second busy polling</td>
</tr>
<tr>
<td>OFF</td>
<td>Disabled</td>
</tr>
<tr>
<td>&lt;time&gt;</td>
<td>Configurable polling time, max. 30 sec, use 'us', 'ms', 's' as units.</td>
</tr>
</tbody>
</table>

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

[SYStem.state window > WaitReset]

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

Default: OFF = 3 msec.

Allows to add additional wait time after reset.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>1 sec delay</td>
</tr>
<tr>
<td>OFF</td>
<td>3 msec delay</td>
</tr>
<tr>
<td>&lt;time&gt;</td>
<td>Selectable time delay, min. 50 usec, max. 30 sec, use 'us', 'ms', 's' as units.</td>
</tr>
</tbody>
</table>

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 `SYSstem.Option WaitReset` is enabled (ON) and `SYSstem.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).

<table>
<thead>
<tr>
<th>nRESET (nSRST)</th>
<th>nTRST</th>
<th>CPU State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>reset</td>
</tr>
<tr>
<td></td>
<td>&gt;1 s (ON)</td>
<td>running</td>
</tr>
<tr>
<td></td>
<td></td>
<td>debug</td>
</tr>
</tbody>
</table>

©1989-2019 Lauterbach GmbH
SYStem.Option ZoneSPACES  Enable symbol management for ARM zones

Format:  

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

Default: OFF

The **SYStem.Option ZoneSPACES** command is relevant 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.

<table>
<thead>
<tr>
<th>OFF</th>
<th>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.</th>
</tr>
</thead>
</table>
| 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. |
Overview of Debugging with Zones

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:

<table>
<thead>
<tr>
<th></th>
<th>Non-secure mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Example: Linux user application</td>
</tr>
<tr>
<td>Z</td>
<td>Secure mode</td>
</tr>
<tr>
<td></td>
<td>Example: Secure crypto routine</td>
</tr>
<tr>
<td>H</td>
<td>Hypervisor mode</td>
</tr>
<tr>
<td></td>
<td>Example: XEN hypervisor</td>
</tr>
<tr>
<td>M</td>
<td>64-bit EL3/Monitor mode</td>
</tr>
<tr>
<td></td>
<td>Example: Trusted boot stage / monitor</td>
</tr>
</tbody>
</table>

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

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

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

SYStem.Option ZoneSPACES ON

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

; 2. Load the sysmon symbols for secure mode (access classes Z:, ZP: and ZD: are used for the symbols) but without offset:
Data.LOAD.Elf sysmon Z:0x0 /NoCODE

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

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

Example: Symbolic Memory Access

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

Data.dump swapper_pg_dir

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

Example: Deleting Zone-specific Symbols

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

sYmbol.Delete N:0x0--0xffffffff ; non-secure mode (access classes N:);
Example: Zone-specific Debugger Address Translation Setup

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

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

```
SYStem.Option ZoneSPACES ON

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

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

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

; enable address translation and table walk
TRANSlation.ON

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

Often, an operation system (such as a Linux kernel) runs in the context of the guest system. In such a setup with hypervisor and guest OS, it is possible to load both the hypervisor symbols to H: and all OS-related symbols to N:

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

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

   TASK.ACCESS N:

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

**NOTE:**

This debugger setup, which is based on the option ZoneSPACES, allows work with only one guest system simultaneously. If the hypervisor has configured more than one guest, only the guest that is active in the non-secure CPU mode is visible. To work with another guest, the system must continue running until an inactive guest becomes the active guest.

With SYStem.Option.MACHINESPACES enabled, TRACE32 also supports concurrent debugging of a virtualized system with hypervisor and multiple guests.

the CPU specific zones N: Z: H: and M: will be extended by machine specific zones. Each of these zones is identified by a machine ID. Each guest has its own zone because it uses a separate translation table and a separate register set.
Example: Setup for a Guest OS and a Hypervisor

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

```plaintext
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++0xFFFFFFFF to the intermediate physical address range
; starting at I:0x40000000
MMU.FORMAT linux swapper_pg_dir N:0xC0000000++0xFFFFFFFF 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 <taskname>, will automatically refer to tasks running in the same zone as the OS Awareness.
**SYStem.Option ZYNQJTAGINDEPENDENT**  Configure JTAG cascading

| 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. |
ARM specific Benchmarking Commands

The **BMC** (BenchMark Counter) 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.

Counts:

```
PMN<x>
```

**BMC.<counter>.EVENT**  

Exception level (EL)
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:

| 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 | Count 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>. |

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

| NSEL2 | Count non-secure events of EL2. ArmV8.4 only. |
| SEL2 | Count secure events of EL2. ArmV8.4 only. |

Example:

```
BMC.PMN0.EVENT INST ; Count instructions ...  
BMC.PMN0.CountEL13 SEL3 ; ... in secure EL3 only with PMN0
```
Benchmark counter values can be returned with the function `BMC.COUNTER()`.

## 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:

### Special events

- **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 exception 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.
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.

```plaintext
TrOnchip.ContextID ON
Break.Set VectorSwi /Program /Onchip /TASK EKern.exe:Thread1
```

### TrOnchip.CONVert

**Allow extension of address range of breakpoint**

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 <addr_range>` in the figure below).

The debug logic of a processor may be implemented in one of the following three ways:
1. The debug logic does not allow to set range breakpoints, but only single address breakpoints. Consequently the debugger cannot set range breakpoints and returns an error message.

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

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

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

<table>
<thead>
<tr>
<th>ON (default)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>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).</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OFF</th>
<th></th>
</tr>
</thead>
</table>
| **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.MatchASID**  
Extend on-chip breakpoint(trace filter) by ASID

| Format: | TrOnchip.MatchASID [ON | OFF] |
|---------|--------------------------|
|         | TrOnchip.ASID [ON | OFF] (deprecated) |

<table>
<thead>
<tr>
<th>OFF (default)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stop the program execution at on-chip breakpoint if the address matches. Trace filters and triggers become active if the address matches.</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ON</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>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.</strong></td>
<td></td>
</tr>
</tbody>
</table>
TrOnchip.MatchZone: Extend on-chip breakpoint/trace filter by zone

| Format: TrOnchip.MatchZone [ON | OFF] |
|--------------------------------------|
| **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).

---

```plaintext
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
```

©1989-2019 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

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrOnchip.RESet</th>
</tr>
</thead>
</table>

Sets the TrOnchip settings and trigger module to the default settings.
TrOnchip.Set
Set bits in the vector catch register

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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>The core stops on a reset event.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>OSUnloCk</td>
<td>The core stops on a OS unlock event.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>TDA</td>
<td>The core is halted when it accesses debug registers and when the OS-</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>NSEL1</td>
<td>Non-secure exception level 1.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>NSEL2</td>
<td>Non-secure exception level 2.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>SEL1</td>
<td>Secure exception level 1.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>SEL3</td>
<td>Secure exception level 3.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>StepVector</td>
<td>Step into exception handler if ON. Step over exception handler if OFF.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSEL1E</td>
<td>Non-secure exception level 1 entry.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>NSEL1R</td>
<td>Non-secure exception level 1 return.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
<tr>
<td>NSEL2E</td>
<td>Non-secure exception level 2 entry.</td>
</tr>
<tr>
<td>[ON</td>
<td>OFF]</td>
</tr>
</tbody>
</table>
### Additional options for ARMv8.4-A cores (or later):

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSEL2R</td>
<td>Non-secure exception level 2 return. The core stops on an exception return to non-secure EL2, e.g. from a higher exception level.</td>
</tr>
<tr>
<td>SEL1E</td>
<td>Secure exception level 1 entry. The core stops on an exception entry to secure EL1, e.g. from a lower exception level.</td>
</tr>
<tr>
<td>SEL1R</td>
<td>Secure exception level 1 return. The core stops on an exception return to secure EL1, e.g. from a higher exception level.</td>
</tr>
<tr>
<td>SEL3E</td>
<td>Secure exception level 3 entry. The core stops on an exception entry to secure EL3, e.g. from a lower exception level.</td>
</tr>
<tr>
<td>SEL3R</td>
<td>Secure exception level 3 return. The core stops on an exception return to secure EL3, e.g. from a higher exception level.</td>
</tr>
<tr>
<td>SEL2</td>
<td>Secure exception level 2. The core stops when the exception level changes to secure EL2.</td>
</tr>
<tr>
<td>SEL2E</td>
<td>Secure exception level 2 entry. The core stops on an exception entry to secure EL2, e.g. from a lower exception level.</td>
</tr>
<tr>
<td>SEL2R</td>
<td>Secure exception level 2 return. The core stops on an exception return to secure EL2, e.g. from a higher exception level.</td>
</tr>
</tbody>
</table>
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 is only possible for ARMv8.0 or ARMv8.1 based devices. In this case, the debugger will issue a warning that the CPU cannot resume.

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.

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.
Convert breakpoints on scalar variables

Format: \texttt{TrOnchip.VarCONVert [ON | OFF]}

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

| Format: | TrOnchip.state |

Opens the TrOnchip.state window.

---

**TrOnchip.state**

Display on-chip trigger window

[Go to figure]

| 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.
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:

<table>
<thead>
<tr>
<th>CACHE.command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CACHE.view</td>
<td>Display control registers for configuration of all cache levels.</td>
</tr>
<tr>
<td>CACHE.INVALIDATE</td>
<td>Invalidate all entries stored in L1 or L2 instruction/data cache.</td>
</tr>
<tr>
<td>CACHE.FLUSH</td>
<td>Clean and invalidate all entries stored in L1 or L2 instruction/data cache.</td>
</tr>
<tr>
<td>CACHE.DUMP</td>
<td>Display all entries located in L1 or L2 instruction/data cache.</td>
</tr>
</tbody>
</table>

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:

For an overview of the TRACE32 cache support for various ARMv8 cores, please refer to “TRACE32 Cache Support by CPU Type”, page 176.

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

<table>
<thead>
<tr>
<th>ARMv8 Core</th>
<th>L1 IC</th>
<th>L1 DC</th>
<th>L2</th>
<th>L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-A32</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A35</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A53</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A55</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A57</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A72</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A73</td>
<td>Cache analysis only when EL3 is in AArch64 state. No restrictions on cache maintenance.</td>
<td>Cache analysis only when EL3 is in AArch64 state. No restrictions on cache maintenance.</td>
<td>Cache maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A75</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache maintenance</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A76</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>Cache analysis and maintenance</td>
<td>-</td>
</tr>
</tbody>
</table>

**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`
### CPU specific MMU Commands

#### MMU.DUMP

Page wise display of MMU translation table

| Format: MMU.DUMP <table> [<range> | <addr> | <range> <root> | <addr> <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. |
| PageTable | Displays the current MMU translation table entries of the CPU. This command reads all tables the CPU currently uses for MMU translation and displays the table entries. |
| KernelPageTable | Displays the MMU translation table of the kernel. If specified with the MMU FORMAT command, this command reads the MMU translation table of the kernel and displays its table entries. |
| TaskPageTable <task_magic> | <task_id> | <task_name> | <space_id>:0x0 | Displays the MMU translation table entries of the given process. Specify one of the TaskPageTable arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. |
| For information about the first three parameters, see “What to know about Task Magic Numbers, Task IDs and Task Names” (general_ref_t.pdf). |
| See also the appropriate OS Awareness Manuals. |

©1989-2019 Lauterbach GmbH
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 physical 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>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITLB</td>
<td>Displays the contents of the Instruction Translation Lookaside Buffer.</td>
</tr>
<tr>
<td>DTLB</td>
<td>Displays the contents of the Data Translation Lookaside Buffer.</td>
</tr>
<tr>
<td>TLB0</td>
<td>Displays the contents of the Translation Lookaside Buffer 0.</td>
</tr>
<tr>
<td>TLB1</td>
<td>Displays the contents of the Translation Lookaside Buffer 1.</td>
</tr>
<tr>
<td>NonSecPageTable</td>
<td>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.</td>
</tr>
<tr>
<td>SecPageTable</td>
<td>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.</td>
</tr>
<tr>
<td>HypPageTable</td>
<td>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.</td>
</tr>
<tr>
<td>IntermedPageTable</td>
<td>Displays the translation table used by the MMU for the second stage translation of a guest machine. (i.e., intermediate physical address to physical address). This is the table pointed to by MMU register VTTBR. This table is only available in CPUs with Virtualization Extension.</td>
</tr>
<tr>
<td><strong>EL1PageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>EL2PageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>EL3PageTable</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

Next:
- Screenshot of the MMU.DUMP.PageTable Window
- Description of the Columns
- Examples
## Description of Columns in the MMU.DUMP.PageTable Window

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>Logical page address range</td>
</tr>
<tr>
<td>physical</td>
<td>Physical page address range</td>
</tr>
<tr>
<td>sec</td>
<td>Security state of entry (s=secure, ns=non-secure, sns=non-secure entry in secure page table)</td>
</tr>
<tr>
<td>d</td>
<td>Domain</td>
</tr>
<tr>
<td>size</td>
<td>Size of mapped page in bytes</td>
</tr>
</tbody>
</table>
| 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) |
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)

<table>
<thead>
<tr>
<th>Column “permissions” &lt;data_access_permission&gt; Keywords</th>
<th>Description of the &lt;data_access_permission&gt; Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>noaccess</td>
<td>No read access and no write access for this privilege level</td>
</tr>
<tr>
<td>readonly</td>
<td>Read access, no write access for this privilege level</td>
</tr>
<tr>
<td>readwrite</td>
<td>Read and write access for this privilege level</td>
</tr>
<tr>
<td>noaccess (DACR) (AArch32 only)</td>
<td>No read access and no write access for this privilege level as specified by register DACR bits associated with the domain.</td>
</tr>
<tr>
<td>readwrite (DACR) (AArch32 only)</td>
<td>Read and write access for this privilege level (Managers) as specified by register DACR bits associated with the domain.</td>
</tr>
<tr>
<td>reserved (DACR) (AArch32 only)</td>
<td>Reserved setting (unpredictable behavior) as specified by register DACR bits associated with the domain.</td>
</tr>
<tr>
<td>reserved</td>
<td>Reserved setting (unpredictable behavior)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Column “permissions”</th>
<th>Description of the &lt;data_access_permission&gt; Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;data_access_permission&gt; Keywords</td>
<td></td>
</tr>
<tr>
<td><strong>no access</strong> (AArch32 only)</td>
<td>No read access and no write access for non-secure PL1 and PL0</td>
</tr>
<tr>
<td><strong>read access only</strong> (AArch32 only)</td>
<td>Read access and no write access for non-secure PL1 and PL0</td>
</tr>
<tr>
<td><strong>write access only</strong> (AArch32 only)</td>
<td>Write access and no read access for non-secure PL1 and PL0</td>
</tr>
<tr>
<td><strong>read/write access</strong> (AAArch32 only)</td>
<td>Read and write access for non-secure PL1 and PL0</td>
</tr>
<tr>
<td><strong>EL0,1: no access</strong> (AArch64 only)</td>
<td>No read access and no write access for non-secure EL1 and EL0</td>
</tr>
<tr>
<td><strong>EL0,1: read only</strong> (AArch64 only)</td>
<td>Read access and no write access for non-secure EL1 and EL0</td>
</tr>
<tr>
<td><strong>EL0,1: write only</strong> (AArch64 only)</td>
<td>Write access and no read access for non-secure EL1 and EL0</td>
</tr>
<tr>
<td><strong>EL0,1: read/write</strong> (AArch64 only)</td>
<td>Read and write access for non-secure EL1 and EL0</td>
</tr>
</tbody>
</table>
Instruction Execution Permissions for non-Stage2 Page Tables
(AArch64 EL0/1/2/3 and AArch32 PL0/1/2)

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)

<table>
<thead>
<tr>
<th>Column “permissions” &lt;execution_permission&gt; Keywords</th>
<th>Description of the &lt;execution_permission&gt; Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex or exec</td>
<td>Code execution allowed</td>
</tr>
<tr>
<td>xn or notexec</td>
<td>Code execution not allowed</td>
</tr>
<tr>
<td>xnW</td>
<td>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</td>
</tr>
<tr>
<td>xnD</td>
<td>Code execution not allowed because dirty bit modifier (DBM) is set in page table entry</td>
</tr>
<tr>
<td>nopexec</td>
<td>Code execution not allowed for privileged level, code execution allowed for unprivileged level</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Column “permissions”</th>
<th>Description of the &lt;execution_permission&gt; Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;execution_permission&gt;</td>
<td>KEYS</td>
</tr>
<tr>
<td>EL1:xn EL0:xn</td>
<td>No code execution allowed for EL1 and EL0</td>
</tr>
<tr>
<td>EL1:xn EL0:ex</td>
<td>No code execution allowed for EL1, code execution allowed for EL0</td>
</tr>
<tr>
<td>EL1:ex EL0:xn</td>
<td>Code execution allowed for EL1, no code execution allowed for EL0</td>
</tr>
<tr>
<td>EL1:ex EL0:ex</td>
<td>Code execution allowed for EL1 and EL0</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Column “pageflags”</th>
<th>Description of the memory attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>device nGnRnE</td>
<td>Device memory type non-Gathering, non-Reordering, no Early write acknowledgement</td>
</tr>
<tr>
<td>device nGnRE</td>
<td>Device memory type non-Gathering, non-Reordering, Early Write Acknowledgement</td>
</tr>
<tr>
<td>device nGRE</td>
<td>Device memory type non-Gathering, Reordering, Early Write Acknowledgement</td>
</tr>
<tr>
<td>device GRE</td>
<td>Device memory type Gathering, Reordering, Early Write Acknowledgement</td>
</tr>
<tr>
<td>write-thru/write-allocate/trans</td>
<td>Inner and Outer Normal Memory, Write-through transient, write allocate policy</td>
</tr>
<tr>
<td>write-thru/read-allocate/trans</td>
<td>Inner and Outer Normal Memory, Write-through transient, read allocate policy</td>
</tr>
<tr>
<td>write-thru/r-w-allocate/trans</td>
<td>Inner and Outer Normal Memory, Write-through transient, read and write allocate policy</td>
</tr>
<tr>
<td>non-cachable</td>
<td>Inner and Outer Normal Memory, Non-Cacheable</td>
</tr>
<tr>
<td>write-back/write-allocate/trans</td>
<td>Inner and Outer Normal Memory, Write-back transient, write allocate policy</td>
</tr>
<tr>
<td>write-back/read-allocate/trans</td>
<td>Inner and Outer Normal Memory, Write-back transient, read allocate policy</td>
</tr>
<tr>
<td>write-back/r-w-allocate/trans</td>
<td>Inner and Outer Normal Memory, Write-back transient, read and write allocate policy</td>
</tr>
<tr>
<td>write-thru/write-allocate</td>
<td>Inner and Outer Normal Memory, Write-through non-transient, write allocate policy</td>
</tr>
<tr>
<td>write-thru/read-allocate</td>
<td>Inner and Outer Normal Memory, Write-through non-transient, read allocate policy</td>
</tr>
<tr>
<td>write-thru/read-write-allocate</td>
<td>Inner and Outer Normal Memory, Write-through non-transient, read and write allocate policy</td>
</tr>
<tr>
<td>write-back/write-allocate</td>
<td>Inner and Outer Normal Memory, Write-back non-transient, write allocate policy</td>
</tr>
<tr>
<td>Column “pageflags”</td>
<td>Description of the memory attributes</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>write-back/read-allocate</td>
<td>Inner and Outer Normal Memory, Write-back non-transient, read allocate policy</td>
</tr>
<tr>
<td>write-back/read-write-allocate</td>
<td>Inner and Outer Normal Memory, Write-back non-transient, read and write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-thru/wa/t</td>
<td>Inner / Outer Normal Memory, Write-through transient, write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-thru/ra/t</td>
<td>Inner / Outer Normal Memory, Write-through transient, read allocate policy</td>
</tr>
<tr>
<td>I:/O: w-thru/rwa/t</td>
<td>Inner / Outer Normal Memory, Write-through transient, read and write allocate policy</td>
</tr>
<tr>
<td>I:/O: non-cachable</td>
<td>Inner / Outer Normal Memory, Non-Cacheable</td>
</tr>
<tr>
<td>I:/O: w-back/wa/t</td>
<td>Inner / Outer Normal Memory, Write-back transient, write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-back/ra/t</td>
<td>Inner / Outer Normal Memory, Write-back transient, read allocate policy</td>
</tr>
<tr>
<td>I:/O: w-back/rwa/t</td>
<td>Inner / Outer Normal Memory, Write-back transient, read and write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-thru/wa</td>
<td>Inner / Outer Normal Memory, Write-through non-transient, write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-thru/ra</td>
<td>Inner / Outer Normal Memory, Write-through non-transient, read allocate policy</td>
</tr>
<tr>
<td>I:/O: w-thru/rwa</td>
<td>Inner / Outer Normal Memory, Write-through non-transient, read and write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-back/wa</td>
<td>Inner / Outer Normal Memory, Write-back non-transient, write allocate policy</td>
</tr>
<tr>
<td>I:/O: w-back/ra</td>
<td>Inner / Outer Normal Memory, Write-back non-transient, read allocate policy</td>
</tr>
<tr>
<td>I:/O: w-back/rwa</td>
<td>Inner / Outer Normal Memory, Write-back non-transient, read and write allocate policy</td>
</tr>
</tbody>
</table>

**P:** `<permission>` | **U:** `<permission>` |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Code execution allowed for privileged and unprivileged level.</td>
<td>Data access permissions for privileged and unprivileged execution level. <code>&lt;permission&gt;</code> may be readwrite, readonly or noaccess</td>
</tr>
<tr>
<td>exec</td>
<td>Code execution allowed for privileged and unprivileged level</td>
</tr>
<tr>
<td>notexec</td>
<td>Code execution allowed for privileged level but not for unprivileged level</td>
</tr>
<tr>
<td>nopexec</td>
<td>Code execution not allowed for privileged and unprivileged level</td>
</tr>
</tbody>
</table>
Examples for Page Tables in Virtualized Systems

Example 1:

```
SYStem.Option MACHINESPACES ON

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

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

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

Example 2:

```
SYStem.Option MACHINESPACES ON

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

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

Example 3:

```
SYStem.Option MACHINESPACES ON

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

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

;b) With the option /Fulltranslation the intermediate physical 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
```
MMU.List

Compact display of MMU translation table

Format:

```
MMU.List <table> [<range> | <addr> | <range> <root> | <addr> <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-specific 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.

**PageTable**

Lists the current MMU translation of the CPU. This command reads all tables the CPU currently uses for MMU translation and lists the address translation.
<table>
<thead>
<tr>
<th><strong>KernelPageTable</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TaskPageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td>&lt;task_magic&gt;</td>
<td>For information about the first three parameters, see “What to know about Task Magic Numbers, Task IDs and Task Names” (general_ref_t.pdf).</td>
</tr>
<tr>
<td>&lt;task_id&gt;</td>
<td>See also the appropriate OS Awareness Manuals.</td>
</tr>
<tr>
<td>&lt;task_name&gt;</td>
<td>&lt;space_id&gt;:0x0</td>
</tr>
<tr>
<td>&lt;option&gt;</td>
<td>For description of the options, see MMU.DUMP.</td>
</tr>
</tbody>
</table>

**CPU specific Tables in MMU.List <table>**

<table>
<thead>
<tr>
<th><strong>TLB</strong></th>
<th>Displays the contents of the Translation Lookaside Buffer.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NonSecPageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>SecPageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>HypPageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>IntermedPageTable</strong></td>
<td>Displays the translation table used by the MMU for the second stage translation of a guest machine. (i.e., intermediate physical address to physical address). This is the table pointed to by MMU register VTTBR. This table is only available in CPUs with Virtualization Extension.</td>
</tr>
<tr>
<td><strong>EL1PageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>EL2PageTable</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>EL3PageTable</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

---

### 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 CPU to the debugger-internal static translation table.

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

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

| **PageTable** | Loads the current MMU address translation of the CPU. This command reads all tables the CPU currently uses for MMU translation, and copies the address translation into the debugger-internal static translation table. |
| **KernelPageTable** | Loads the MMU translation table of the kernel. If specified with the **MMU FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table. |
## TaskPageTable

<table>
<thead>
<tr>
<th>&lt;task_magic&gt;</th>
<th>&lt;task_id&gt;</th>
<th>&lt;task_name&gt;</th>
<th>&lt;space_id&gt;:0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>For information about the first three parameters, see “What to know about Task Magic Numbers, Task IDs and Task Names” (general_ref_t.pdf).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>See also the appropriate OS Awareness Manual.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ALL

| 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

| OEMAddressTable |
| Loads the OEM Address Table from the CPU to the debugger-internal translation table. |
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:

<table>
<thead>
<tr>
<th>ARMv8 Core</th>
<th>ITLB</th>
<th>DTLB</th>
<th>TLB0</th>
<th>TLB1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-A32</td>
<td>Reads main TLB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A35</td>
<td>Reads main TLB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A53</td>
<td>Reads main TLB</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A55</td>
<td>-</td>
<td>-</td>
<td>Reads L2 TLB</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A57</td>
<td>Reads L1 instruction TLB</td>
<td>Reads L1 data TLB</td>
<td>Reads L2 TLB</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A72</td>
<td>Reads L1 instruction TLB</td>
<td>Reads L1 data TLB</td>
<td>Reads L2 TLB</td>
<td>-</td>
</tr>
<tr>
<td>Cortex-A73</td>
<td>-</td>
<td>-</td>
<td>Reads L2 TLB RAM0</td>
<td>Reads L2 TLB Ram1</td>
</tr>
<tr>
<td>Cortex-A75</td>
<td>-</td>
<td>-</td>
<td>Reads L2 TLB RAM0</td>
<td>Reads L2 TLB Ram1</td>
</tr>
<tr>
<td>Cortex-A76</td>
<td>Reads L1 instruction TLB</td>
<td>Reads L1 data TLB</td>
<td>Reads L2 TLB</td>
<td>-</td>
</tr>
</tbody>
</table>
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.

```plaintext
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**
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.

**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.
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.
Arguments in SMMU Commands

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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;name&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>&lt;smrg_index&gt;</td>
<td>Index of a stream map register group, e.g. 0x04. The indices are listed in the index column of the SMMU.StreamMapTable.</td>
</tr>
<tr>
<td>&lt;cbndx&gt;</td>
<td>Index of a translation context bank.</td>
</tr>
<tr>
<td>&lt;address&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>IntermediatePT</td>
<td>Used to switch between stage 1 and stage 2 page table or register view:</td>
</tr>
<tr>
<td></td>
<td>• Omit this option to view the translation table entries or registers of stage 1.</td>
</tr>
<tr>
<td></td>
<td>• Include this option to view the translation table entries or registers of stage 2.</td>
</tr>
</tbody>
</table>
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:**

<table>
<thead>
<tr>
<th><code>&lt;base_address&gt;</code></th>
<th>Logical or physical base address of the memory-mapped SMMU register space.</th>
</tr>
</thead>
</table>

**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.
<table>
<thead>
<tr>
<th><strong>&lt;name&gt;</strong></th>
<th>User-defined name of an SMMU. The name must be unique and can be max. 9 characters long.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;smmu_type&gt;</strong></td>
<td>Defines the type of the ARM system MMU IP block: MMU400, MMU401, or MMU500.</td>
</tr>
</tbody>
</table>

**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.

**Example:**

```plaintext
;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:**

```plaintext
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:**

<table>
<thead>
<tr>
<th><strong>&lt;name&gt;</strong></th>
<th>For a description of &lt;name&gt;, click here.</th>
</tr>
</thead>
</table>

**Example:**

```plaintext
SMMU.Clear myGPU ;deletes the SMMU named myGPU
```
Using the **SMMU.Register** command group, you can view and modify the peripheral registers of an SMMU. The command group provides the following commands:

<table>
<thead>
<tr>
<th>Command Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMMU.Register.ContextBank</td>
<td>Display the registers of a context bank</td>
</tr>
<tr>
<td>SMMU.Register.Global</td>
<td>Display the global registers of an SMMU</td>
</tr>
<tr>
<td>SMMU.Register.StreamMapRegGrp</td>
<td>Display the registers of an SMRG</td>
</tr>
</tbody>
</table>

**Example:**

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

;highlight changes in orange in any SMMU.Register.* window
SETUP.Var %SpotLight.on
```
Format: SMMU.Register.ContextBank <name> <cbndx>

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

**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
SMMU/Register/Global

Display global registers of SMMU

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:

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>For a description of &lt;name&gt;, click here.</th>
</tr>
</thead>
</table>

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.
SMMU.Register.StreamMapRegGrp

Display registers of an SMRG

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:

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>For a description of &lt;name&gt;, etc., click here.</th>
</tr>
</thead>
</table>

Example:

SMMU.StreamMapRegGrp.Register myGPU 0x0D /IntermediatePT
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**.
SMMU.SSDtable

Display security state determination table

Format:  
\[ \text{SMMU.SSDtable } \langle \text{name} \rangle [\langle \text{start\_index} \rangle ] \]

Displays the security state determination table (SSD table) as a bit field consisting of \text{s} (secure) or \text{ns} (non-secure) entries. If the SMMU has no SSD table defined, you receive an error message in the \text{AREA} window.

A  In the SSD table, the black arrow indicates the \langle \text{start\_index} \rangle, 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 (\text{s}) or non-secure (\text{ns}) memory domain.

You may find the SSD table easier to interpret by reducing the width of the \text{SMMU.SSDtable} window. Example for the raw data 0x68 in the SSD table:

C  In the \text{Data.dump} window, the black arrow indicates the dumped raw data from the SSD table.

D  The 1st white column (00 to 07) relates to the 1st raw data column. The 2nd white column (08 to 0F) relates to the 2nd raw data column, etc.
Arguments:

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>For a description of &lt;name&gt;, click here.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;start_index&gt;</td>
<td>Starts the display of the SSD table at the specified SSD index. See SSD index column in the SMMU.SSDtable window.</td>
</tr>
</tbody>
</table>

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).
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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SMMU.StreamMapRegGrp.ContextReg</strong></td>
<td>Shows the registers of the context bank associated with the stage 1 and/or stage 2 translation.</td>
</tr>
<tr>
<td><strong>SMMU.StreamMapRegGrp.Dump</strong></td>
<td>Dumps the page table associated with the stage 1 and/or stage 2 translation page wise.</td>
</tr>
<tr>
<td><strong>SMMU.StreamMapRegGrp.list</strong></td>
<td>Lists the page table entries associated with the stage 1 and/or stage 2 translation in a compact format.</td>
</tr>
</tbody>
</table>
SMMU.StreamMapRegGrp.ContextReg

Display context bank registers

<table>
<thead>
<tr>
<th>Format:</th>
<th>SMMU.StreamMapRegGrp.ContextReg  &lt;args&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;args&gt;:</td>
<td>&lt;name&gt; &lt;smrg_index&gt; [/IntermediatePT]</td>
</tr>
</tbody>
</table>

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. |
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.
**SMMU.StreamMapRegGrp.Dump**

**Page-wise display of SMMU page table**

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”.

![SMMU.StreamMapRegGrp.Dump Window]

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:**

<table>
<thead>
<tr>
<th><code>&lt;name&gt;</code></th>
<th>For a description of <code>&lt;name&gt;</code>, etc., click here.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IntermediatePT</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>

**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
### Description of Columns

This table describes the columns of the following windows:

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

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>Logical page address range</td>
</tr>
<tr>
<td>physical</td>
<td>Physical page address range</td>
</tr>
<tr>
<td>sec</td>
<td>Security state of entry (s=secure, ns=non-secure, sns=non-secure entry in secure page table)</td>
</tr>
<tr>
<td>d</td>
<td>Domain</td>
</tr>
<tr>
<td>size</td>
<td>Size of mapped page in bytes</td>
</tr>
<tr>
<td>permissions</td>
<td>Access permissions (P=privileged, U=unprivileged, exec=execution allowed)</td>
</tr>
<tr>
<td>glb</td>
<td>Global page</td>
</tr>
<tr>
<td>shr</td>
<td>Shareability (no=non-shareable, yes=shareable, inn=inner shareable, out=outer shareable)</td>
</tr>
<tr>
<td>pageflags</td>
<td>Memory attributes (see <a href="#">Description of the memory attributes</a>)</td>
</tr>
<tr>
<td>tablewalk</td>
<td>Only for <strong>SMMU.StreamMapRegGrp.Dump</strong>: 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)</td>
</tr>
</tbody>
</table>
**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:**

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>For a description of &lt;name&gt;, etc., click here.</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntermediatePT</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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
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.

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

<table>
<thead>
<tr>
<th>&lt;name&gt;</th>
<th>For a description of &lt;name&gt;, click here.</th>
</tr>
</thead>
</table>
| 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.
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
```

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).

**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.

---

**About the SMMU.StreamMapTable Window**

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

<table>
<thead>
<tr>
<th>Popup Menu</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1 Page Table &gt;</td>
<td>(--)</td>
</tr>
<tr>
<td>Stage 2 Page Table &gt;</td>
<td></td>
</tr>
<tr>
<td>• List</td>
<td>• <code>SMMU.StreamMapRegGrp.list</code></td>
</tr>
<tr>
<td>• Dump</td>
<td>• <code>SMMU.StreamMapRegGrp.Dump</code></td>
</tr>
<tr>
<td>Peripherals &gt;</td>
<td>(--)</td>
</tr>
<tr>
<td>• Global Configuration Registers</td>
<td>• <code>SMMU.Register.Global</code></td>
</tr>
<tr>
<td>• Stream Mapping Registers</td>
<td>• <code>SMMU.Register.StreamMapRegGrp</code></td>
</tr>
<tr>
<td>• Context Bank Registers of Stage 1</td>
<td>• <code>SMMU.Register.ContextBank</code></td>
</tr>
<tr>
<td>and Context Bank Registers of Stage 2</td>
<td></td>
</tr>
<tr>
<td>Security State Determination Table (SSD)</td>
<td><code>SMMU.SSDtoable</code></td>
</tr>
</tbody>
</table>
## Description of Columns: SMMU.StreamMapTable Window

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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)  
<p>| cbndx                              | Displays the context bank index (cbndx) associated with the translation context of <strong>stage 1</strong> or <strong>stage 2</strong>. |</p>
<table>
<thead>
<tr>
<th>Column Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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. |
### Description of Values

#### Values in the Column “context type”

<table>
<thead>
<tr>
<th>Description</th>
<th>Context type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context defines a stage 2 translation only</td>
<td>s2 translation only</td>
</tr>
<tr>
<td>Context defines a stage 1 translation, followed by a stage 2 translation (nested translation)</td>
<td>s1 trsl - s2 trsl</td>
</tr>
<tr>
<td>Context defines a stage 1 translation followed by a stage 2 fault</td>
<td>s1 trsl - s2 fault</td>
</tr>
<tr>
<td>Context defines a stage 1 translation followed by a stage 2 bypass</td>
<td>s1 trsl - s2 byp</td>
</tr>
<tr>
<td>Context defines a stage 1 translation followed by a stage 2 translation, but SMMU has no stage 1 (SMMU configuration fault)</td>
<td>fault (s1 trsl-s2 trsl)</td>
</tr>
<tr>
<td>Context defines a stage 1 translation followed by a stage 2 fault, but SMMU has no stage 1 (SMMU configuration fault)</td>
<td>fault (s1 trsl-s2 flt)</td>
</tr>
<tr>
<td>Context defines a stage 1 translation followed by a stage 2 bypass, but SMMU has no stage 1 (SMMU configuration fault)</td>
<td>fault (s1 trsl-s2 byp)</td>
</tr>
<tr>
<td>Context defines a fault</td>
<td>fault</td>
</tr>
<tr>
<td>Context defines bypass mode</td>
<td>bypass mode</td>
</tr>
<tr>
<td>Context type is improperly defined</td>
<td>reserved</td>
</tr>
</tbody>
</table>

#### 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”

<table>
<thead>
<tr>
<th>Description</th>
<th>Page table uses the 32-bit short descriptor format (32-bit targets only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short descr.</td>
<td>Page table uses the 32-bit long descriptor (LPAE) format (32-bit targets only)</td>
</tr>
<tr>
<td>Long descr.</td>
<td>Page table uses the 32-bit short descriptor format (64-bit targets only)</td>
</tr>
<tr>
<td>AArch32 Shrt</td>
<td>Page table uses the 32-bit long descriptor (LPAE) format (64-bit targets only)</td>
</tr>
<tr>
<td>AArch32 Long</td>
<td>Page table uses the 64-bit long descriptor (LPAE) format (64-bit targets only)</td>
</tr>
<tr>
<td>AArch64 Long</td>
<td>Page table uses the 64-bit long descriptor (LPAE) format (64-bit targets only)</td>
</tr>
</tbody>
</table>
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.
Target Adaption

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).
Connector Type and Pinout

Debug Cable


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

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


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


### Available Tools

<table>
<thead>
<tr>
<th>CPU</th>
<th>ICE</th>
<th>FIRE</th>
<th>ICD DEBUG</th>
<th>ICD MONITOR</th>
<th>ICD TRACE</th>
<th>POWER INTEGRATOR</th>
<th>INSTRUCTION SIMULATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>88F3710</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88F3720</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88F7020</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88F7040</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88F8020</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88F8040</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM6526</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM6527</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM6528</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM6546</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM6548</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APM883204-X1</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APM883208-X1</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APM883308-X1</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APM883408-X1</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>APM883408-X2</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCM2837</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCM4908</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCM5871</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CN81XX</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CN83XX</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CN99XX</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A32</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A35</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A53</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A55</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A57</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A72</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A73</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A75</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-A76</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORTEX-R52</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRA802M</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRA804M</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>ICE</td>
<td>FIRE</td>
<td>ICD DEBUG</td>
<td>ICD MONITOR</td>
<td>ICD TRACE</td>
<td>POWER INTEGRATOR</td>
<td>INSTRUCTION SIMULATOR</td>
</tr>
<tr>
<td>------------------</td>
<td>-----</td>
<td>------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
<td>------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>E2V_LS1043A</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2V_LS1046A</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2V_LS1088A</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2V_LS2088A</td>
<td>YES</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS5433</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS7420</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS7570</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS7580</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS7870</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS8890</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXYNOS8895</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALKOR</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8DUALX</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8MDUAL</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8MMINIQUAD</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8MQUAD</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8MQUADLITE</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8QUAD</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8QUADMAX</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8QUADPLUS</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8QUADX</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMX8QUADXPLUS</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KIRIN620</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KIRIN960</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRYO2XX</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRYOA</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1012A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1023A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1026A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1043A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1046A</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS1088A</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS2044A</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS2048A</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS2080A</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS2084A</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS2088A</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PXA1908</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PXA1918</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PXA1928</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PXA1936</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 32-Bit Compilers

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Company</th>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADA</td>
<td>GNAT PRO</td>
<td>AdaCore</td>
<td>ELF/DWARF</td>
<td>not all ADA constructs/DWARF</td>
</tr>
<tr>
<td>C</td>
<td>ARMCC</td>
<td>ARM Ltd.</td>
<td>AIF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>ARMCC</td>
<td>ARM Ltd.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>REALVIEW-MDK</td>
<td>ARM Ltd.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>GCCARM</td>
<td>Free Software Foundation, Inc.</td>
<td>COFF/STABS</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>GCCARM</td>
<td>Free Software Foundation, Inc.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>GREENHILLS-C</td>
<td>Greenhills Software Inc.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>ICCARM</td>
<td>IAR Systems AB</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>ICCV7-ARM</td>
<td>Imagecraft Creations Inc.</td>
<td>ELF/DWARF</td>
<td>ARM7</td>
</tr>
<tr>
<td>C</td>
<td>CARM</td>
<td>ARM Germany GmbH</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>HIGH-C</td>
<td>Synopsys, Inc</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>TI-C</td>
<td>Texas Instruments</td>
<td>COFF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>GNU-C</td>
<td>Wind River Systems</td>
<td>COFF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>D-CC</td>
<td>Wind River Systems</td>
<td>ELF</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>ARM-SDT-2.50</td>
<td>ARM Ltd.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>REALVIEW-MDK</td>
<td>ARM Ltd.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>Language</td>
<td>Compiler</td>
<td>Company</td>
<td>Option</td>
<td>Comment</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>----------------------------------------</td>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>C++</td>
<td>GCCARM</td>
<td>Free Software Foundation, Inc.</td>
<td>COFF/STABS</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>GNU</td>
<td>Free Software Foundation, Inc.</td>
<td>EXE/STABS</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>GCCARM</td>
<td>Free Software Foundation, Inc.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>GREENHILLS-C++</td>
<td>Greenhills Software Inc.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>MSVC</td>
<td>Microsoft Corporation</td>
<td>EXE/CV5</td>
<td>WindowsCE</td>
</tr>
<tr>
<td>C++</td>
<td>HIGH-C++</td>
<td>Synopsys, Inc</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>GNAT PRO</td>
<td>AdaCore</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>XCODE</td>
<td>Apple Inc.</td>
<td>Mach-O</td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>GCC</td>
<td>HighTec EDV-Systeme GmbH</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>VX-ARM</td>
<td>TASKING</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
</tbody>
</table>
64-Bit Compilers

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Company</th>
<th>Option</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>ARMCC (CLANG)</td>
<td>ARM Ltd.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>GREENHILLS-C</td>
<td>Greenhills Software Inc.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
<tr>
<td>C/C++</td>
<td>GNU-GCC</td>
<td>Free Software Foundation, Inc.</td>
<td>ELF/DWARF</td>
<td></td>
</tr>
</tbody>
</table>

Target Operating Systems

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>L4Re.org</td>
<td>Fiasco.OC</td>
<td></td>
</tr>
<tr>
<td>freeRTOS</td>
<td>FreeRTOS</td>
<td>up to v9</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Kernel Version 2.4, 2.6, 3.x, 4.x</td>
</tr>
<tr>
<td>Mentor Graphics Corporation</td>
<td>Nucleus</td>
<td></td>
</tr>
<tr>
<td>Sysgo AG</td>
<td>PikeOS</td>
<td>up to 4.2.1</td>
</tr>
<tr>
<td>QNX Software Systems</td>
<td>QNX</td>
<td>7.0</td>
</tr>
<tr>
<td>Wind River Systems</td>
<td>VxWorks</td>
<td>7.x</td>
</tr>
</tbody>
</table>

Boot Loaders

<table>
<thead>
<tr>
<th>Company</th>
<th>Product</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DENX Software Engineering</td>
<td>U-Boot</td>
<td></td>
</tr>
<tr>
<td>American Megatrends Inc.</td>
<td>UEFI Aptio V</td>
<td></td>
</tr>
<tr>
<td>Intel Corporation</td>
<td>UEFI TianoCore</td>
<td></td>
</tr>
</tbody>
</table>
## 3rd-Party Tool Integrations

<table>
<thead>
<tr>
<th>CPU</th>
<th>Tool</th>
<th>Company</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>WINDOWS CE PLATF. BUILDER</td>
<td>-</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CODE::BLOCKS</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C++TEST</td>
<td>-</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>ADENEO</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>X-TOOLS / X32</td>
<td>blue river software GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CODEWRIGHT</td>
<td>Borland Software Corporation</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CODE CONFIDENCE TOOLS</td>
<td>Code Confidence Ltd</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CODE CONFIDENCE TOOLS</td>
<td>Code Confidence Ltd</td>
<td>-</td>
<td>Linux</td>
</tr>
<tr>
<td>EASYCODE</td>
<td>EASYCODE GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>ECLIPSE</td>
<td>Eclipse Foundation, Inc</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CHRONVIEW</td>
<td>Inchron Gmbh</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>LDRA TOOL SUITE</td>
<td>LDRA Technology, Inc.</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>UML DEBUGGER</td>
<td>LieberLieber Software GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>SIMULINK</td>
<td>The MathWorks Inc.</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>ATTOL TOOLS</td>
<td>MicroMax Inc.</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>VISUAL BASIC INTERFACE</td>
<td>Microsoft Corporation</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>LABVIEW</td>
<td>NATIONAL INSTRUMENTS Corporation</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>TPT</td>
<td>PikeTec GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>CANTATA</td>
<td>QA Systems Ltd</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>RAPITIME</td>
<td>Rapita Systems Ltd.</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>RHAPSODY IN MICROC</td>
<td>IBM Corp.</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>RHAPSODY IN C++</td>
<td>IBM Corp.</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>TESSY</td>
<td>Razorcat Development GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>DA-C</td>
<td>RistanCASE</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>TRACEANALYZER</td>
<td>Symtavision GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>ECU-TEST</td>
<td>TraceTronic GmbH</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>UNDODB</td>
<td>Undo Software</td>
<td>-</td>
<td>Linux</td>
</tr>
<tr>
<td>TA INSPECTOR</td>
<td>Vector</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>VECTORCAST UNIT TESTING</td>
<td>Vector Software</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>VECTORCAST CODE COVERAGE</td>
<td>Vector Software</td>
<td>-</td>
<td>Windows</td>
</tr>
<tr>
<td>OrderNo</td>
<td>Code</td>
<td>Text</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| LA-3743 | JTAG-ARMV8-A/R | **Debugger for Cortex-A/R (ARMv8 32/64-bit)**  
  Supports ARMv8-A/R based Cortex-A and Cortex-R 32/64-bit cores  
  supports 5-pin standard JTAG, cJTAG and SWD,  
  (0.4 V - 5 V)  
  Multicore debugging included  
  includes software for Windows, Linux and MacOSX  
  requires Power Debug Module  
  cJTAG and SWD require  
  Power Debug Interface USB 2.0/USB 3.0,  
  Power Debug Ethernet, PowerTrace, Power Debug II  
  or PowerDebug PRO |
| LA-3743A | JTAG-ARMV8-A/R-A | **Debug Cortex-A/R (ARMv8 32/64-bit) Add.**  
  Supports ARMv8-A/R based Cortex-A and Cortex-R 32/64-bit cores  
  only suitable for debug cables newer than 07/2008  
  please add the base serial number of your debug cable to your order |
| LA-3743X | JTAG-ARMV8-A/R-X | **Debug Cortex-A/R (ARMv8 32/64-bit) Ext.**  
  Supports ARMv8-A/R based Cortex-A and Cortex-R 32/64-bit cores  
  only suitable for debug cables newer than 07/2008  
  please add the base serial number of your debug cable to your order |
| LA-3717  | MES-AD-JTAG20 | **Measuring Adapter JTAG 20**  
  Adapter to measure JTAG signals by a logic analyzer  
  or to disconnect single JTAG lines from the target |
| LA-3862  | CON-ARM/MIPI34-MIC | **ARM Conv. ARM-20, MIPI-34 to Mictor-38**  
  Converter to connect the ARM Debug Cable or  
  the CombiProbe to a Mictor connector on the target. This is needed if you want to debug  
  without a Preprocessor and if there is only  
  a Mictor connector on the target.  
  The trace signals of the CombiProbe are  
  connected to the lowest four trace signals of  
  the Mictor (ETMv3 pinout, continuous mode).  
  But tracing is normally no use case due to  
  the bandwidth limitations of the CombiProbe. |
| LA-2746  | CON-ARM20-ECU14 | **Conv. ARM-20 MIPI-20 to 10-pin ECU14**  
  ARM Converter ARM-20 or MIPI-20 to 10-pin ECU14 |
| LA-2720  | CON-ARM20-MIPS14 | **Converter ARM-20 to MIPS-14**  
  Converter to connect an ARM Debug Cable to a MIPS-14 connector  
  on the target |
## Order Information

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-3743</td>
<td>JTAG-ARMV8-A/R</td>
<td>Debugger for Cortex-A/R (ARMv8 32/64-bit)</td>
</tr>
<tr>
<td>LA-3717</td>
<td>MES-AD-JTAG20</td>
<td>Measuring Adapter JTAG 20</td>
</tr>
<tr>
<td>LA-3862</td>
<td>CON-ARM/MIPS34-MIC</td>
<td>ARM Conv. ARM-20, MIP34 to Mictor-38</td>
</tr>
<tr>
<td>LA-2746</td>
<td>CON-ARM20-ECU14</td>
<td>Conv. ARM-20 MIPI-20 to 10-pin ECU14</td>
</tr>
<tr>
<td>LA-2720</td>
<td>CON-ARM20-MIPS14</td>
<td>Converter ARM-20 to MIPS-14</td>
</tr>
</tbody>
</table>

### Additional Options

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Code</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-2101</td>
<td>AD-HS-20</td>
<td>Adapter Half-Size 20 pin</td>
</tr>
<tr>
<td>LA-3722</td>
<td>CON-JTAG20-MICTOR</td>
<td>ARM Converter ARM-20 to Mictor-38</td>
</tr>
<tr>
<td>LA-3770</td>
<td>CONV-ARM20/MPI34</td>
<td>Converter ARM-20 to MIPI-10/20/34 Arm/RISC-V</td>
</tr>
<tr>
<td>LA-3788</td>
<td>DAISY-CHAINER-JTAG20</td>
<td>Daisy Chainer 4 JTAG 20</td>
</tr>
<tr>
<td>LA-7760A</td>
<td>EJTAG-MIPS32-A</td>
<td>Debugger License for MIPS32 Add.</td>
</tr>
<tr>
<td>LA-3501</td>
<td>GALVANIC-ISOLATION</td>
<td>Galvanic Isolation for Debug Cable</td>
</tr>
<tr>
<td>LA-3744A</td>
<td>JTAG-APEX-A</td>
<td>JTAG Debugger Lic. for APEX ADD.</td>
</tr>
<tr>
<td>LA-3750A</td>
<td>JTAG-ARC-A</td>
<td>JTAG Debugger License for ARC Add.</td>
</tr>
<tr>
<td>LA-7747</td>
<td>JTAG-ARM-CON-14-20</td>
<td>ARM Converter ARM-20 to/from ARM-14</td>
</tr>
<tr>
<td>LA-3726</td>
<td>JTAG-ARM-CON-20-20</td>
<td>ARM Converter 2x ARM-20 to ARM-20</td>
</tr>
<tr>
<td>LA-7748</td>
<td>JTAG-ARM-CON-20-TI14</td>
<td>Converter ARM-20 to TI-14</td>
</tr>
<tr>
<td>LA-3780</td>
<td>JTAG-ARM-CON-20-TI20</td>
<td>Converter ARM-20 to TI-14 or TI-20-Compact</td>
</tr>
<tr>
<td>LA-7744X</td>
<td>JTAG-ARM10-X</td>
<td>JTAG Debugger Extension for ARM10</td>
</tr>
<tr>
<td>LA-7765X</td>
<td>JTAG-ARM11-X</td>
<td>JTAG Debugger Extension for ARM11</td>
</tr>
<tr>
<td>LA-7746X</td>
<td>JTAG-ARM7-X</td>
<td>JTAG Debugger Extension for ARM7</td>
</tr>
<tr>
<td>LA-7742X</td>
<td>JTAG-ARM9-X</td>
<td>JTAG Debugger Extension for ARM9</td>
</tr>
<tr>
<td>LA-2705A</td>
<td>JTAG-ARP32-A</td>
<td>JTAG Debugger for ARP32 Add.</td>
</tr>
<tr>
<td>LA-7838A</td>
<td>JTAG-C6XXX-A</td>
<td>JTAG Debugger License for TMS320C6xxx Add.</td>
</tr>
<tr>
<td>LA-3711A</td>
<td>JTAG-CEVAX-A</td>
<td>JTAG Debugger License for CEVA-X Additional</td>
</tr>
<tr>
<td>LA-3741A</td>
<td>JTAG-HEXAGON-A</td>
<td>JTAG Debugger License for Hexagon Add.</td>
</tr>
<tr>
<td>LA-3730A</td>
<td>JTAG-MICROBLAZE-A</td>
<td>JTAG Debug. License for MicroBlaze Additonal</td>
</tr>
<tr>
<td>LA-2756A</td>
<td>JTAG-PRU-A</td>
<td>JTAG Debugger for PRU Add.</td>
</tr>
<tr>
<td>LA-2717A</td>
<td>JTAG-RISC-V-A</td>
<td>JTAG Debugger for RISC-V Add.</td>
</tr>
<tr>
<td>LA-7774</td>
<td>JTAG-TEAK-JAM-20</td>
<td>JTAG Debugger for Teak/TeakLite JAM 20 (ICD)</td>
</tr>
<tr>
<td>LA-3747A</td>
<td>JTAG-UBI32-A</td>
<td>JTAG/SPi Debugger License for UBI32 Add.</td>
</tr>
<tr>
<td>LA-7762X</td>
<td>JTAG-XSCALE-X</td>
<td>JTAG Debugger Extension for XSCALE (ICD)</td>
</tr>
<tr>
<td>LA-3760A</td>
<td>JTAG-XTENSA-A</td>
<td>JTAG Debugger License for Xtensa Add.</td>
</tr>
<tr>
<td>LA-7970X</td>
<td>TRACE-LICENSE-ARM</td>
<td>Trace License for ARM (Debug Cable)</td>
</tr>
</tbody>
</table>