LAUTERBACH
DEVELOPMENT TOOLS

CEVA-X Debugger and Trace

MANUAL

CEVA-X Debugger and Trace

CEVA-X Debugger and Trace

Version 06-Jun-2024

Introduction

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.

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known CEVA-X 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/cevax/` 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. Select the device prompt for the ICD Debugger and reset the system.

```
B::

RESet
```

The device prompt `B::` is normally already selected in the 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.

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

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

3. Set the JTAG frequency

```
SYStem.JtagClock <frequency>
```

The default value is 1.0 MHz.

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

```
MAP.BOnchip <range>
```

The B(reak)Onchip information is necessary to decide where on-chip breakpoints must be used. On-chip breakpoints are necessary to set program breakpoints to FLASH/ROM. The sections of FLASH and ROM depend on the specific CPU and its chip selects.

5. Enter debug mode.

```
SYStem.Up
```

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

6. Load the program.

```
Data.LOAD.COFF program.a     ; COFF specifies the format, program.a
                             ; is the file name)
```

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

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

A typical start sequence for the TeakLiteDev-C is shown below. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** *<file>*. Other sequences can be found in the ~~/demo/ directory.

```
B::                           ; Select the ICD device prompt

WinClear                      ; Clear all windows

SYS.CPU TeakLiteDev-C         ; Select CPU

SYStem.JtagClock 10MHz        ; Choose JTAG frequency

SYStem.UP                     ; Reset the target and enter debug mode

Data.LOAD.COFF demo.a         ; Load the application with option large
                              ; memory model and verify the process

Register.Set PC start         ; Set program counter

List.Mix                      ; Open source code window          *)

Go main                       ; Run and break at main()

Register.view /SpotLight      ; Open register window             *)

Var.Local                     ; Open window with local variables *)
```

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

Troubleshooting

No information available

FAQ

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

CPU Specific Implementations

Breakpoints

There are two types of breakpoints available: Software breakpoints and on-chip breakpoints.

Software Breakpoints

Software breakpoints are the default breakpoints for program breakpoints. A software breakpoint is implemented by patching a break code into the memory.

There is no restriction in the number of software breakpoints.

On-chip Breakpoints

The resources for the on-chip breakpoints are provided by the CPU.

The following list gives an overview of the supported on-chip breakpoints:

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

Family	Onchip Breakpoints	Program Breakpoints	Read/Write Breakpoints	Data Value Breakpoints
CEVA-X	4 instruction 4 read/write	4 single address	4 single address or range	2

Disassembler

Starting with CEVA-ToolBox v18, TRACE32 must be configured to use the disassembler libraries provided by Ceva. The only exception from this rule is the NeuPro (Ceva-XM6), which can optionally still use the build-in disassembler of TRACE32.

MS Windows

1. Install CEVA-ToolBox for your CPU.
2. Browse to <CEVA-ToolBox>\<version>\<cpu>\cevatools\bin.
3. Copy the following two files to <TRACE32>\bin\windows64:
   - cevaxasmsrv.dll
   - <cpu>db.dll
4. Add the following line to your *.cmm script:

```
apu.load ~~/bin/windows64/t32cevadislink.dll "cevaxasmsrv.dll" "<cpu>"
```

Linux

1. Install CEVA-ToolBox for your CPU.
2. Optional: Browse to <CEVA-ToolBox>/<version>/<cpu>/cevatools/bin.
3. Optional: Copy the following two files to <TRACE32>/bin/pc_linux64:
   - libcevaxasmsrv.so
   - lib<cpu>db.so
4. Add the path of the disassembler libraries (step 2 or step 3) to LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=<path_to_libraries>
```

5. Add the following line to your *.cmm script:

```
apu.load ~~/bin/pc_linux64/t32cevadislink.so "libcevaxasmsrv.so" "<cpu>"
```

SYStem.CONFIG.state

Display target configuration

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.CONFIG** *<parameter>*
	SYStem.MultiCore *<parameter>* (deprecated)
<parameter>:	**CONNECTOR** [**MIPI34** | **MIPI20T**]
(DebugPort)	**CORE** *<core> <chip>*
	DEBUGPORT [**DebugCable0** | **DebugCableA** | **DebugCableB**]
	DEBUGPORTTYPE [**JTAG** | **SWD** | **CJTAG**]
	Slave [**ON** | **OFF**]
	SWDP [**ON** | **OFF**]
	SWDPIDLEHIGH [**ON** | **OFF**]
	SWDPTargetSel *<value>*
	TriState [**ON** | **OFF**]
<parameter>:	**DAPDRPOST** *<bits>*
(JTAG)	**DAPDRPRE** *<bits>*
	DAPIRPOST *<bits>*
	DAPIRPRE *<bits>*
	DRPOST *<bits>*
	DRPRE *<bits>*
	IRPOST*<bits>*
	IRPRE *<bits>*
	Slave [**ON** | **OFF**]
	TAPState *<state>*
	TCKLevel *<level>*
	TriState [**ON** | **OFF**]
<parameter>:	**CFGCONNECT** *<code>*
(MultiTap)	**MULTITAP** [**NONE** | **IcepickA** | **IcepickB** | **IcepickC** | **IcepickD** | **IcepickBB** | **IcepickBC** | **IcepickCC** | **IcepickDD** | **STCLTAP1** | **STCLTAP2** | **STCLTAP3** | **MSMTAP** *<irlength> <irvalue> <drlength> <drvalue>* **JtagSEQuence** *<sub_cmd>*]
<parameter>:	**AHBAPn.Base** *<address>*
(AccessPorts)	**AHBAPn.HPROT** [*<value>* | *<name>*]
	AHBAPn.Port *<port>*
	AHBAPn.RESet
	AHBAPn.view
	AHBAPn.XtorName *<name>*
	APBAPn.Base *<address>*
	APBAPn.Port *<port>*

<parameter>: (AccessPorts cont.)	**APBAPn.RESet** **APBAPn.view** **APBAPn.XtorName** *<name>* **AXIAPn.ACEEnable** [**ON** | **OFF**] **AXIAPn.Base** *<address>* **AXIAPn.CacheFlags** *<value>* **AXIAPn.HPROT** [*<value>* | *<name>*] **AXIAPn.Port** *<port>* **AXIAPn.RESet** **AXIAPn.view** **AXIAPn.XtorName** *<name>* **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.HPROT** [*<value>* | *<name>*] **MEMORYAPn.Port** *<port>* **MEMORYAPn.RESet** **MEMORYAPn.view** **MEMORYAPn.XtorName** *<name>*
<parameter>: **(COmponents)**	**COREDEBUG.Base** *<address>* **COREDEBUG.RESet** **COREDEBUG.view** **CTI.Base** *<address>* **CTI.Config** [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**] **CTI.RESet** **CTI.view** **ETB.ATBSource** *<source>* **ETB.Base** *<address>* **ETB.Name** *<string>* **ETB.NoFlush** [**ON** | **OFF**] **ETB.RESet** **ETB.Size** *<size>* **ETB.STackMode [NotAvailbale | TRGETM | FULLTIDRM | NOTSET | FULLSTOP | FULLCTI]** **ETB.view**

<parameter>:
(COmponents cont.)

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

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

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

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

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

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

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

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

©198

<parameter>:
(Deprecated)

COREBASE *<address>*
CTIBASE *<address>*
CTICONFIG [**NONE** | **ARMV1** | **ARMPostInit** | **OMAP3** | **TMS570** | **CortexV1** | **QV1**]
DEBUGBASE *<address>*
ETBBASE *<address>*
ETBFUNNELBASE *<address>*
ETFBASE *<address>*
ETMBASE *<address>*
ETMETBFUNNELPORT *<port>*
ETMFUNNEL2PORT *<port>*
ETMFUNNELPORT *<port>*
ETMTPIUFUNNELPORT *<port>*
FUNNEL2BASE *<address>*
FUNNELBASE *<address>*
HTMBASE *<address>*
HTMETBFUNNELPORT *<port>*
HTMFUNNEL2PORT *<port>*
HTMFUNNELPORT *<port>*
HTMTPIUFUNNELPORT *<port>*
TPIUBASE *<address>*
TPIUFUNNELBASE *<address>*
view

AHBACCESSPORT *<port>*
APBACCESSPORT *<port>*
AXIACCESSPORT *<port>*
COREJTAGPORT *<port>*
DEBUGACCESSPORT *<port>*
JTAGACCESSPORT *<port>*
MEMORYACCESSPORT *<port>*

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

Some commands need a certain CPU type selection (**SYStem.CPU** *<type>*) to become active and might additionally depend on further settings.

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

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

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.
CORE *<core> <chip>*	The command helps to identify debug and trace resources which are commonly used by different cores. The command might be required in a multicore environment if you use multiple debugger instances (multiple TRACE32 PowerView GUIs) to simultaneously debug different cores on the same target system. Because of the default setting of this command debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=1 *<chip>*=2 ... each debugger instance assumes that all notified debug and trace resources can exclusively be used. But some target systems have shared resources for different cores, for example a common trace port. The default setting causes that each debugger instance controls the same trace port. Sometimes it does not hurt if such a module is controlled twice. But sometimes it is a must to tell the debugger that these cores share resources on the same *<chip>*. Whereby the "chip" does not need to be identical with the device on your target board: debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=2 *<chip>*=1
CORE *<core> <chip>* (cont.)	For cores on the same *<chip>,* the debugger assumes that the cores share the same resource if the control registers of the resource have the same address. Default: *<core>* depends on CPU selection, usually 1. *<chip>* 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,...).
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.
Slave [**ON** | **OFF**]	If several debuggers share the same debug port, all except one must have this option active. JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave ON**. Default: OFF. Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).
SWDPIdleHigh [**ON** | **OFF**]	Keep SWDIO line high when idle. Only for Serialwire Debug mode. Usually the debugger will pull the SWDIO data line low, when no operation is in progress, so while the clock on the SWCLK line is stopped (kept low). You can configure the debugger to pull the SWDIO data line high, when no operation is in progress by using **SYStem.CONFIG SWDPIdleHigh ON** Default: OFF.
SWDPTargetSel *<value>*	Device address in case of a multidrop serial wire debug port. Default: none set (any address accepted).
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

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

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

Multiple TAPs can be controlled by one JTAG interface by daisy-chaining the TAPs (serial connection). If you want to talk to one TAP in the chain, you need to send a BYPASS pattern (all ones) to all other TAPs. For this case the debugger needs to know the position of the TAP it wants to talk to. The TAP position can be defined with the first four commands in the table below.

… **DRPOST** *<bits>*	Defines the TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TDI signal and the TAP you are describing. In BYPASS mode, each TAP contributes one data register bit. See possible TAP types and example below. Default: 0.
… **DRPRE** *<bits>*	Defines the TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TAP you are describing and the TDO signal. In BYPASS mode, each TAP contributes one data register bit. See possible TAP types and example below. Default: 0.
… **IRPOST** *<bits>*	Defines the TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between TDI signal and the TAP you are describing. See possible TAP types and example below. Default: 0.
… **IRPRE** *<bits>*	Defines the TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the TAP you are describing and the TDO signal. See possible TAP types and example below. Default: 0.

NOTE:	If you are not sure about your settings concerning **IRPRE**, **IRPOST**, **DRPRE**, and **DRPOST**, you can try to detect the settings automatically with the **SYStem.DETECT.DaisyChain** command.

Slave [**ON** | **OFF**]	If several debuggers share the same debug port, all except one must have this option active. JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**. Default: OFF. Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).
TAPState *<state>*	This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable. 0 Exit2-DR 1 Exit1-DR 2 Shift-DR 3 Pause-DR 4 Select-IR-Scan 5 Update-DR 6 Capture-DR 7 Select-DR-Scan 8 Exit2-IR 9 Exit1-IR 10 Shift-IR 11 Pause-IR 12 Run-Test/Idle 13 Update-IR 14 Capture-IR 15 Test-Logic-Reset Default: 7 = Select-DR-Scan.
TCKLevel *<level>*	Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target. Default: 0.
TriState [**ON** | **OFF**]	TriState has to be used if several debug cables are connected to a common JTAG port. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: • nTRST must have a pull-up resistor on the target. • TCK can have a pull-up or pull-down resistor. • Other trigger inputs need to be kept in inactive state. Default: OFF.

TAP types:

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

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

<parameters> describing a system level TAP "MultiTap"

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

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.
MULTITAP [**NONE** | **IcepickA** | **IcepickB** | **IcepickC** | **IcepickD** | **IcepickM** | **IcepickBB** | **IcepickBC** | **IcepickCC** | **IcepickDD** | **STCLTAP1** | **STCLTAP2** | **STCLTAP3** | **MSMTAP** *<irlength> <irvalue> <drlength> <drvalue>* **JtagSEQuence** *<sub_cmd>*]	Selects the type and version of the MULTITAP. In case of MSMTAP you need to add parameters which specify which IR pattern and DR pattern needed to be shifted by the debugger to initialize the MSMTAP. Please note some of these parameters need a decimal input (dot at the end). IcepickXY means that there is an Icepick version "X" which includes a subsystem with an Icepick of version "Y". For a description of the **JtagSEQuence** subcommands, see **SYStem.CONFIG.MULTITAP JtagSEQuence**.

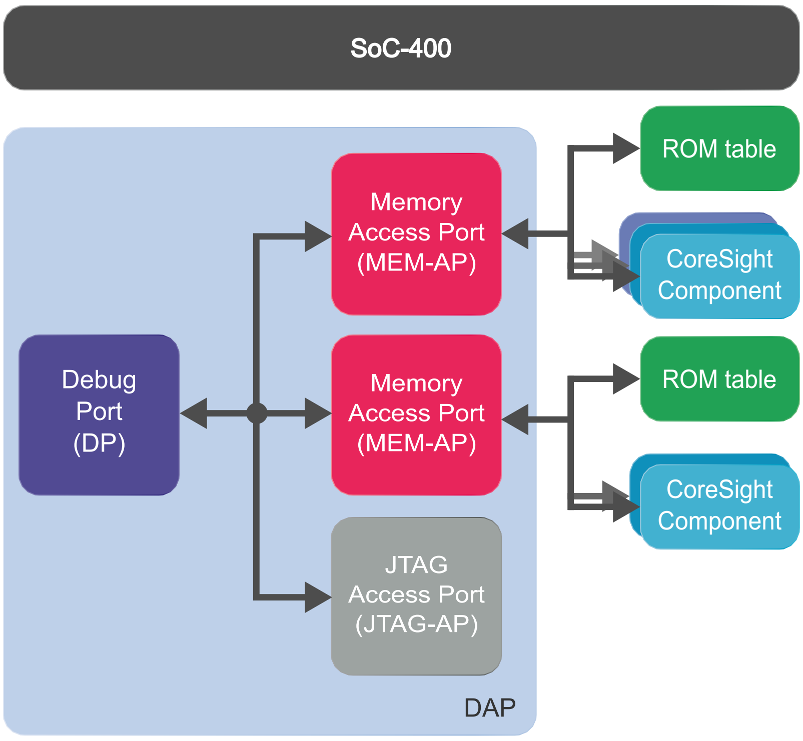
An Access Port (AP) is a CoreSight module from ARM which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

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:, “DAP”, “E:”. The interface to these buses is called Memory Access Port (MEM-AP).

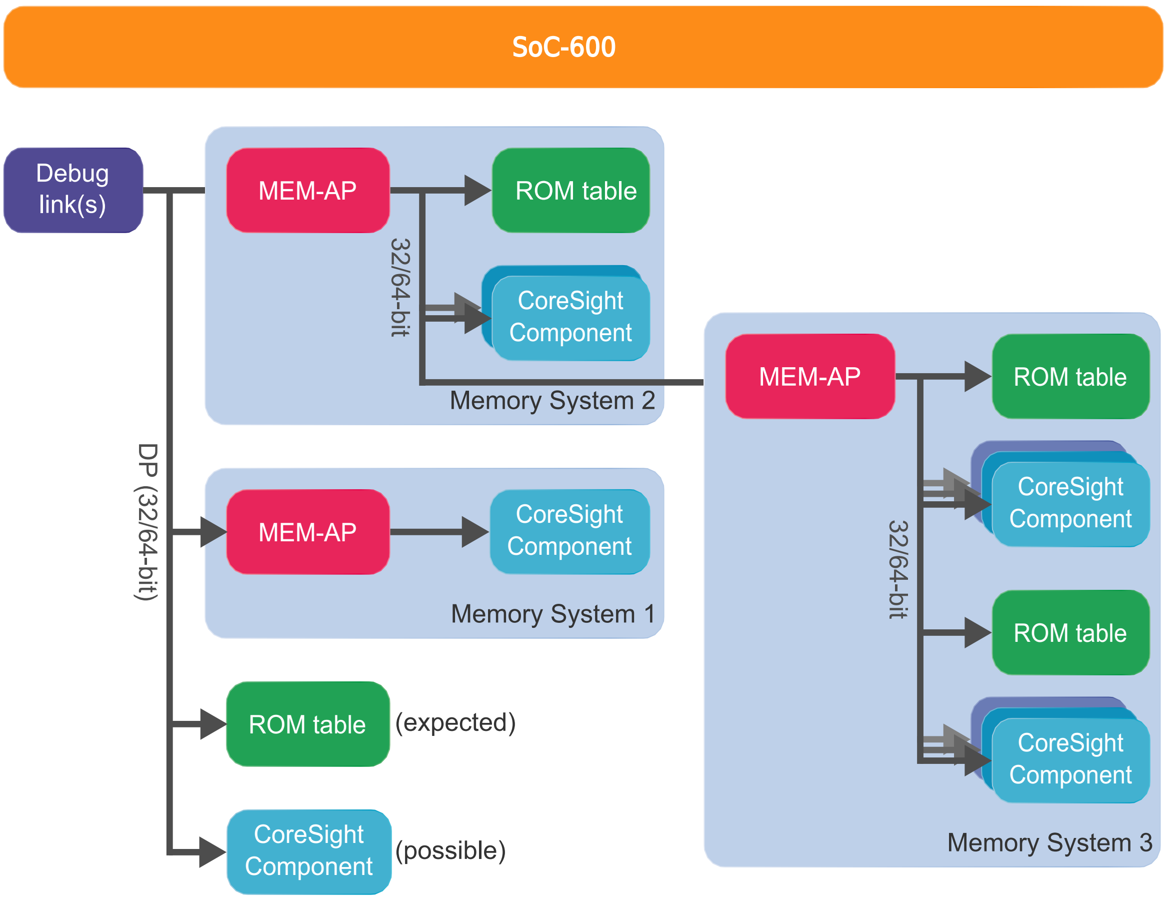
2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number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

3. 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 [*<value>* | *<name>*]	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 [*<value>* | *<name>*]	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 [*<value>* | *<name>*]	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**]	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>*	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.

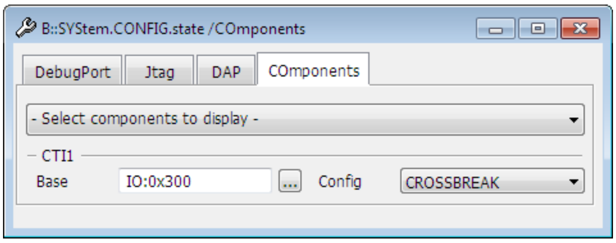
AHBAPn.Port *<port>* AHBACCESSPORT *<port>* (deprecated)	Access Port Number (0-255) of a SoC-400 system which shall be used for "AHBn:" access class. Default: *<port>*=0.
APBAPn.Port *<port>* APBACCESSPORT *<port>* (deprecated)	Access Port Number (0-255) of a SoC-400 system which shall be used for "APBn:" access class. Default: *<port>*=1.
AXIAPn.Port *<port>* AXIACCESSPORT *<port>* (deprecated)	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>* DEBUGACCESSPORT *<port>* (deprecated)	AP access port number (0-255) of a SoC-400 system where the debug register can be found (typically on APB). Used for "DAP:" access class. Default: *<port>*=1.
JTAGAPn.CorePort *<port>* COREJTAGPORT *<port>* (deprecated)	JTAG-AP port number (0-7) connected to the core which shall be debugged.
JTAGAPn.Port *<port>* JTAGACCESSPORT *<port>* (deprecated)	Access port number (0-255) of a SoC-400 system of the JTAG Access Port.
MEMORYAPn.Port *<port>* MEMORYACCESSPORT *<port>* (deprecated)	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.

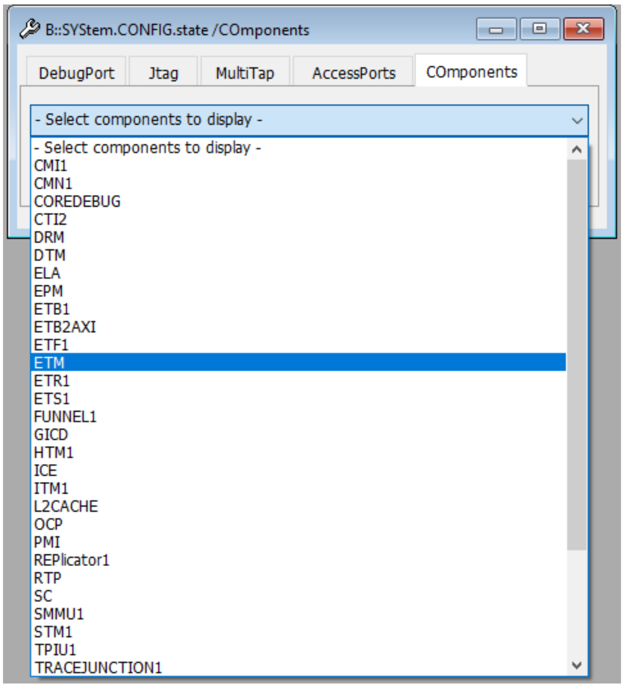
SoC-600 Specific Commands

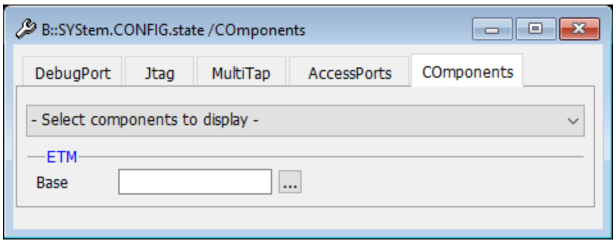
AHBAPn.Base *<address>*	This command informs the debugger about the start address of the register block of the “AHBAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write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

<parameters> describing debug and trace "Components"

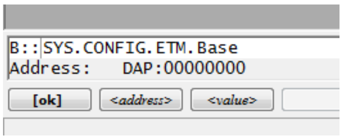
On the **Components** tab in the **SYStem.CONFIG.state** window, you can comfortably add the debug and trace components your chip includes and which you intend to use with the debugger's help.


B::SYStem.CONFIG.state /COmponents
DebugPort
Jtag
DAP
COmponents
- Select components to display -
CTI1
Base
IO:0x300
...
Config
CROSSBREAK




B::SYStem.CONFIG.state /COmponents
DebugPort
Jtag
MultiTap
AccessPorts
COmponents
- Select components to display -
ETM
Base
...

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.

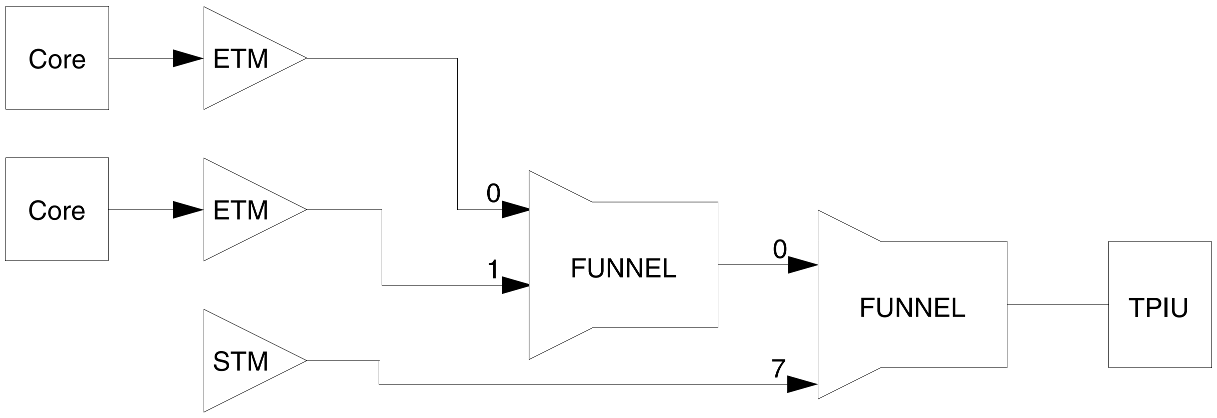


```
B::SYS.CONFIG.ETM.Base
Address:    DAP:00000000
```

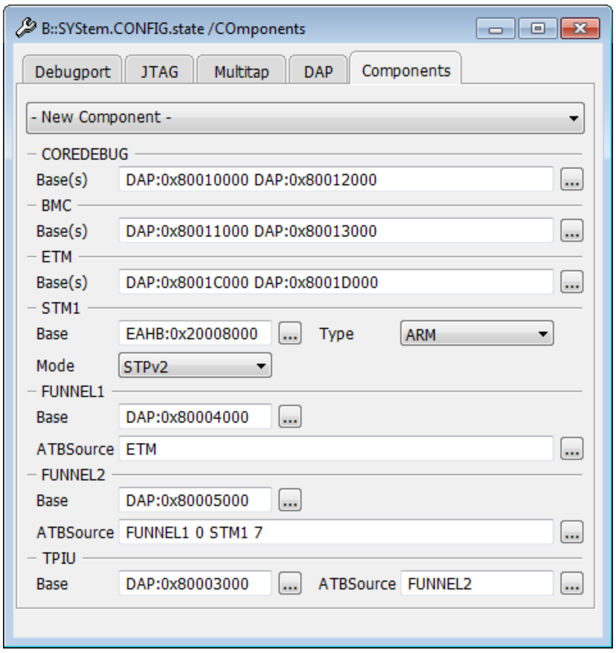
You can have several of the following components: CMI, ETB, ETF, ETR, FUNNEL, STM.
Example: FUNNEL1, FUNNEL2, FUNNEL3,...

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.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
```



- New Component -

COREDEBUG
Base(s) DAP:0x80010000 DAP:0x80012000

BMC
Base(s) DAP:0x80011000 DAP:0x80013000

ETM
Base(s) DAP:0x8001C000 DAP:0x8001D000

STM1
Base EAHB:0x20008000 Type ARM
Mode STPv2

FUNNEL1
Base DAP:0x80004000
ATBSource ETM

FUNNEL2
Base DAP:0x80005000
ATBSource FUNNEL1 0 STM1 7

TPIU
Base DAP:0x80003000 ATBSource FUNNEL2

… **.ATBSource** *<source>*	Specify for components collecting trace information from where the trace data are coming from. This way you inform the debugger about the interconnection of different trace components on a common trace bus. You need to specify the "... .Base *<address>*" or other attributes that define the amount of existing peripheral modules before you can describe the interconnection by "... .ATBSource *<source>*". A CoreSight trace FUNNEL has eight input ports (port 0-7) to combine the data of various trace sources to a common trace stream. Therefore you can enter instead of a single source a list of sources and input port numbers. **Example**: SYStem.CONFIG FUNNEL.ATBSource ETM 0 HTM 1 STM 7 Meaning: The funnel gets trace data from ETM on port 0, from HTM on port 1 and from STM on port 7. In an SMP (Symmetric MultiProcessing) debug session where you used a list of base addresses to specify one component per core you need to indicate which component in the list is meant:

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

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

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

… **.BASE** *<address>*

This command informs the debugger 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 COREBEBUG, CTI, ETB, ETF, ETM, ETR a list of base addresses to specify one component per core.

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

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

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

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

CTI.Config *<type>*

Informs about the interconnection of the core Cross Trigger Interfaces (CTI). Certain ways of interconnection are common and these are supported by the debugger e.g. to cause a synchronous halt of multiple cores.

NONE: The CTI is not used by the debugger.
ARMV1: This mode is used for ARM7/9/11 cores which support synchronous halt, only.
ARMPostInit: Like ARMV1 but the CTI connection differs from the ARM recommendation.
OMAP3: This mode is not yet used.
TMS570: Used for a certain CTI connection used on a TMS570 derivative.
CortexV1: The CTI will be configured for synchronous start and stop via CTI. It assumes the connection of DBGRQ, DBGACK, DBGRESTART signals to CTI are done as recommended by ARM. The CTIBASE must be notified. “CortexV1” is the default value if a Cortex-A/R core is selected and the CTIBASE is notified.
QV1: This mode is not yet used.

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.

ETR.CATUBase *<address>*

Base address of the CoreSight Address Translation Unit (CATU).

FUNNEL.Name *<string>*

It is possible that different funnels have the same address for their control register block. This assumes they are on different buses and for different cores. In this case it is needed to give the funnel different names to differentiate them.

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

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

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

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

STM.Mode [**NONE** | **XTIv2** | **SDTI** | **STP** | **STP64** | **STPv2**]	Selects the protocol type used by the System Trace Module (STM).
STM.Type [**None** | **Generic** | **ARM** | **SDTI** | **TI**]	Selects the type of the System Trace Module (STM). Some types allow to work with different protocols (see STM.Mode).
TPIU.Type [**CoreSight** | **Generic**]	Selects the type of the Trace Port Interface Unit (TPIU). CoreSight: Default. CoreSight TPIU. TPIU control register located at TPIU.Base *<address>* will be handled by the debugger. Generic: Proprietary TPIU. TPIU control register will not be handled by the debugger.

Components and Available Commands

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

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.

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

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

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

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

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

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

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

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.

… **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 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”. For a list of possible components including a short description see **Components and Available Commands**.
… **PORT** *<port>*	Informs the debugger about which trace source is connected to which input port of which funnel. A CoreSight trace funnel provides 8 input ports (port 0-7) to combine the data of various trace sources to a common trace stream. **Example**: SYStem.CONFIG STMFUNNEL2PORT 3 Meaning: The System Trace Module (STM) is connected to input port #3 on FUNNEL2. On an SMP debug session some of these commands can have a list of *<port>* parameter. In case there are dedicated funnels for the ETB and the TPIU their base addresses are specified by ETBFUNNELBASE, TPIUFUNNELBASE respectively. And the funnel port number for the ETM are declared by ETMETBFUNNELPORT, ETMTPIUFUNNELPORT respectively. For a list of possible components including a short description see **Components and Available Commands**.

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.
view	Opens a window showing most of the SYStem.CONFIG settings and allows to modify them.

Deprecated and New Commands

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

SYStem.CONFIG *<parameter>*

<parameter>: (Deprecated)	*<parameter>*: (New)
COREBASE *<address>*	**COREDEBUG.Base** *<address>*
CTIBASE *<address>*	**CTI.Base** *<address>*
CTICONFIG *<type>*	**CTI.Config** *<type>*
DEBUGBASE *<address>*	**COREDEBUG.Base** *<address>*
ETBBASE *<address>*	**ETB1.Base** *<address>*
ETBFUNNELBASE *<address>*	**FUNNEL4.Base** *<address>*
ETFBASE *<address>*	**ETF1.Base** *<address>*
ETMBASE *<address>*	**ETM.Base** *<address>*
ETMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource ETM** *<port>* (1)
ETMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource ETM** *<port>* (1)

ETMFUNNELPORT *<port>*	**FUNNEL1.ATBSource ETM** *<port>* (1)
ETMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource ETM** *<port>* (1)
FUNNEL2BASE *<address>*	**FUNNEL2.Base** *<address>*
FUNNELBASE *<address>*	**FUNNEL1.Base** *<address>*
HSMBASE *<address>*	**HSM.Base** *<address>*
HTMBASE *<address>*	**HTM.Base** *<address>*
HTMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource HTM** *<port>* (1)
HTMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource HTM** *<port>* (1)
HTMFUNNELPORT *<port>*	**FUNNEL1.ATBSource HTM** *<port>* (1)
HTMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource HTM** *<port>* (1)
TPIUBASE *<address>*	**TPIU.Base** *<address>*
TPIUFUNNELBASE *<address>*	**FUNNEL3.Base** *<address>*
view	**state**

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

SYStem.CONFIG.EXTMEM External program memory

Format:	**SYStem.CONFIG.EXTMEM.***<sub_cmd>* …
<sub_cmd>:	**Base** *<address>*| **L2A** *<attribute>* | **QOS** *<value>* | **RESet**

Base	Defines the base address of common external program and data memory.
L2A	Level 2 cache attributes.
QOS	Quality of Service attributes.
RESet	Resets EXTMEM settings.

SYStem.CPU
Select the used CPU

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**OAK,PMB8870P** | **PMB8870S** (OAK cores)
	TeakLiteDev-A | **TeakLiteDev-B** | **TeakLiteDev-C** | **PMB8875** | **88i6523** (TeakLite cores)
	TEAK-REVA | **TEAK-RTL2_0** | **TEAK_REVB** | **XPERTTEAK** (Teak cores)

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

SYStem.JtagClock
Define JTAG clock

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

Default: 1 MHz.

Selects the frequency for the debug interface.

SYStem.LOCK
Lock and tristate the debug port

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.

SYStem.MemAccess — Select run-time memory access method

Format:	**SYStem.MemAccess Enable | StopAndGo | Denied** **SYStem.ACCESS** (deprecated)

Enable **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.

SYStem.Mode — Establish the communication with the target

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** **Up** **Attach**

Down	Disables the debugger (default). The state of the CPU remains unchanged. The JTAG port is tristated.
Up	Reset the target, sets the CPU to debug mode and stops the CPU.
Attach	No reset happens, the mode of the core (running or halted) does not change. The debug port will be initialized. After this command has been executed, the user program can, for example, be stopped with the **Break** command.
StandBy **NoDebug** **Go**	Not available for CEVA-X.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step. For 56800E processors IMASKASM ON is necessary for HLL stepping and stepping from software breakpoints.

SYStem.Option.IMASKHLL   Disable interrupts while HLL single stepping

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 single step the interrupt mask bits are restored to the value before the step.

SYStem.Option.OVERLAY   Enable overlay support

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

Default: OFF.

ON	Activates the overlay extension and extends the address scheme of the debugger with a 16 bit virtual overlay ID. Addresses therefore have the format *<overlay_id>***:***<address>*. This enables the debugger to handle overlaid program memory.
OFF	Disables support for code overlays.
WithOVS	Like option **ON**, but also enables support for software breakpoints. This means that TRACE32 writes software breakpoint opcodes to both, the *execution area* (for active overlays) and the *storage area*. This way, it is possible to set breakpoints into inactive overlays. Upon activation of the overlay, the target's runtime mechanisms copies the breakpoint opcodes to the execution area. For using this option, the storage area must be readable and writable for the debugger.

Example:

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

SYStem.Option.PB0size   Setup size of internal program memory

Format:	**SYStem.Option.PB0size** *<size>*
<size>	**32KB** | **64KB** | **128KB** | **256KB** | **512KB** | **1024KB**

Sets the size of the internal program memory (PTCM), usually detected automatically.

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.VCU.INSTances Number of available VCUs

Format:	**SYStem.VCU.INSTances** <*count*>

Default: 0

Specifies the number of implemented Vector Computation Units (VCU) of the SOC.

SYStem.VCU.MLD MLD available or not

Format:	**SYStem.VCU.MLD** [**ON** | **OFF**]

Default: OFF

Defines whether the VCU instance(s) features a Maximum-Likelyhood-Decoder (ON) or not (OFF).

General Restrictions

Setting the PC	In cases where the program counter consists of the PC register and program page extension bits, the program counter can be set by the register PP.

CEVA-X specific ETM Command

ETM.BranchBBC

Control branch BBC mode

Format:	**ETM.BranchBBC** [**ON** | **OFF**]

Controls branch BBC mode.

ETM.IgnoreISyncPredicate

Ignore I-Sync predicates

Format:	**ETM.IgnoreISyncPredicate** [**ON** | **OFF**]

Default: OFF.

Workaround in ETM trace decoder for bad predicates at I-Sync.

ETM.LoopBBC

Branch broadcast

Format:	**ETM.LoopBBC** [**ON** | **OFF**]

Enables/disables branch-broadcasting globally.

ETM.PredicateAddress

Set predicate address

Format:	**ETM.PredicateAddress** [*<address>*]

Configures predicate address.

ETM.PredicatePeriod — Predicated counter in ETM wrapper

Format: **ETM.PredicatePeriod** [*<cycles>*]

This command sets up predicated counter in ETM wrapper.

ETM.TimeStampInjectorTraceID — CoreSight ATB ID

Format: **ETM.TimeStampInjectorTraceID** *<id_tsi0> <id_tsi1>* **...**

Some SoCs feature a global timestamp unit, which allows a correlation of multiple core traces in time. In order to do that, each core in turn is a associated with a so-called timestamp injector. The timestamps from the timestamp injectors are not directly injected into the core trace, but have their own CoreSight ATB IDs.

ETM.WrapperFilter — Global breakpoint enable

Format: **ETM.WrapperFilter** [**ON** | **OFF**]

Disables or enabled all data and program breakpoints at once.

Default: ON.

ETM.WrapperSTALL — Enable/disable wrapper stall

Format: **ETM.WrapperSTALL** [**ON** | **OFF**]

Default: ON.

TrOnchip Commands

The OCEM registers can be used to break on several conditions.

TrOnchip.CONVert — Adjust range breakpoint in on-chip resource

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

The on-chip breakpoints can only cover specific ranges. If a range cannot be programmed into the breakpoint, it will automatically be converted into a single address breakpoint when this option is active. This is the default. Otherwise an error message is generated.

```
TrOnchip.CONVert ON
Break.Set 0x1000--0x17ff /Write        ; sets breakpoint at range
Break.Set 0x1001--0x17ff /Write        ; 1000--17ff sets single breakpoint
…                                      ; at address 1001

TrOnchip.CONVert OFF                   ; sets breakpoint at range
Break.Set 0x1000--0x17ff /Write        ; 1000--17ff
Break.Set 0x1001--0x17ff /Write        ; gives an error message
```

TrOnchip.VarCONVert — Adjust complex breakpoint in on-chip resource

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

The on-chip breakpoints can only cover specific ranges. If you want to set a marker or breakpoint to a complex variable, the on-chip break resources of the CPU may be not powerful enough to cover the whole structure. If the option **TrOnchip.VarCONVert** is set to **ON**, the breakpoint will automatically be converted into a single address breakpoint. This is the default setting. Otherwise an error message is generated.

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

Format:	**TrOnchip.RESet**

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

TrOnchip.Set — Set breakpoint

Format:	**TrOnchip.Set.***<trigger>* [**ON** | **OFF**]
<trigger>:	**EXT1**
	EXT2
	EXT3
	EXT4

Sets a trigger condition which causes the core to stop. Not all events listed below may be available on all CPUs.

EXT1	Trigger on external input #1.
EXT2	Trigger on external input #2.
EXT3	Trigger on external input #3.
EXT4	Trigger on external input #4.

TrOnchip.state — Display “Trigger-Onchip” dialog

Format:	**TrOnchip.state**

Control panel to configure the on-chip breakpoint registers.

Ceva Specific Benchmarking Commands

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

The counters can be read by the application at run-time and by the debugger only when stopped.

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

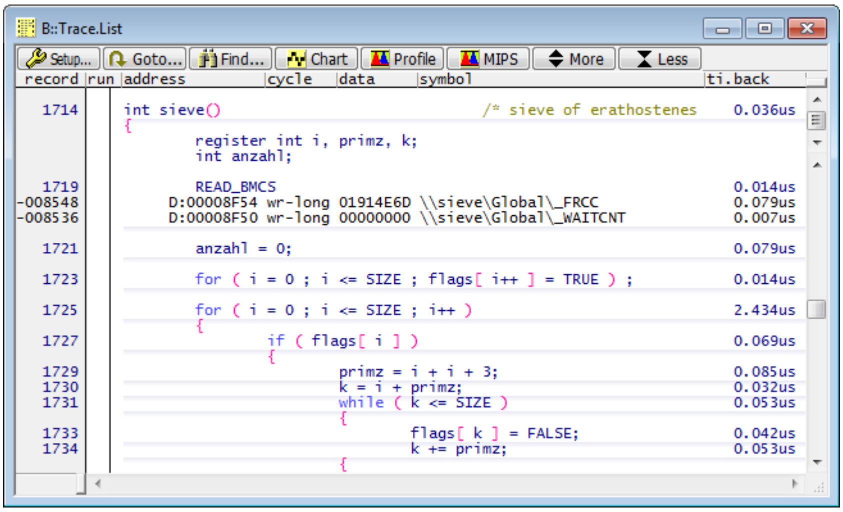
In addition the Ceva architecture offers the possibility to embed profile counter values in the program trace flow. Even though not supported by hardware directly, a little software workaround can “unlock” this useful feature.

The general approach is as follows:

1. Read profile counters by application code
2. Write counter values to a dummy location in the internal data memory
3. Configure the ETM-R4 to track data writes
4. Configure the ETM-R4 wrapper to filter out all data writes we are not interested in

For illustration we will take the *Free Running Clock Counter* (FRCC) and the *Wait Counter* (WAITCNT) as an example. Below steps have been carried out on a Ceva-XC4500 and may vary on other target systems.

In our sieve demo the result will look like this:



```
record |run |address          |cycle  |data     |symbol                        |ti.back
   1714      int sieve()                                /* sieve of erathostenes    0.036us
             {
                     register int i, primz, k;
                     int anzahl;

   1719              READ_BMCS                                                  0.014us
-008548        D:00008F54 wr-long 01914E6D \\sieve\Global\_FRCC                  0.079us
-008536        D:00008F50 wr-long 00000000 \\sieve\Global\_WAITCNT               0.007us

   1721              anzahl = 0;                                                0.079us

   1723              for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;          0.014us

   1725              for ( i = 0 ; i <= SIZE ; i++ )                            2.434us
                     {
   1727                      if ( flags[ i ] )                                  0.069us
                             {
   1729                              primz = i + i + 3;                         0.085us
   1730                              k = i + primz;                             0.032us
   1731                              while ( k <= SIZE )                        0.053us
                                     {
   1733                                      flags[ k ] = FALSE;                0.042us
   1734                                      k += primz;                        0.053us
                                     {
```

Before we can start with step #1, let’s start with some preliminary thoughts:

The ETM-R4 wrapper by Ceva offers three comparators. They can be used as filters for three different data addresses or for one range + one single address. This leads to three possible implementations of step 2):

- Declare three dummy variables somewhere in the internal memory => maximum of three profile counters to be traced.
- Write all profile counter values to the same dummy variable => no limitation of profile counters but debugger cannot distinguish between them.
- Declare a dummy variable for each counter and make sure their addresses are coherent => no limitations.

The first two items are quite straightforward and do not need any more explanation. Hence we will continue with the last one.

Unfortunately the Ceva inline assembler does not recognize structures etc. written in C language and putting all dummy variables together does not necessarily mean that the compiler will map them to continuous memory addresses. The most comfortable way to reserve a memory range for the dummy variables is to put them into a separate “dummy” data section:

```
volatile unsigned int FRCC __asm__("FRCC") __attribute__ ((section
(".DSECT dummy")));

volatile unsigned int WAITCNT __asm__("WAITCNT") __attribute__ ((section
(".DSECT dummy")));

... (etc.)
```

Note that we have to use the “volatile’ keyword. Later on we will make use of the labels *FRCC* and *WAITCNT* in the inline assembler statements only, which would be skipped by the compiler otherwise. As a result the compiler will optimize *FRCC* and *WAITCNT* away and the linker will fail when trying to link C- and assembler object files.

Now we can define a macro which can be used anywhere in our application code to output the profile counters (step #1 & #2):

```
#define READ_BMCS { \
__asm__("nop");\                                           *1, *2
__asm__("nop");\                                           *1, *2
__asm__("push{4dw} modx");\                                *2
__asm__("push{dw} r0");\
__asm__("push{dw} r7");\
__asm__("mov #0x3, mod2");\                                *2
__asm__("push{dw} a0");\
__asm__("push{dw} a7");\
__asm__("in{dw, cpm} @0x08, r0");\
__asm__("nop");\
__asm__("in{dw, cpm} @0x0c, r7");\
__asm__("nop");\
__asm__("nop");\
__asm__("nop");\
__asm__("nop");\
__asm__("mov r0, a0");\
__asm__("mov r7, a7");\
__asm__("st{dw} a0, [#_FRCC] || pop{dw}r7");\
__asm__("st{dw} a7, [#_WAITCNT] || pop{dw}r0");\
__asm__("pop{dw} a7");\
__asm__("pop{dw} a0");\
__asm__("pop{4dw} modx");\                                 *2
}
```

***1** We only need those two ‘nop’s if the macro is placed after an instruction which modifies the stack pointer (e.g. “push”, function entry).

***2** If the application does not make use of the *mod2* register, we can omit these assembler instructions and put `__asm__("mov #0x3, mod2")` somewhere else in an initialization routine.

Finally we just need to configure TRACE32 (step #3 & #4):

```
; Trace data write accesses
ETM.DataTrace Write

; Will also switch on FRCC automatically
BMC.WaitCounter ON

; Set range comparators for data write accesses
Break.Set var.address(WAITCNT)--var.end(FRCC) /Write /TraceData
```

Format:	**BMC.CLOCKS.FORMAT** *<format>*

Sets up the display format for the for each benchmark cycle counter.

```
BMC.CLOCKS.FORMAT DECimal         ; Display the cycle counter value in
                                  ; decimal format.

BMC.CLOCKS.FORMAT HEXadecimal     ; Display the cycle counter value in
                                  ; hexadecimal format.
```

Memory Classes

Memory Class	Description
D	Data memory
P	Program memory

JTAG Connector

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

Pins 17 and 19 are not used.

This is a standard 20 pin double row connector (pin-to-pin spacing: 0.100 in.).

We strongly recommend to use a connector on your target with housing and having a center polarization (e.g. AMP: 2-827745-0). A connection the other way around indeed causes damage to the output driver of the debugger.