LAUTERBACH
DEVELOPMENT TOOLS

STRED Debugger and Trace

MANUAL

STRED Debugger and Trace

STRED Debugger and Trace

Version 06-Jun-2024

Introduction

This documentation describes the processor specific settings and features for the TRACE32 debugger.

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

Brief Overview of Documents for New Users

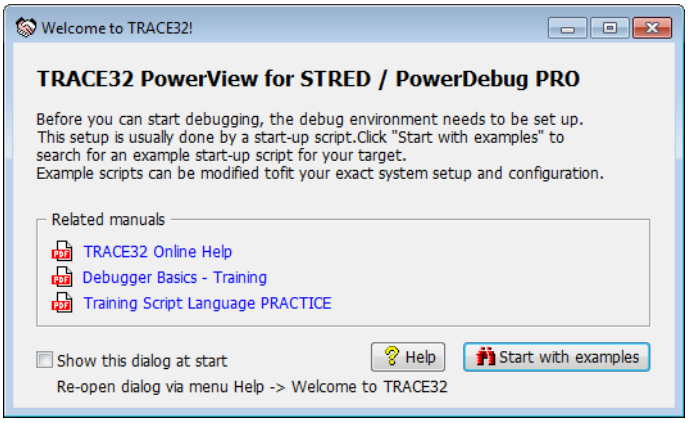
Architecture-independent information:

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

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

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



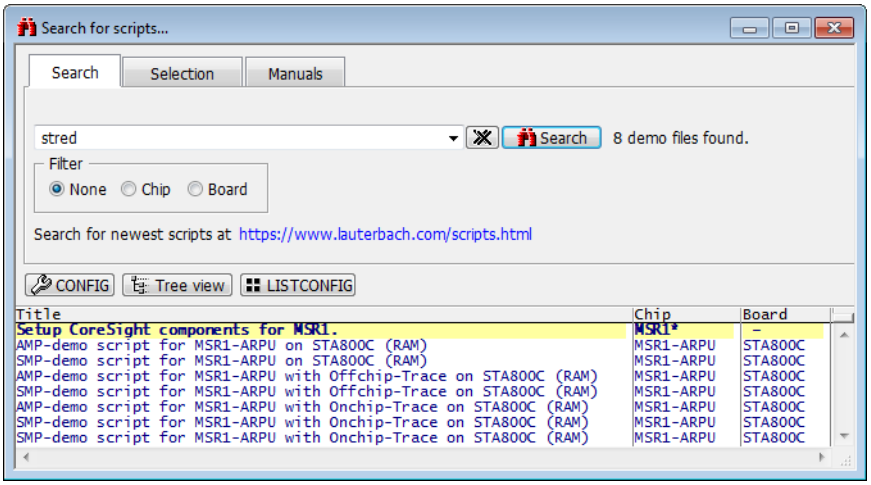
Demo and Start-up Scripts

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

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

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

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



You can also manually navigate in the ~~/`demo/stred/` subfolder of the system directory of TRACE32.

Warning

WARNING:	To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF. Recommendation for the software start: 1. Disconnect the Debug Cable from the target while the target power is off. 2. Connect the host system, the TRACE32 hardware and the Debug Cable. 3. Power ON the TRACE32 hardware. 4. Start the TRACE32 software to load the debugger firmware. 5. Connect the Debug Cable to the target. 6. Switch the target power ON. 7. Configure your debugger e.g. via a start-up script. Power down: 1. Switch off the target power. 2. Disconnect the Debug Cable from the target. 3. Close the TRACE32 software. 4. Power OFF the TRACE32 hardware.

Starting up the debugger is done as follows:

1. Reset the debugger.

```
RESet
```

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

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

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

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

3. Connect to target.

```
SYStem.Up
```

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

You might want to connect to a running program without causing a target reset. In this case use **SYStem.Mode Attach** instead. A **Break.direct** command will halt the processor.

4. Load the program you want to debug.

```
Data.LOAD.Elf sieve.elf
```

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

Troubleshooting

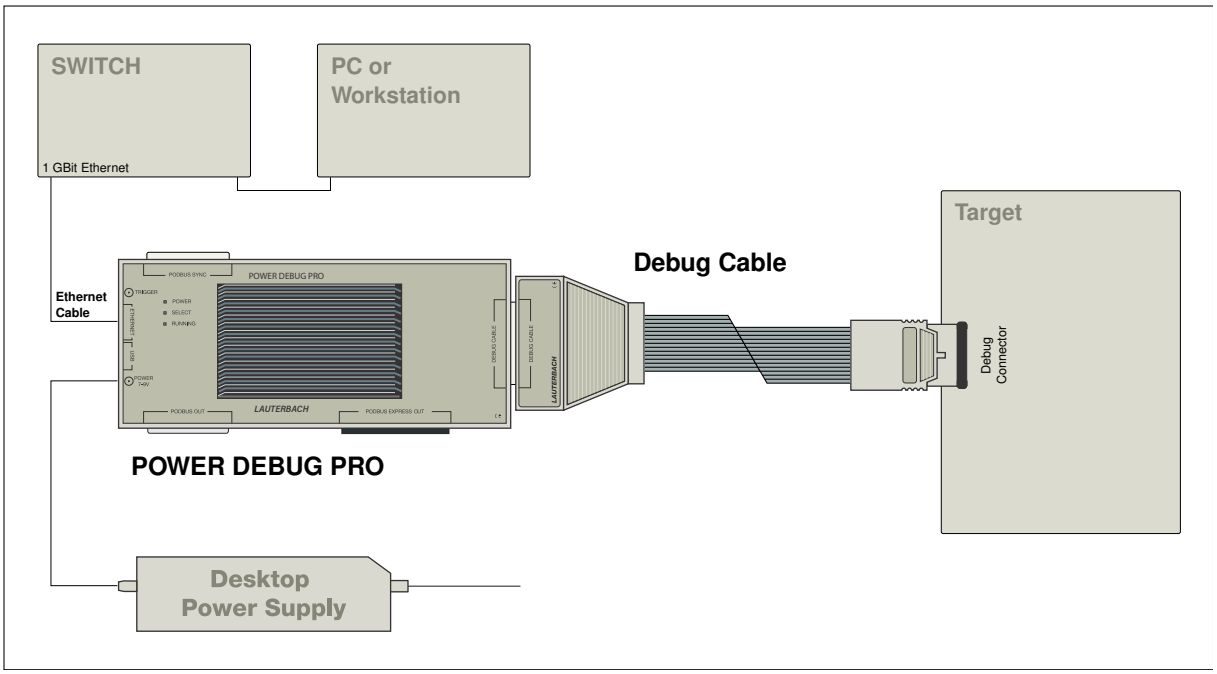
SYStem.Up Errors

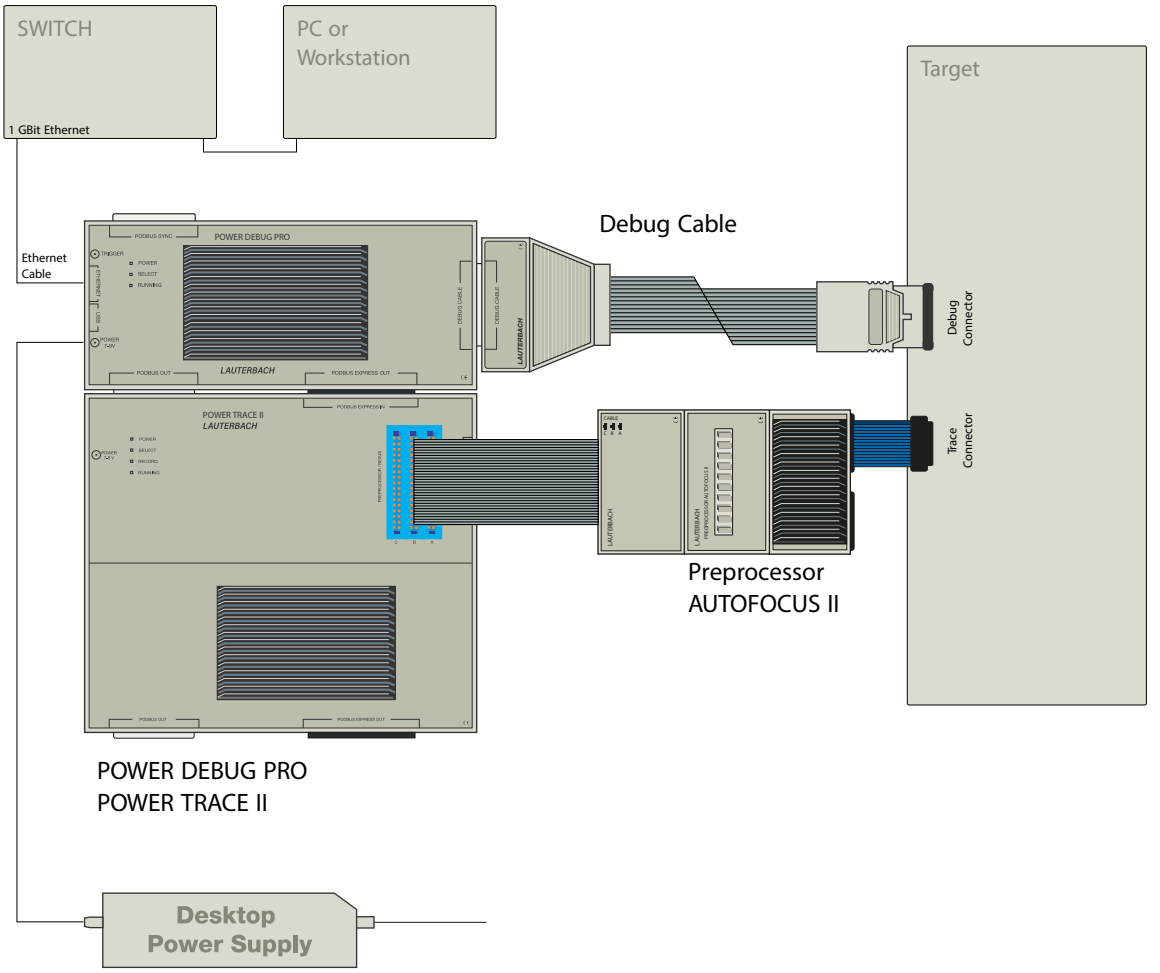
No information available.

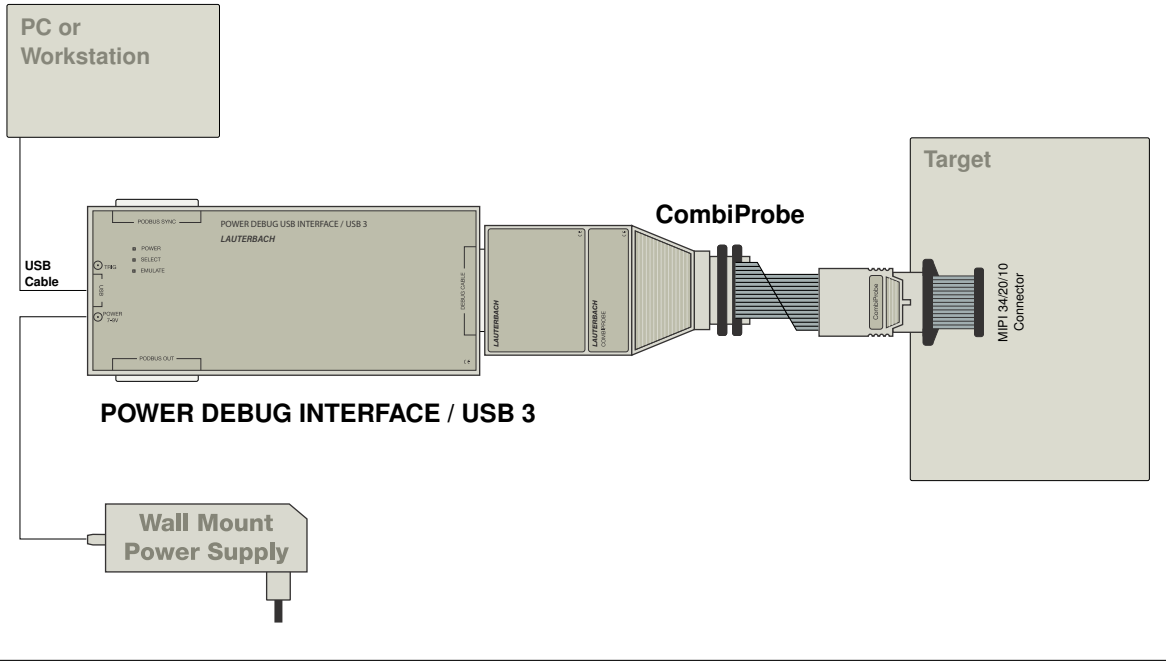
FAQ

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

System Overview




SWITCH
1 GBit Ethernet
PC or
Workstation
Target
Debug Cable
Ethernet
Cable
POWER DEBUG PRO
POWER TRACE II
LAUTERBACH
Debug
Connector
Trace
Connector
Preprocessor
AUTOFOCUS II
POWER DEBUG PRO
POWER TRACE II
Desktop
Power Supply


PC or
Workstation
Target
CombiProbe
USB
Cable
POWER DEBUG USB INTERFACE / USB 3
LAUTERBACH
MIPI 34/20/10
Connector
POWER DEBUG INTERFACE / USB 3
Wall Mount
Power Supply

Breakpoints

There are two implementations for breakpoints:

- Software breakpoints
- On-chip breakpoints

Software Breakpoints

In order to stop the program execution at a selected instruction, the code at the break location is patched by an **illegal instruction**. If the **illegal instruction** comes to the execution stage of the pipeline, an illegal instruction trap is raised, the program execution is stopped by the debug code and the debug mode becomes active.

Software breakpoints can be set to instructions in RAM and with some preparations also to instructions in FLASH (see **FLASH.Create** and **FLASH.AUTO**). Software breakpoints on instructions in FLASH should only be used, if the number of on-chip breakpoints is insufficient.

The number of software breakpoints is unlimited.

On-chip Breakpoints

This implementation is called on-chip, because the debugger uses resources provided by the processor to set a breakpoint.

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

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

Family	On-chip Breakpoints	Instruction Breakpoints	Read/Write Breakpoints	Data Breakpoints
STRED	4 instruction 1 read/write	4 single address	1 range as bit mask break before make ReadWrite only	—

On-chip Breakpoints on instructions

On-chip breakpoints are handled by the CPU internally and do not require to modify the program memory. Therefore they can be used to set a breakpoint on an instruction in FLASH or ROM.

With the command **MAP.BOnchip** *<range>* it is possible to instruct the debugger to use On-chip breakpoints for the specified range as default (it is still possible to override this with parameters like /SOFT for the break.set command). Typically it is used for FLASH/ROM memories. If a breakpoint is set within the specified address range, the debugger uses automatically the available on-chip breakpoints.

Use the command **MAP.List** to see for which address ranges the debugger uses on-chip breakpoints.

Example for Breakpoints

For example, if a target system has FLASH from `0x0` to `0xFFFFF` and RAM from `0x100000` to `0x11FFFF`. The command to configure TRACE32 correctly for this configuration is:

```
Map.BOnchip 0x0--0xFFFFF
```

Example software breakpoints:

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

Break.Set func1 /Program                 ; Software BP on symbol in RAM
```

Example on-chip breakpoints:

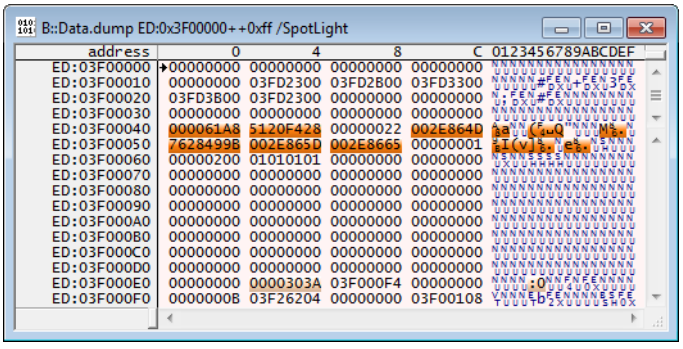
```
Break.Set 0x1000 /Program                ; On-chip BP on FLASH address

Break.Set func2 /Program                 ; On-chip BP on symbol in FLASH

Break.Set 0x101000 /Program /Onchip      ; Force On-chip BP on RAM address
```

Runtime Access

The STRED can read/write memory while the program execution is running, but can not read the Program Counter (PC). Runtime access can be enabled or disabled with the **SYStem.MemAccess** command. To access memory at runtime, the access class extension E should be used.

Example:

```
; view memory with runtime access
Data.dump ED:0x3F00000++0xff /SpotLight
```



Access Classes

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

The following access classes are available for STRED:

Access Class	Description
AHB, APB	Memory access via bus masters, so named Memory Access Ports (MEM-AP), provided by a Debug Access Port (DAP). The DAP is a ARM CoreSight component.
D	Data Memory
DSU	Memory access via Debug Support Unit (DSU) via JTAG Access Port (JTAG-AP) provided by a Debug Access Port (DAP). For target systems without DAP, the DSU is directly accessed via JTAG.
E	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)
P	Program Memory
VM	Virtual Memory (memory on the debug system)

Example:

```
; view data memory
Data.dump D:0x0000--0x1fff

; view program code
List.auto P:0x10000

; view memory via DSU access
Data.dump DSU:0x0000--0x1fff

; view memory via AHB access
Data.dump AHB:0x0000--0x1fff
```

Debug Code

Debugging a STRED CPU requires a few instructions of debug code to support operations that can only be done by the CPU, but not directly by the debugger. For example, to halt a running CPU, the debugger requests a debug interrupt and the CPU is halted by the zzz instruction that is part of the debug code.

The instruction sequence of the debug code is:

```
sync
zzz
rfe
msr       %r0, %sr2
jpda      +4
stw       [%r0], %r1
.word     0xffff
```

The binary code is:

```
0x03, 0x0E, 0x00, 0x00, 0x00, 0xFE, 0x00, 0xF8
0x02, 0xFB, 0x02, 0xDC, 0x10, 0x9B, 0xFF, 0xFF
```

By default, the debug code is written to the CPU memory on SYStem.Mode Up or SYStem.Mode Attach. The debugger will also set the debug interrupt handler and the trap handler to the base address of the debug code. The base address of the debug code can be set with **SYStem.Option.DbgBase**.

If the debug code is included in the executed program code, **SYStem.Option.DbgOvwr** can be used to prevent the debugger from writing the debug code. In this case, **SYStem.Option.DbgBase** must be used to inform the debugger about the location of the debug code.

If the executed program code implements it's own trap handler routine, **SYStem.Option.DbgTrap** can be used to prevent the debugger from overwriting the trap handler. In this case, the program code should also include the debug code and must jump to the debug code base address when a debug trap (illegal instruction, ibreak or dbreak) is detected.

Tracing

Processors of the STRed series implement an **RTM** for trace support. This section provides examples for typical RTM trace configurations. It does not include configurations specific for certain chips or trace hardware.

For information about Coresight component configuration please refer to **"Setup of the Debugger for a CoreSight System"** (app_arm_coresight.pdf). For information about trace hardware setup refer to the **"AutoFocus User's Guide"** (autofocus_user.pdf). For information about target system specific trace configurations refer to the technical reference manual or data sheet of the respective chip or target system.

Example: Basic RTM Flow Trace

The trace capture setup is controlled by the **RTM** command group. This examples shows a simple setup for RTM flow trace on a single core. It is possible to select multiple cores for trace. By default all cores are traced..

```
RTM.Trace ON        ; RTM conditions and filter are set by the debugger
RTM.CoreENable 0.   ; trace only core 0
RTM.ON              ; enable trace recording
```

Example: RTM Flow Trace with Manual Register Configuration

This example demonstrates a manual RTM configuration and behaves exactly as the previous flow trace example. Setting **RTM.Trace** to "OFF" prevents the debugger from overwriting the RTM condition, trigger and filter registers. As this method provides full control over the RTM registers, it can be adjusted for more sophisticated use cases.

Please refere to the ReISC Trace Module Specification for more information about how to configure the RTM registers.

```
RTM.Trace OFF

PRIVATE &rtm
&rtm=COMPonent.BASE("RTM",0)

Data.Set &rtm+0x000 %Long 0x301 ; enable funnel ports
Data.Set &rtm+0x380 %Long 0x001 ; enable log for processor 0
Data.Set &rtm+0x280 %Long 0x488 ; enable flow trace

RTM.ON
```

Example: Basic RTM Data Trace

This examples shows a setup for RTM flow trace and data trace. Every read and write access to any address is recorded.

```
RTM.Trace ON       ; RTM conditions and filter are set by the debugger
RTM.CoreENable 0.  ; trace only core 0
RTM.DataTrace ON   ; enable data trace
RTM.ON             ; enable trace recording
```

Example: RTM Data Trace with Address Range

This example uses the manual RTM register configuration to set up a data trace restricted to a specific address range. The **RTM.DataTrace** configuration is ignored when **RTM.Trace** is set to "OFF".

```
RTM.Trace OFF

PRIVATE &rtm
&rtm=COMPonent.BASE("RTM",0)

Data.Set &rtm+0x000 %Long 0x301      ; enable all funnel ports
Data.Set &rtm+0x380 %Long 0x001      ; enable log for processor 0
Data.Set &rtm+0x280 %Long 0x488      ; enable flow trace

; enable data trace in the address range 0x1000--0x1FFFF
Data.Set &rtm+0x180 %Long 0x1000     ; address range start
Data.Set &rtm+0x184 %Long 0x1FFF     ; address range end
Data.Set &rtm+0x188 %Long 0x03000001 ; log address and data trace
Data.Set &rtm+0x284 %Long 0x00020000 ; enable data trace

RTM.ON
```

SYStem.CONFIG Configure debugger according to target topology

Format: **SYStem.CONFIG** *<parameter>*

<parameter>: **(DebugPort)**

CONNECTOR
CORE *<core>* *<chip>*
CoreNumber *<number>*
DEBUGPORT [**DebugCable0** | **DebugCableA** | **DebugCableB**]
DEBUGPORTTYPE [**JTAG** | **SWD**]
Slave [**ON** | **OFF**]
SWDP [**ON** | **OFF**]
SWDPIdleHigh *<value>*
SWDPTargetSel *<value>*
TriState [**ON** | **OFF**]

<parameter>: **(JTAG)**

DRPOST *<bits>*
DRPRE *<bits>*
IRPOST *<bits>*
IRPRE *<bits>*
TAPState *<state>*
TCKLevel *<level>*

<parameter>: **(AccessPorts)**

AHBAPn.Base *<address>*
AHBAPn.HPort *<value>* *<name>*
AHBAPn.Port *<port>*
AHBAPn.RESet
AHBAPn.view
AHBAPn.XtorName *<name>*

APBAPn.Base *<address>*
APBAPn.Port *<port>*
APBAPn.RESet
APBAPn.view
APBAPn.XtorName *<name>*

AXIAPn.ACEEnable [**ON** | **OFF**]
AXIAPn.Base *<address>*
AXIAPn.CacheFlags *<value>*
AXIAPn.HPort *<value>* *<name>*
AXIAPn.Port *<port>*
AXIAPn.RESet
AXIAPn.view
AXIAPn.XtorName *<name>*

<*parameter*>: (AccessPorts cont.)

DEBUGAPn.Port *<port>*
DEBUGAPn.RESet
DEBUGAPn.view
DEBUGAPn.XtorName *<name>*

JTAGAPn.Base *<address>*
JTAGAPn.Port *<port>*
JTAGAPn.CorePort *<port>*
JTAGAPn.RESet
JTAGAPn.view
JTAGAPn.XtorName *<name>*

MEMORYAPn.HPort *<value>* *<name>*
MEMORYAPn.Port *<port>*
MEMORYAPn.RESet
MEMORYAPn.view
MEMORYAPn.XtorName *<name>*

<*parameter*>: **(COmponents)**

CTI.Base *<address>*
CTI.Config *<type>*
CTI.RESet
CTI.view
CTI.Config [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**]

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

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

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

REP.ATBSource *<source>*
REP.Base *<address>*
REP.Name *<string>*

<parameter>: (COmponents cont.)	**REP.RESet** **REP.view** **RTM.Base** *<address>* **RTM.RESet** **STM.Base** *<address>* **STM.Mode** *<mode>* **STM.RESet** **STM.Type** *<type>* **TPIU.ATBSource** *<source>* **TPIU.Base** *<address>* **TPIU.Name** *<string>* **TPIU.RESet** **TPIU.Type** [**CoreSight** | **Generic**]
<parameter>: **(Core)**	**DSP.ProcessorID** **DSP.RESet** **DSP.state** **DSP.***<register> <address>*
<parameter>: **(Deprecated)**	**CTICONFIG** [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1** | **ARPU**]

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

Default values are set by **SYStem.CPU** depending on the selected chip. Ideally there is no need for any further **SYStem.CONFIG** command, but some target systems require to adjust the default setup.

SYStem.CONFIG settings shall be modified after the **SYStem.CPU** command and before starting the debug session e.g. by **SYStem.Up**.

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 a CombiProbe is used, MIPI20T if a µTrace (MicroTrace) or debug cable 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 multiple debugger instances (multiple TRACE32 PowerView GUIs) are used to simultaneously debug different cores on the same target system.

Because of the default setting of this command

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

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

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

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

CoreNumber *<numer>*

Configures the number of cores to be considered in a Symmetric MultiProcessing (SMP) debug session. There are core types which can be used as single core processor or as scalable multicore processor (with all cores of the same type). If more than one core should be debugged in an SMP debug session it is necessary to specify the number of cores.

Default: 1.

DEBUGPORT [**DebugCable0** | **DebugCableA** | **DebugCableB**]

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.

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

Specifies the used debug port type. It assumes the selected type is supported by the target.

Default: JTAG.

Slave [**ON** | **OFF**]

If several debuggers share the same debug port, all except one must have this option active.

JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave ON**.

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

SWDP [**ON** | **OFF**]

With this command the normal JTAG interface can be changed to the serial wire debug mode. SWDP (Serial Wire Debug Port) uses just two signals instead of five. It is required that the target and the debugger hard- and software supports this interface.

Default: OFF.

SWDPIdleHigh *<value>*

Keep SWDIO line high when idle. Only for Serial Wire 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).

Default: OFF.

SWDPTargetSel *<value>*

Device address in case of a multidrop serial wire debug port.

Default: none set (any address accepted).

TriState [**ON** | **OFF**]

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

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

Default: OFF.

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

A JTAG interface provides access to a Test Access Port controller (TAP). The TAP implements a state machine to read and write data to an Instruction Register (IR) and a Data Register (DR).

A JTAG interface is controlled by 5 signals:

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

Multiple TAPs can be controlled via a single JTAG interface by daisy-chaining the TAPs (serial connection). To access one specific TAP in the chain, a BYPASS pattern (all ones) must be sent to all other TAPs. To handle this the debugger needs to know the position of the TAP within the JTAG scan chain. The TAP position can be defined with the DRPOST, DRPRE, IRPOST and IRPRE commands in the table below.

DRPOST *<bits>*

Defines the TAP position in a JTAG scan chain. Should be configured to the number of TAPs in the JTAG chain between the TDI signal and the described TAP, as every TAP contributes one data register bit in BYPASS mode.

For target systems where the core TAP is only accessibly via the Debug Access Port (DAP), this command defines the position of the core TAP on the JTAG Access Port (JTAG-AP) of the DAP.

Default: 0.

DRPRE *<bits>*

Defines the TAP position in a JTAG scan chain. Should be configured to the number of TAPs in the JTAG chain between the described TAP and the TDO signal, as every TAP contributes one data register bit in BYPASS mode.

For target systems where the core TAP is only accessibly via the Debug Access Port (DAP), this command defines the position of the core TAP on the JTAG Access Port (JTAG-AP) of the DAP.

Default: 0.

IRPOST *<bits>*

Defines the TAP position in a JTAG scan chain. Should be configured to the number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the TDI signal and the described TAP.

For target systems where the core TAP is only accessibly indirectly via the ARM CoreSight DAP, this command defines the position of the core TAP on the JTAG Access Port (JTAG-AP) of the DAP.

Default: 0.

IRPRE *<bits>*

Defines the TAP position in a JTAG scan chain. Should be configured to the number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the described TAP and the TDO signal.

For target systems where the core TAP is only accessibly via the Debug Access Port (DAP), this command defines the position of the core TAP on the JTAG Access Port (JTAG-AP) of the DAP.

Default: 0.

TAPState *<state>*

This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.

0 Exit2-DR
1 Exit1-DR
2 Shift-DR
3 Pause-DR
4 Select-IR-Scan
5 Update-DR
6 Capture-DR
7 Select-DR-Scan
8 Exit2-IR
9 Exit1-IR
10 Shift-IR
11 Pause-IR
12 Run-Test/Idle
13 Update-IR
14 Capture-IR
15 Test-Logic-Reset

Default: 7 = Select-DR-Scan.

TCKLevel *<level>*

Level of the TCK signal when all debuggers are in tristate mode. Usually defined by a pull-up or pull-down resistor on the target.

Default: 0.

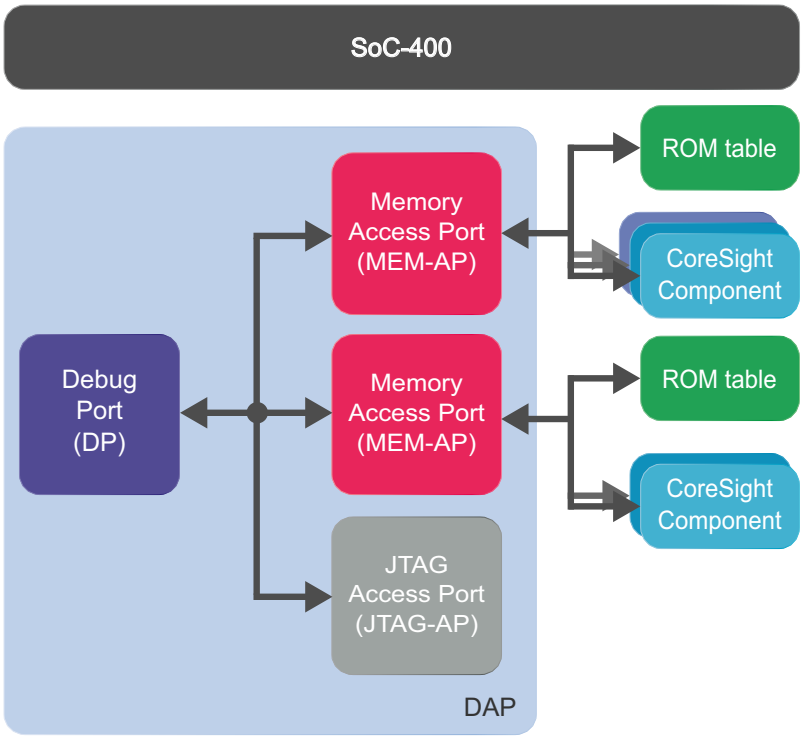
An Access Port (AP) is a CoreSight module from Arm which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

1. Different memory busses (AHB, APB, AXI). This is especially important if the on-chip debug register needs to be accessed this way. You can access the memory buses by using certain access classes with the debugger commands: “AHB:”, “APB:”, “AXI:, “DP”, “E:”. The interface to these buses is called Memory Access Port (MEM-AP).

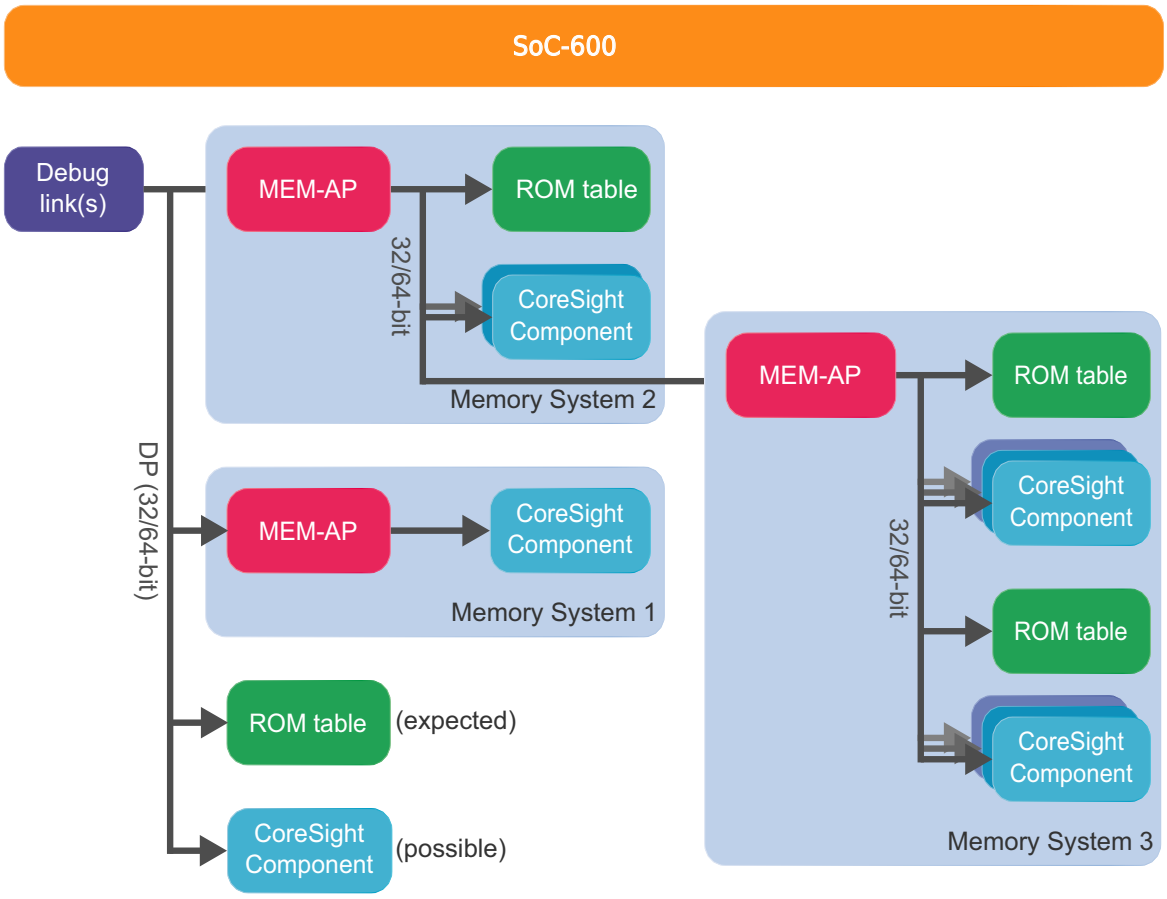
2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number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

3. A transactor name for virtual connections to AMBA bus level transactors can be configured by the property **SYStem.CONFIG.*APn.XtorName** *<name>*. A JTAG or SWD transactor must be configured for virtual connections to use the property “Port” or “Base” (with “DP:” access) in case XtorName remains empty.

Example 1: SoC-400



Example 2: SoC-600



AHBAPn.HPROT *<port>*
SYStem.Option.AHBH-PROT *<port>* (deprecated)

Default: 0.
Selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight AHB Access Port, when using the AHB: memory class.

AXIAPn.HPROT *<port>*
SYStem.Option.AXIHPROT *<port>* (deprecated)

Default: 0.
This option selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight AXI Access Port, when using the AXI: memory class.

MEMORYAPn.HPROT *<port>*

Default: 0.
This option selects the value used for the HPROT bits in the Control Status Word (CSW) of a CoreSight Memory Access Port, when using the E: memory class.

AXIAPn.ACEEnable [**ON** | **OFF**]
SYStem.Option.AXIACEEnable [**ON** | **OFF**] (deprecated)

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

AXIAPn.CacheFlags *<value>*
SYStem.Option.AXICACHEFLAGS *<value>* (deprecated)

Default: DeviceSYStem (=0x30: Domain=0x3, Cache=0x0).
This option configures the value used for the Cache and Domain bits in the Control Status Word (CSW[27:24]->Cache, CSW[14:13]->Domain) of an Access Port, when using the AXI: memory class.

The below offered selection options are all non-bufferable. Alternatively you can enter a <value>, where value[5:4] determines the Domain bits and value[3:0] the Cache bits.

<name>	Description
DeviceSYStem	=0x30: Domain=0x3, Cache=0x0
NonCacheableSYStem	=0x32: Domain=0x3, Cache=0x2
ReadAllocateNonShareable	=0x06: Domain=0x0, Cache=0x6
ReadAllocateInnerShareable	=0x16: Domain=0x1, Cache=0x6
ReadAllocateOuterShareable	=0x26: Domain=0x2, Cache=0x6
WriteAllocateNonShareable	=0x0A: Domain=0x0, Cache=0xA
WriteAllocateInnerShareable	=0x1A: Domain=0x1, Cache=0xA
WriteAllocateOuterShareable	=0x2A: Domain=0x2, Cache=0xA
ReadWriteAllocateNonShareable	=0x0E: Domain=0x0, Cache=0xE
ReadWriteAllocateInnerShareable	=0x1E: Domain=0x1, Cache=0xE
ReadWriteAllocateOuterShareable	=0x2E: Domain=0x2, Cache=0xE

AHBAPn.XtorName *<name>*
AHB bus transactor name that shall be used for “AHBn:” access class.

APBAPn.XtorName *<name>*
APB bus transactor name that shall be used for “APBn:” access class.

AXIAPn.XtorName *<name>*
AXI bus transactor name that shall be used for “AXIn:” access class.

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

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

... .RESet
Undo the configuration for this access port. This does not cause a physical reset for the access port on the chip.

... .view
Opens a window showing the current configuration of the access port.

AHBAPn.Port *<port>*
Access Port Number (0-255) of a SoC-400 system which shall be used for “AHBn:” access class. Default: *<port>*=0.

APBAPn.Port *<port>*
Access Port Number (0-255) of a SoC-400 system which shall be used for “APBn:” access class. Default: *<port>*=1.

AXIAPn.Port *<port>*
Access Port Number (0-255) of a SoC-400 system which shall be used for “AXIn:” access class. Default: port not available.

DEBUGAPn.Port *<port>*
AP access port number (0-255) of a SoC-400 system where the debug register can be found (typically on APB). Used for “DAP:” access class. Default: *<port>*=1.

JTAGAPn.CorePort *<port>*
JTAG-AP port number (0-7) connected to the core which shall be debugged.

JTAGAPn.Port *<port>*
Access port number (0-255) of a SoC-400 system of the JTAG Access Port.

MEMORYAPn.Port *<port>*
AP access port number (0-255) of a SoC-400 system where system memory can be accessed even during runtime (typically an AHB). Used for “E:” access class while running, assuming “**SYStem.MemAccess DAP**”. Default: *<port>*=0.

AHBAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “AHBAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.AHBAP1.Base DP:0x80002000
Meaning: The control register block of the AHB access ports starts at address 0x80002000.

APBAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “APBAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.APBAP1.Base DP:0x80003000
Meaning: The control register block of the APB access ports starts at address 0x80003000.

AXIAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “AXIAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

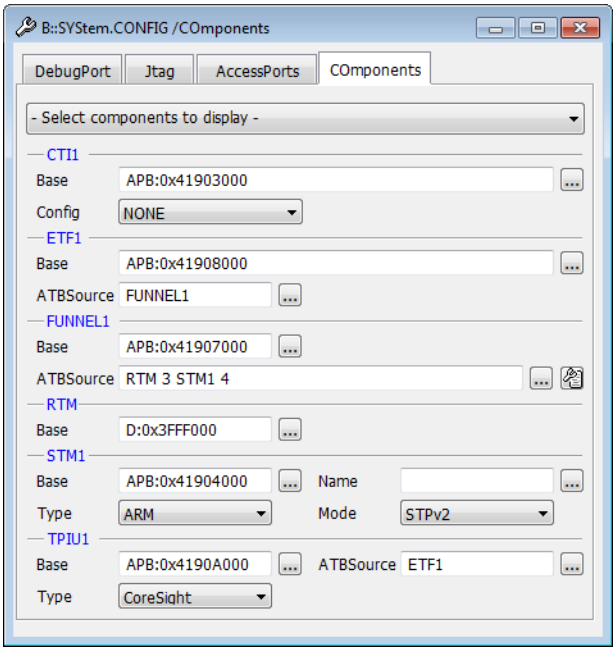
Example: SYStem.CONFIG.AXIAP1.Base DP:0x80004000
Meaning: The control register block of the AXI access ports starts at address 0x80004000.

JTAGAPn.Base *<address>*

This command informs the debugger about the start address of the register block of the “JTAGAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Example: SYStem.CONFIG.JTAGAP1.Base DP:0x80005000
Meaning: The control register block of the JTAG access ports starts at address 0x80005000.

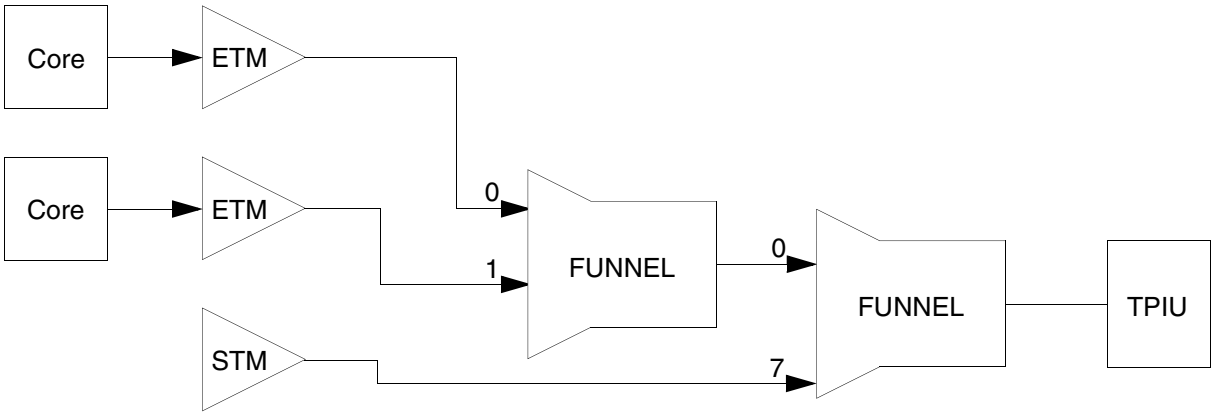
The debugger must be configured to know all debug and trace components of the target system which shall be used for debugging. On the **Components** tab in the **SYStem.CONFIG.state** window, the current configuration of debug and trace components can be viewed:



New components can be added here or with the respective commands (see the table below).

It is possible to have several of the following components: ETB, ETF, ETR, FUNNEL, STM.
For these components, a index (starting from 1) is appended to the component name to create a unique identifier, e.g. FUNNEL1, FUNNEL2. A component identifier without index always refers to the first component (with index 1)., e.g. FUNNEL equals FUNNEL1.

Example:




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

<component>**.ATBSource** *<source>*

Configures for trace information collecting components from where the trace data is coming. This way the debugger is informed about the interconnection of different trace components.

A base address must be configured for components using the **<component>.Base** command before they can used as source for **<component>.ATBSource <source>**.

A CoreSight trace FUNNEL has eight input ports (port 0-7) to combine the data of various trace sources to a common trace stream. Therefore <source> is a list of up to eight source and port pairs when a FUNNEL is configured.

Example: FUNNEL1.ATBSource ETF1 0 ETF2 1 ETF3 5

In an Symmetric MultiProcessing (SMP) debug session, a list of base addresses can be used to specify one component per core. To identify a certain component as source, the core index (starting from 0) must be appended to the source name.

Example:
ETF.Base 0x1000 0x2000 0x3000
FUNNEL.ATBSource ETF.0 0 ETF.1 2 ETF.2 4

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

*<component>***.Base** *<address>*

Configures the start address of a register block used to configure a component. This is also used to notify the existence of the component.

In an Symmetric MultiProcessing (SMP) debug session, a list of base addresses can be used to specify one component per core for CTI, ETB, ETF and ETR:

*<component>***.Base** *<address1> <address2> <address3> ...*

Example: ETF.Base 0x1000 0x2000 0x3000

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

... **.Name**

The name is a freely configurable identifier to describe how many instances exists in a target systems chip. TRACE32 PowerView GUI shares with other opened PowerView GUIs settings and the state of components identified by the same name and component type. Components using different names are not shared. Other attributes as the address or the type are used when no name is configured.

Example: **Shared None-Programmable Funnel:**
PowerView1:
SYStem.CONFIG.FUNNEL.PROGramable OFF
SYStem.CONFIG.FUNNEL.Name "shared-funnel-1"
PowerView2:
SYStem.CONFIG.FUNNEL.PROGramable OFF
SYStem.CONFIG.FUNNEL.Name "shared-funnel-1"
SYStem.CONFIG.Core 2. 1. ; merge configuration to describe a target system with one chip containing a single none-programmable FUNNEL.

<component>.**RESet** — Undo the configuration for the 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 **Available Components**.

<component>.**view** — Opens a window showing the current configuration of the component.

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

CTI.Config *<type>* — Configures the interconnection of 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.
ARPU: This mode should be used for a MSR1-ARPU CTI.
others: Please check the ARM or ARMv8 manual for other CTI types.

ETB.NoFlush [**ON** | **OFF**] — Deactivates the ETB flush request at the end of trace recording. This is a workaround for a hardware bug on certain chips. Some trace data is lost at the end of the recording. Don't use it if not needed.

ETB.Size *<size>* — Specifies the size of the Embedded Trace Buffer (ETB). 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.

ETB.STackMode [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULLSTOP** | **FULLCTI**]

Specifies the which method is used to implement the Stack mode of the on-chip trace.
NotAvailable: stack mode is not available for this on-chip trace.
TRGETM: the trigger delay counter of the onchip-trace is used. It starts by a trigger signal that must be provided by a trace source. Usually those events are routed through one or more CTIs to the on-chip trace.
FULLTIDRM: trigger mechanism for TI devices.
NOTSET: the method is derived by other GUIs or hardware. detection.
FULLSTOP: on-chip trace stack mode by implementation.
FULLCTI: on-chip trace provides a trigger signal that is routed back to on-chip trace over a CTI.

FUNNEL.PROGrammable [**ON** | **OFF**]

Default is ON. If set to ON the peripheral is controlled by TRACE32 in order to route ATB trace data through the ATB bus network. If PROGrammable is configured to value OFF then TRACE32 will not access the FUNNEL registers and the base address doesn't need to be configured. This can be useful for FUNNELs that don't have registers or when those registers are read-only. TRACE32 need still be aware of the connected ATB trace sources and sink in order to know the ATB topology. To build a complete topology across multiple instances of PowerView the property Name should be set at all instances to a chip wide unique identifier.

HTM.Type [**CoreSight** | **WPT**]

Selects the type of the AMBA AHB Trace Macrocell (HTM). CoreSight is the type as described in the ARM CoreSight manuals. WPT is a NXP proprietary trace module.

STM.Mode *<mode>*

Selects the protocol type used by the System Trace Module (STM).

STM.Type *<type>*

Selects the type of the System Trace Module (STM). Some types allow to work with different protocols.

TPIU.Type [**CoreSight** | **Generic**]

Selects the type of the Trace Port Interface Unit (TPIU).

CoreSight: A CoreSight TPIU, control register located at TPIU.Base *<address>* will be handled by the debugger.

Generic: A proprietary TPIU, control register will not be handled by the debugger.

Default: CoreSight.

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

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

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

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

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

HTM.Base *<address>*
HTM.RESet
HTM.Type [**CoreSight** | **WPT**]
AMBA AHB Trace Macrocell (HTM) - ARM CoreSight module
Trace source generating trace data of access to an AHB (AMBA High-performance Bus).

RTM.Base *<address>*
RTM.RESet
ReISC Trace Module (RTM)
A real time trace engine providing instructions and data tracing for the ReISC or STRed processor family.

STM.Base *<address>*
STM.Mode *<mode>*
STM.RESet
STM.Type *<type>*
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.Name *<string>*
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).

<parameters> which are "Deprecated"

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.

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.
ARPU: This mode is used for the ARPU CTI connection on MSR1 cores.

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

SYStem.CONFIG *<parameter>*

<parameter>: (Deprecated)	*<parameter>*: (New)
CTICONFIG *<type>*	**CTI.Config** *<type>*

<parameters> describing the STRED "Core"

STRED cores provide memory mapped registers for debug support. Since the memory map of these registers depends on the implementation, it is necessary to configure the debugger accordingly to debug a specific target system. Please note that normally the **SYStem.CPU** selection already configures proper default settings and using the commands described in this section shouldn't be necessary.

DSP.ProcessorID Configures the processor ID that is used by the target system to identify a specific processor within a multi-core system.

DSP.RESet Resets all SYStem.CONFIG.DSP settings to default values depending on the current **SYStem.CPU** selection.

DSP.state Displays the **SYStem.CONFIG.DSP.state** window to display and modify **SYStem.CONFIG.DSP** settings.

DSP.*<register>* *<addresses>* Configures the processor specific location of debug and system registers required for debugging.

In a Symmetric MultiProcessing (SMP) debug session, debug and system register addresses can be independent per core. Therefore one address per core should be specified or a single address if the register is common for all cores.

For a list of all configurable debug and system registers see **Configurable Debug and System Registers**.

Configurable Debug and System Registers

DbgIRQ DSP debug Interrupt request register.

DbgRESet DSP reset register.

DbgCLocK DSP clock control register.

DbgWakeUp DSP wakeup register.

DbgGPIO Base address of DSP general purpose debug registers.

BootPC Defines the Program Counter (PC) start value after a system reset.

DbgConFiG Base address of DSP configuration registers (used to configure e.g. trap handling or on-chip breakpoints).

DbgINTerrupt Base address of the debug interrupt configuration registers.

systemMISC System MISC register (used for interrupt masking).

DspNUMber DSP number (or Processor ID) register.

Format:	**SYStem.CONFIG.state** [**/***<tab>*]
<tab>:	**DebugPort** | **Jtag** | **AccessPorts** | **COmponents**

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger's operations.

Alternatively, you can modify the target configuration settings via the TRACE32 command line with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<tab>	Opens the **SYStem.CONFIG.state** window on the specified tab. For tab descriptions, see below.
DebugPort (default)	The **DebugPort** tab informs the debugger about the debug connector type and the communication protocol it shall use. For descriptions of the commands on the **DebugPort** tab, see **DebugPort**.
Jtag	The **Jtag** tab informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip. For descriptions of the commands on the **Jtag** tab, see **Jtag**.
AccessPorts	This tab informs the debugger about an Arm CoreSight Access Port (AP) and about how to control the AP to access chip-internal memory busses (AHB, APB, AXI) or chip-internal JTAG interfaces. For a descriptions of a corresponding commands, refer to **AP**.
COmponents	The **COmponents** tab informs the debugger (a) about the existence and interconnection of on-chip CoreSight debug and trace modules and (b) informs the debugger on which memory bus and at which base address the debugger can find the control registers of the modules. For descriptions of the commands on the **COmponents** tab, see **COmponents**.

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**STRED5 | STRED6 | MSR1-ARPU |** …

Default: STRED5.

Selects the processor type and, depending on the selected processor, configures default settings for the **SYStem.CONFIG** and **SYStem.Option** command groups.

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

SYStem.BdmClock (deprecated)

<*frequency*>: **10000.** … **40000000.**

Default frequency: 10 MHz.

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

<*frequency*>	• The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window. • Besides a decimal number like “100000.” short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.
RTCK	The debug clock is controlled by the RTCK signal (**R**eturned **TCK**).
ARTCK	Accelerated method to control the debug clock by the RTCK signal (Accelerated Returned TCK). This option is only relevant for JTAG debug ports.
CTCK	With this option higher debug port speeds can be reached. The TDO/SWDIO signal will be sampled by a signal which derives from TCK/SWCLK, but which is timely compensated regarding the debugger-internal driver propagation delays (**C**ompensation by **TCK**). This feature can be used with a debug cable version 3 or newer. If it is selected, although the debug cable is not suitable, a fixed frequency will be selected instead (minimum of 10 MHz and selected clock).
CRTCK	With this option higher debug port speeds can be reached. The TDO/SWDIO signal will be sampled by the RTCK signal. This compensates the debugger-internal driver propagation delays, the delays on the cable and on the target (**C**ompensation by **RTCK**). This feature requires that the target provides an RTCK signal. In contrast to the **RTCK** option, the TCK/SWCLK is always output with the selected, fixed frequency.

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

Default: OFF.

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

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

<mode>: **Enable**
Denied
StopAndGo

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

Enable (default)
CPU (deprecated)
Memory access during program execution to target is enabled.

Denied
Memory access during program execution to target is disabled.

StopAndGo
Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.

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

SYStem.Attach (alias for SYStem.Mode Attach)
SYStem.Down (alias for SYStem.Mode Down)
SYStem.Up (alias for SYStem.Mode Up)

<mode>:
Down
NoDebug
Prepare
Go
Attach
Up

Down (Default) — Disables the debug connection. The state of the DSP remains unchanged. The JTAG port is tristated.

NoDebug — Same as **Down**.

Prepare — Resets the DSP and establishes the debug connection. The debugger does not check the state of the core or system components. This can be used, for example, to set up clock configurations on the target before switching to **System.Mode Up**.

Go — Resets the DSP, establishes the debug connection and starts the user program immediately.

Attach — The debug connection is established without resetting the DSP. Select **Down** before connecting the debugger cable to the target and switch then to **Attach**.

Up — Resets the DSP and establishes the debug connection. The DSP is stopped in debug mode after the reset.

StandBy — Not available.

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

Default: 0

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

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

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

Default: OFF.

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

SYStem.Option.AXICACHEFLAGS Configure AXI-AP cache bits

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

Default: DeviceSYStem (=0x30: Domain=0x3, Cache=0x0).

This option configures the value used for the Cache and Domain bits in the Control Status Word (CSW[27:24]->Cache, CSW[14:13]->Domain) of an AXI Access Port of a DAP, when using the AXI: memory class.

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

Default: 0

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

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

Default: ON.

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

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

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

OFF Only for test purposes: Debug power is **not** requested and **not** checked by the debugger. The control bit is set to 0.

Use case:

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

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

SYStem.Option.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.

Format: **SYStem.Option.DAPREMAP** {*<address_range> <address>*}

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

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

SYStem.Option.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.

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

Configures the start address of the memory location reserved for the debug code. For more information about the debug code, see **"Debug Code"**, page 16.

SYStem.Option.DbgOvwr Allow debug code overwrite

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

Default: ON.

If enabled, the debugger will write the debug code to the target system memory on **SYStem.Up** or **SYStem.Attach**. For more information about the debug code, see **"Debug Code"**, page 16.

This option should be disabled if the debug code must not be written by the debugger, but is already available at the expected memory location (**SYStem.Option.DbgBase**) on the target system.

SYStem.Option.DbgTrap Allow trap handler address overwrite

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

Default: ON.

If enabled, the debugger will set the trap handler address to the debug code base address. For more information about the debug code, see **"Debug Code"**, page 16.

This option must be disabled if the code running on the target system should use it's own trap handler routine. In this case, this trap handler routine must take care that IBREAK, DBREAK and ILLEGAL_INSTRUCTION exceptions are forwarded to the debug code.

Format:	**SYStem.Option.DEBUGPORTOptions** *<option>*
<option>:	**SWITCHTOSWD.**[**TryAll** | **None** | **JtagToSwd** | **LuminaryJtagToSwd** | **DormantToSwd** | **JtagToDormantToSwd**] **SWDTRSTKEEP.**[**DEFault** | **LOW** | **HIGH**]

Default: SWITCHTOSWD.TryAll, SWDTRSTKEEP.DEFault.

See Arm CoreSight manuals to understand the used terms and abbreviations and what is going on here.

SWITCHTOSWD tells the debugger what to do in order to switch the debug port to serial wire mode:

TryAll	Try all switching methods in the order they are listed below. This is the default. Normally it does not hurt to try improper switching sequences. Therefore this succeeds in most cases.
None	There is no switching sequence required. The SW-DP is ready after power-up. The debug port of this device can only be used as SW-DP.
JtagToSwd	Switching procedure as it is required on SWJ-DP without a dormant state. The device is in JTAG mode after power-up.
LuminaryJtagToSwd	Switching procedure as it is required on devices from LuminaryMicro. The device is in JTAG mode after power-up.
DormantToSwd	Switching procedure which is required if the device starts up in dormant state. The device has a dormant state but does not support JTAG.
JtagToDormantToSwd	Switching procedure as it is required on SWJ-DP with a dormant state. The device is in JTAG mode after power-up.

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

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

SYStem.Option.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.EnTRST Control TAP reset

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

Default: ON.

To set the debug interface in a defined state the TAP is reset by driving the TRST pin low and additionally holding TMS low for five 5 TCKs. By setting the EnTRST option to OFF only the TMS method is used. The reason for introducing this command was that in some target systems several chips were connected to the TRST line, which must not be reset together with the debug TAP.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

If enabled, all interrupts will be masked during assembler single-step operations by use of the disable interrupt bit in the MISC system register. After the single step the register is restored to the original value. If the option is disabled, the MISC system register is not modified.

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

Default: OFF.

If enabled, all interrupts will be masked during HLL single-step operations by use of the disable interrupt bit in the MISC system register. After the single step the register is restored to the original value. If the option is disabled, the MISC system register is not modified.

CPU specific Benchmarking Commands

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

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

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

BMC.<counter>.MODE — Configure counter mode

Format: **BMC.<counter>.MODE** [**USER** | **SUPERUSER** | **ANY**]

Default: ANY.

Configures if the assigned event is counted while the core is in a specific operating mode.

USER	Event is counted only if the core is operating in user mode.
SUPERUSER	Event is counted only if the core is operating in super user mode.
ANY	Event is always counted.

TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

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

Format: **TrOnchip.RESet**

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

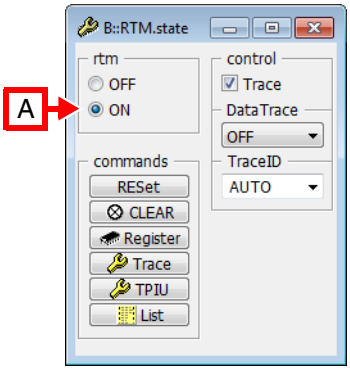
RTM — Trace source RTM

The RTM (ReISC Trace Module) is a multi-processor trace source providing real-time instruction and data tracing for STRED processors. A variety of trigger and filter settings are supported for flexible, selective tracing. The multi-processor design allows cross trace and trigger conditions on up to 16 processors.

The **RTM** command group is used to configure the trace source RTM. The **RTM** command group is locked if TRACE32 is not informed about the existence of a RTM trace source. For some chips, the RTM is a default setting, and the **RTM** command group is automatically activated after the chip is selected with the **SYStem.CPU** command. If the CPU selection does not include an RTM by default, TRACE32 can be manually informed about the start address of the RTM configuration space, as shown in the example below. Please refer to the manual of your chip to check if an RTM is available and to get the right start address.

```
; Use the CPU selection to get a default setup.
; An RTM might be already included.
SYStem.CPU <cpu>
IF COMPonent.AVAILABLE("RTM")==FALSE()
(
  ; Add RTM manually by defining the start address
  SYStem.CONFIG.RTM.Base <address>
)
```

For configuration of the trace source RTM, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **RTM.state** window.



A For descriptions of the commands in the **RTM.state** window, please refer to the **RTM.*** commands in this chapter. **Example**: For information about **ON**, see **RTM.ON**.

Exceptions:

- The **Trace** button opens the **Trace** window, see **<trace>.state**.
- The **TPIU** button opens the **TPIU** window, see **TPIU.state**.
- The **List** buttons opens the **Trace.List** window, see **<trace>.List**.

See also

- RTM.CLEAR
- RTM.CoreENable
- RTM.DataTrace
- RTM.OFF
- RTM.ON
- RTM.Register
- RTM.RESet
- RTM.state
- RTM.Trace
- RTM.TraceID

RTM.CLEAR Reset RTM registers to default values

Format: **RTM.CLEAR**

Resets RTM registers to their default values. This can be used as preparation for manual RTM configurations (see RTM.Trace OFF).

See also

- RTM
- RTM.state

RTM.CoreENable Select specific cores for RTM trace

Format: **RTM.CoreENable** <logical_core_numbers>

Enables or disables trace data generation for specific cores. Cores are identified by their logical core numbers assigned by TRACE32 as they are used by **CORE.select**. If no logical core number is specified, RTM trace is enabled for all cores.

Examples:

```
RTM.CoreENable 0.              ; enable RTM trace only for core 0

RTM.CoreENable 0. 2.           ; enable RTM trace for core 0 and core 2

RTM.CoreENable                 ; enable RTM trace for all cores
```

See also

- RTM
- RTM.state

Format:	**RTM.DataTrace** *<option>*
<option>:	**OFF** | **Address** | **Data** | **ON** | **OnlyAddress** | **OnlyData** | **Only**

Configures which elements are included in the data trace:

OFF	Trace program flow without data trace.
Address	Trace program flow and addresses of memory accesses.
Data	Trace program flow and data values of memory accesses.
ON	Trace program flow, addresses and data values of memory accesses.
OnlyAddress	Trace addresses of memory accesses without program flow.
OnlyData	Trace data values of memory accesses without program flow.
Only	Trace addresses and data values of memory accesses without program flow.

NOTE:	There is no correlation between addresses and data values of memory accesses in a RTM data trace.

See also

- RTM
- RTM.state

RTM.OFF Switch RTM off

Format:	**RTM.OFF**

Disables the RTM trace source functionality.

See also

- RTM
- RTM.state

Format: **RTM.ON**

Enables the RTM trace source functionality.

See also

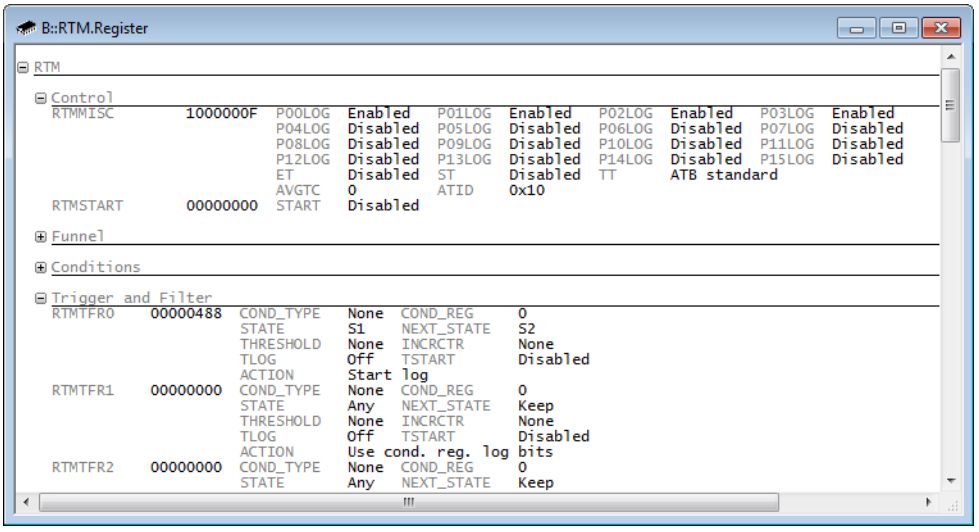
- RTM
- RTM.state

RTM.Register Display RTM registers

Format: **RTM.Register**

Displays RTM registers and the registers of trace-related modules.

Example:



See also

- RTM
- RTM.state

Format: **RTM.RESet**

Resets all settings of the **RTM.state** window to their default.

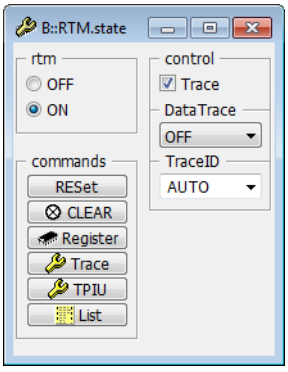
See also

- RTM
- RTM.state

RTM.state Display RTM settings

Format: **RTM.state**

Opens the **RTM.state** window to configure the trace source RTM and view the current configuration.



See also

- RTM
- RTM.CLEAR
- RTM.CoreENable
- RTM.DataTrace
- RTM.OFF
- RTM.ON
- RTM.Register
- RTM.RESet
- RTM.Trace
- RTM.TraceID

RTM.Trace Disable RTM configuration by the debugger

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

Default: ON

Controls if the RTM trace configuration is done by the debugger. If disabled, the debugger still modifies the RTM start bit and the RTM trace ID, but nothing else. Processor log bits, funnel port enable bits, conditions, filters and triggers must be configured manually.

ON	Enables RTM trace configuration by the debugger.
OFF	Disables RTM trace configuration by the debugger. RTM registers must be manually configured.

Disabling RTM trace configuration by the debugger is a feature intended for expert users, as it requires detailed knowledge about the RTM. Please refer to the RTM reference manual for more information.

See also

- RTM
- RTM.state

RTM.TraceID — Set RTM trace ID range

Format:	**RTM.TraceID** *<id>*
<id>:	**AUTO** | **0x10** | **0x20** | **0x30** | **0x40** | **0x50** | **0x60**

Default: AUTO.

Configures the trace ID range used by the RTM. Trace IDs are used to identify the trace source of a trace data packet within a common trace stream. Each core that is traced by the RTM is assigned a unique trace ID that equals the trace ID of the first core plus the index of the core (starting at 0).

AUTO	Allow TRACE32 to automatically select a trace ID range for the RTM.
0x10, 0x20, 0x30, 0x40, 0x50, 0x60	Manually select the trace ID range used by the RTM. The specified value is the trace ID used for trace data from the first processor connected to the RTM. The trace ID must be a multiple of 16 in the range from 16 (0x10) to 96 (0x60). The RTM uses a unique trace ID for each connected processor in ascending order.

See also

- RTM
- RTM.state

Target Adaption

The STRED is usually debugged and traced as part of an Arm/Cortex multi-core chip. In this case the adaptation is defined by the Arm/Cortex and its target adaptation.

Target Adaption for ARM

Probe Cables

For debugging, two kinds of probe cables can be used to connect the debugger to the target:

- CombiProbe

For an off-chip core trace, an additional trace probe cable “Preprocessor” is needed.

Interface Standards JTAG, Serial Wire Debug, cJTAG

Debug Cable and CombiProbe support the following interface standards:

- JTAG (IEEE 1149.1)
- Serial Wire Debug (CoreSight ARM)
- Compact JTAG (IEEE 1149.7, cJTAG)

The different interface standards are supported by the same connector. Only some signals get a different function. The interface standard to be used can be selected with the TRACE32 commands listed below. This assumes of course that your target supports the selected interface standard.

Interface	TRACE32 Commands
JTAG	The JTAG interface is activated with **SYStem.CONFIG DEBUGPORTTYPE JTAG**.
Serial Wire Debug	The Serial Wire Debug interface is activated with **SYStem.CONFIG DEBUGPORTTYPE SWD**. In a multidrop configuration, you need to specify the address of your debug client with **SYStem.CONFIG SWDPTARGETSEL**.
Compact JTAG (cJTAG)	Compact JTAG (cJTAG) interface is activated with **SYStem.CONFIG DEBUGPORTTYPE CJTAG**. Your system might need bug fixes which can be activated with **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).

Debug Cable

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

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

CombiProbe

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

The CombiProbe will always be delivered with 10-pin, 20-pin, 34-pin connectors. The CombiProbe cannot detect which one is used. If you use the trace of the CombiProbe, you need to inform TRACE32 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 cable you want to use with **SYStem.CONFIG DEBUGPORT DebugCableA | DebugCableB**. The CombiProbe can detect the location of the cable if only one is connected.

Preprocessor

Adaption for ARM ETM Preprocessor Mictor: See **https://www.lauterbach.com/adetmmictor.html**.

Adaption for ARM ETM Preprocessor MIPI-60: See **https://www.lauterbach.com/adetmmipi60.html**.

Adaption for ARM ETM Preprocessor HSSTP: See **https://www.lauterbach.com/adetmhsstp.html**.