LAUTERBACH
DEVELOPMENT TOOLS

Hexagon Debugger

MANUAL

Hexagon Debugger

Hexagon Debugger

Version 06-Jun-2024

History

29-Mar-2024	New command SYStem.Option.MACHINESPACES.
20-Jul-2022	For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.
Sep-2021	New commands SYStem.StuffInstruction.Assemble and SYStem.StuffInstruction.Opcode.

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 Hexagon 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/hexagon/ 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.

Quick Start of the ICD Debugger for Hexagon

This chapter should help you to prepare your Debugger for Hexagon. Depending on your application not all steps might be necessary. For some applications additional steps might be required that are not described in this Quick Start section.

1. Hexagon Conceptual Basics

Especially when starting to get familiar with the Hexagon architecture these points are of exceptional importance:

- Hexagon is a secure platform: by default, debugging is prohibited. Whether the user can debug a specific application or not is configured by the application which is executed.

  If you write your own application, please consult the Hexagon documentation on how to enable debugging. If you are using a third-party application please contact the vendor of this application for a debug-enabled version.

- Beside from “debugging not allowed” there are two debugging levels:

  - **Untrusted debugging** requires a debug monitor running under the control of the application and RTOS.

  - **Trusted debugging** allows full control over the Hexagon core. See also **Hexagon Security** for more information on the Hexagon debug modes.

- Because the debugger does not have any access to the core by default, Hexagon needs to be configured via some external “instance”. Normally an Arm core is responsible for configuration and loading at least an initial application for enabling debugging. Please see the chipset’s documentation on how to do this.

So before being able to establish a debug communication, e.g. by a **SYStem.Mode Up** command, the following steps have to be performed:

1. Prepare an application that has debugging enabled (trusted mode debugging)
2. Configure the Hexagon core via the Arm core or debugger and load the application to the DSP.
3. After configuration, release the Hexagon core from reset (via Arm core) so the application can enable debugging.
4. Establish debug connection via the Hexagon debugger.

2. Prepare the Start

To prevent damage please take care to follow this sequence all the time you are preparing a start.

1. Connect the debug cable to your target. Check the orientation of the connector. Pin 1 of the debug cable is marked with a small triangle next to the nose of the **target connector.**
2. Power up your TRACE32 system.
3. Start the TRACE32 Debugger Software for Hexagon.
4. Power up your target.

On some target systems, Hexagon and its JTAG port may have to be enabled by the Arm core first. Please refer to the documentation of your target system and/or contact the manufacturer to obtain information on how this has to be done.

3. Select the Clock for the JTAG Communication

You can select the JTAG clock frequency which the Debugger uses to communicate with the target. This can be either done in the JtagClock field in the SYStem window, or by using the command line with the command **SYStem.JtagClock**. The maximum clock frequency depends on your target board design. The default clock frequency is 1 MHz.

Examples:

```
SYStem.JtagClock 3.0MHz                 ; Use 3 MHz as debug clock.

SYStem.JtagClock RTCK                   ; Use the return clock for
                                        ; adjusting the debug clock.
```

4. Configure the Debugger According to the Needs of the Application

Most of the available settings can be configured with the **SYStem** window. Set the **SYStem** options in this window according to application program. The default values should be fine for most cases.

```
SYStem.Option.PC 0x0                    ; Park PC at a save location
                                        ; while thread is in debug mode.
```

5. Enter Debug Mode

```
SYStem.Up                                   ; Enter debug mode.
```

This command asserts a reset to the Hexagon core. After the reset is deasserted, the DSP will enter debug mode and start executing. When there is valid startup code which enables debugging, the core will stop at an built-in software breakpoint.

6. Load the Application

To load an image to the target, use command **Data.Load.<file_format> <file>**. The debugger supports various file formats:

```
Data.Load.ELF <file> /Verify     ; Load application file in ELF format.
```

On Hexagon platforms, the Hexagon code is normally loaded via an Arm core and not via the Hexagon debugger, so you will also need the following load options to load only the symbol information (see **Data.Load** for details):

```
/NoCODE                          ; Do not load any code or data.

/NoReg                           ; Do not set registers.
```

7. View the Source Code

Use the command **Data.List** to view the source code at the location of the program counter.

8. Write a Start-up Script

Now the quick start is done. If you were successful you can start to debug. LAUTERBACH recommends that you prepare a start-up script (a PRACTICE file) to be able to do all the necessary actions with only one command. Here is a typical start sequence:

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

WinCLEAR                         ; Clear all windows.

SYStem.Reset                     ; Set all options in the SYStem
                                 ; window to default values.

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

```
Data.LOAD.ELF application.elf       ; Load application from
                                    ; object file (ELF file).

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

Data.List                           ; Open disassembly window. *)

Register.view /SpotLight            ; Open register window. *)

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

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

Var.Watch %Spotlight flags ast      ; Open watch window for variables *)

Break.Set main                      ; Set software breakpoint to
                                    ; address of symbol main.
```

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

For information about how to build a start-up script (*.cmm file), refer to **"Training Basic Debugging"** (training_debugger.pdf). There you can also find some information on basic actions with the 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. So if there are questions related to the CPU the Processor Architecture Manual should be your first choice.

Debugger Basics

This chapter provides an introduction into the Hexagon specific debugger basics.

Memory Classes and Memory Access

Memory Class	Description
P:	Program memory.
D:	Data memory.
SD:	Data memory using on-chip System MMU.
REG:	Pseudo memory class for CPU Register set. No real-time access while CPU is running.

Prefixing an E as attribute to the memory class will make memory accesses possible even when the CPU is running. See **SYStem.MemAccess** and **SYStem.CpuAccess** for more information.

Stack Display

Hexagon has a stack which is displayed at the right side of the register window. When SP points to D:0x00000000 nothing is displayed.

Hexagon Security

Hexagon has three debug modes:

1. No debugging allowed.
2. Untrusted debug.

   The debugger communicates with a debug monitor integrated in the kernel. This allows debugging of only a few resources, e.g. some dedicated user applications or tasks.
3. Trusted debug.

   The debugger has full access and control over Hexagon.

TRACE32 only supports trusted debug.

The application running on the target selects the debug mode in its startup code. After this is done, a hard-coded software breakpoint will halt the DSP.

Virtual Hardware Threads

The debugger supportsup to the eight virtual hardware threads, the real number of threads is decided after CPU selection or based onthe value of REVID register. The PowerView GUI only shows one thread at a time. Threads can be switched with the **CORE** command:

```
CORE 2                                          ; switch to hardware thread 2.
```

Currently all breakpoints take effect on all running threads: if only one thread breaks all other threads will also break. When a breakpoint is hit, PowerView will automatically switch to the thread responsible for the break. If more than one thread hits a breakpoint at the same time, it will switch to the thread with the lowest thread number. While single stepping no automatic thread switch is performed, even when a breakpoint is hit.

Thread Compounds

Threads can be assigned to a thread compound. If any of the **Go**, **Step** or **Break** command is executed to virtual hardware thread, the command is applied to all the threads in this compound simultaneously.

By default, all virtual hardware threads are assigned to the same thread compound. Only in very few and special cases it may be necessary to remove or add a thread to the thread compound by using the **CORE** command group.

Note that breakpoints - especially software breakpoints are always assigned to all virtual hardware threads, no matter whether a thread is assigned to a thread compound or not.

Thread States

The debugger displays the Hexagon's current state in the status bar. Beside from the global states, e.g. Power Down or Reset, which are the same for all threads, each one can have its own state independently form the other states:

stopped	The thread is stopped for debug. The debugger can access memory, registers and peripherals.
running	The thread is currently executing code. Memory, registers and peripherals can not be accessed. Program execution can be interrupted.

A thread may be in a special state where it can not be debugged. These states are special implementations of the running state. In these states a thread can not be debugged, also a **Break** command does not have any effect. Other threads or an SoC event may release the thread from its state.

All these states are read from core- or thread internal status registers and not from any hardware signals. The SoC is basically available: in case of a core status the entire Hexagon is not enabled, in power down or reset by another SoC component. In case of a thread status only the specific thread is affected.

running(reset)	The core reported to be in reset state.
running(pwr_down)	The core (and register ISDBCST) is unreachable so we assume a power down mode.
running(no power)	JTAG-AP of DAP controller not responding, we assume power down as reason
running(res/pwr_down)	The core reported via ISDBCST register to be in reset or power collapse mode.
running(off)	The thread is currently in OFF mode. This reflects the E-bit of the MODECTL register.
running(waiting)	The thread is in WAIT mode, waiting to be resumed. This reflects the W-bit of the MODECTL register.
running(all_wait)	All threads are in WAIT mode. The core may perform special power saving functions.
running(no_access)	The debugger is unable to access the core or thread.

Refer to the Hexagon Architecture Documentation for more information on the modes and transitions.

On-chip Breakpoints

The Hexagon core supports two on-chip program breakpoints.

The Hexagon ETM supports up to 8 read-, write- or read/write (access) breakpoints on single addresses or 4 on ranges with the following restriction:

Hexagon ETM has 4 trigger units which can implement either one address range or up to two single addresses. Each range or single address can be programmed as read- or write breakpoint or combined read/write (access) breakpoint. An optional data condition (data breakpoint) can be added.

Program Breakpoints

By default the In-Circuit Debugger for Hexagon uses software breakpoints to implement program breakpoints. Software breakpoint means that the code at the desired memory location is modified by the debugger to make the CPU break when the program counter hits this address. After a break the original contents of the memory location are restored.

```
Break.Set P:0x00000012                ; Set a Software breakpoint.

Break.Set P:0x00000212 /Onchip        ; Set an on-chip breakpoint.

Break.Set <symbol>                    ; Set a breakpoint to <symbol>.
```

Read and Write Breakpoints

Read or write breakpoints are set up as follows:

```
Break.Set D:0x00000010 /Read          ; Set a breakpoint on a read access
                                      ; for a data address.

Break.Set D:0x00000212 /ReadWrite     ; Set a breakpoint on a read or
                                      ; write access for a data address.
```

Data Breakpoints

Data breakpoints are set up as follows. Note that a data breakpoint always has to be combined with a read-, write or read/write breakpoint.

```
Break.Set DA:0x00000010 /Read         ; Set a Breakpoint on a 16 bit
/DATA.Word 0xCAFE                     ; data read access of 0xCAFE on
                                      ; data address 0x00000010:

Break.Set myVariable /Write           ; Set a breakpoint on any write
/DATA.auto 0xAFFE                     ; access 0xAFFE to myVariable.
                                      ; Length information is taken
                                      ; from object file.
```

Restrictions

On-chip Breakpoints	For Hexagon there are only two on-chip breakpoint available. If more on-chip breakpoints are used the debugger will report an invalid breakpoint configuration. The on-chip breakpoint can only be set on code and not on data.

Troubleshooting

SYStem.Up Errors

The SYStem.UP command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command this may have the following reasons:

- The target has no power.
- There is a short-circuit on at least one output line of the CPU.
- There is a problem with the electrical connection between the debug module and the target - check if the debug cable is plugged correctly and if the target is built corresponding to the definition of the used **JTAG connector**.

FAQ

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

SYStem.CONFIG.state — Display target configuration

Format:	**SYStem.CONFIG.state** [/*<tab>*]
<tab>:	**DebugPort** | **Jtag** | **MultiTap** | **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**.
MultiTap	Informs the debugger about the existence and type of a System/Chip Level Test Access Port. The debugger might need to control it in order to reconfigure the JTAG chain or to control power, clock, reset, and security of different chip components. For descriptions of the commands on the **MultiTap** tab, see **MultiTap**.

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>:
(DebugPort)

CORE *<core>* *<chip>*
CoreNumber *<number>*
DEBUGPORT [**DebugCable0** | **DebugCableA** | **DebugCableB**]
DEBUGPORTTYPE [**JTAG** | **SWD** | **CJTAG**]
Slave [**ON** | **OFF**]
SWDPIDLEHIGH [**ON** | **OFF**]
SWDPTargetSel *<value>*
DAP2SWDPTargetSel *<value>*
TriState [**ON** | **OFF**]

<parameter>:
(JTAG)

DAPDRPOST *<bits>*
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>:
(MultiTap)

MULTITAP [**NONE** | **MSMTAP** *<irlength>* *<irvalue>* *<drlength>* *<drvalue>*
| **JtagSEQuence** *<sub_cmd>*]

<parameter>:
**(AccessPorts
)**

AHBAPn.Base *<address>*
AHBAPn.HPROT [*<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.HPROT [*<value>* | *<name>*]

<parameter>: (AccessPorts cont.)

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

ETF.ATBSource *<source>*
ETF.Base *<address>*
ETF.Name *<string>*
ETF.NoFlush [**ON** | **OFF**]
ETF.RESet
ETF.Size *<size>*

<parameter>: (COmponents cont.)

ETF.STackMode [**NotAvailbale** | **TRGETM** | **FULLTIDRM** | **NOTSET** | **FULL** **STOP** | **FULLCTI**]
ETF.view

ETM.Base *<address>*
ETM.RESet

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

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**]

TPIU.ATBSource *<source>*
TPIU.Base *<address>*
TPIU.Name *<string>*
TPIU.RESet
TPIU.Type [**CoreSight** | **Generic**]
TPIU.view

<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>*
<parameter>: (Deprecated cont.)	**FUNNEL2BASE** *<address>* **FUNNELBASE** *<address>* **STMETBFUNNELPORT***<port>* **STMFUNNEL2PORT***<port>* **STMFUNNELPORT***<port>* **STMTPIUFUNNELPORT***<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.

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**.

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,...).

CoreNumber *<number>*

Number of cores to be considered in an SMP (symmetric multiprocessing) debug session.

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

It specifies which probe cable shall be used e.g. "DebugCableA" or "DebugCableB". At the moment only the CombiProbe allows to connect more than one probe cable.

Default: depends on detection.

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

It specifies the used debug port type “JTAG”, “SWD”, “CJTAG”, “CJTAG-SWD”. It assumes the selected type is supported by the target.

Default: JTAG.

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.

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).
For CortexM: Please check also
SYStem.Option.DISableSOFTRES [ON | OFF]

TAPState *<state>*

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

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

Default: 7 = Select-DR-Scan.

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.

MULTITAP
[**NONE** | **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).

For a description of the **JtagSEQuence** subcommands, see **SYStem.CONFIG.MULTITAP JtagSEQuence**.

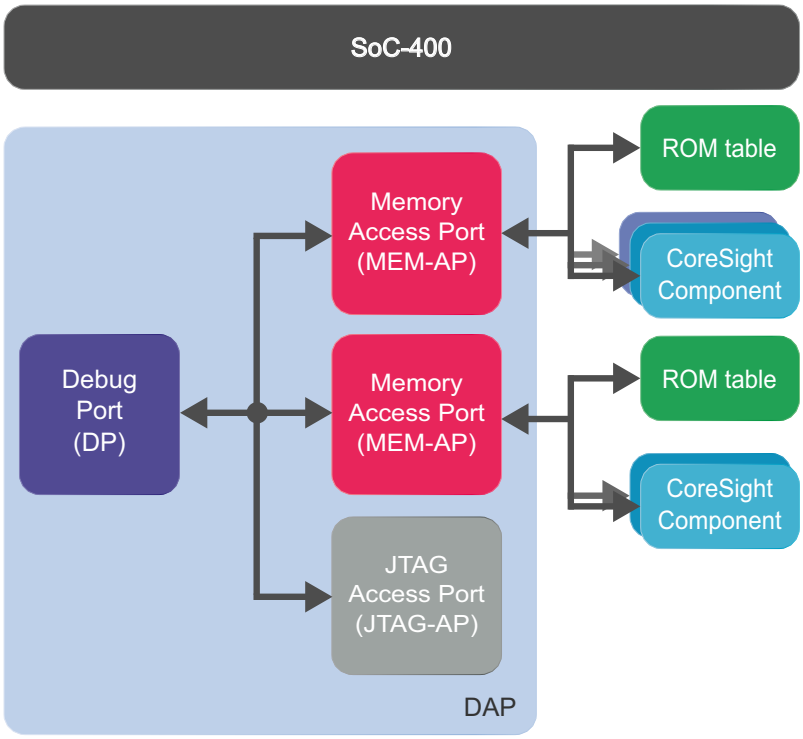
An Access Port (AP) is a CoreSight module from Arm which provides access via its debug link (JTAG, cJTAG, SWD, USB, UDP/TCP-IP, GTL, PCIe...) to:

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).

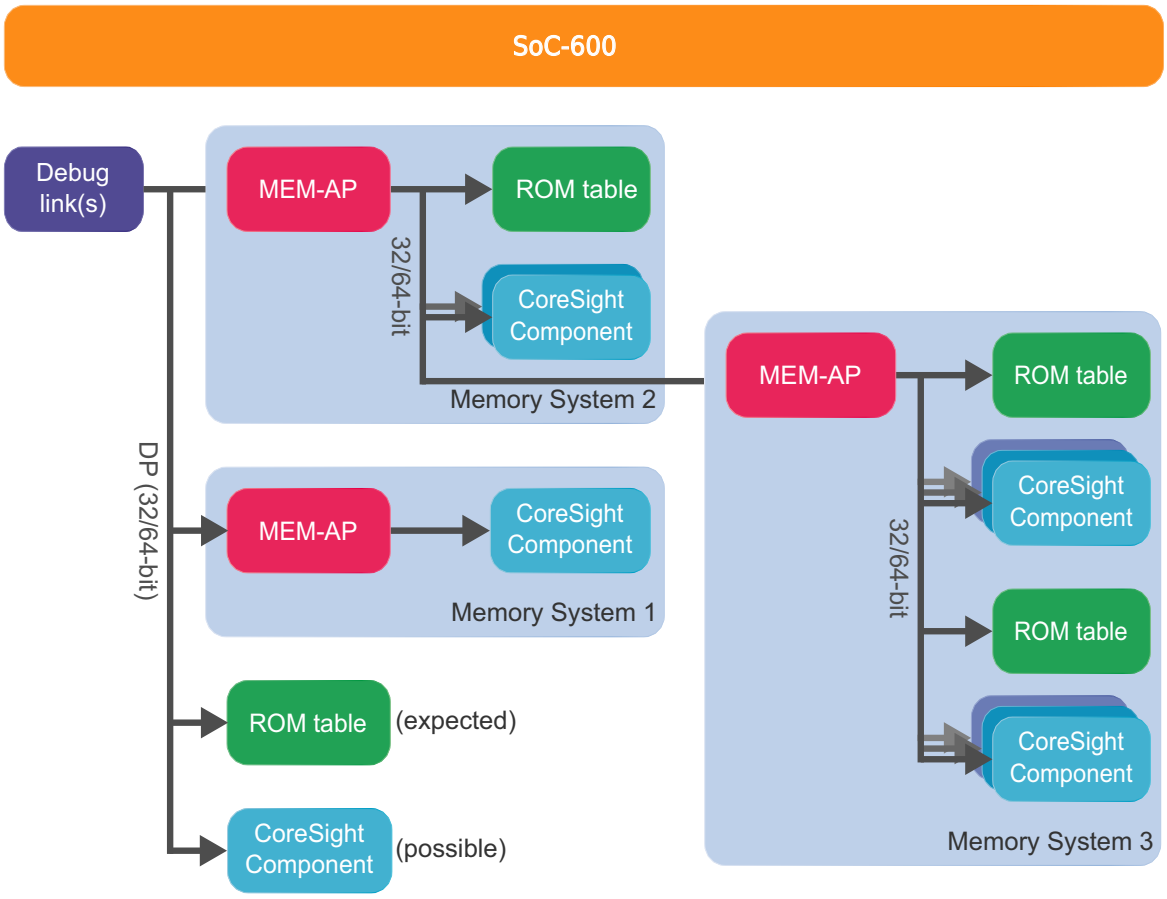
2. Other, chip-internal JTAG interfaces. This is especially important if the core you intend to debug is connected to such an internal JTAG interface. The module controlling these JTAG interfaces is called JTAG Access Port (JTAG-AP). Each JTAG-AP can control up to 8 internal JTAG interfaces. A port number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

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.

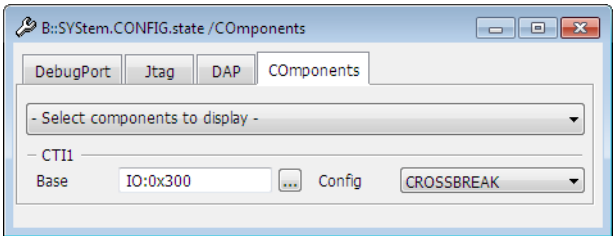
Example: SYStem.CONFIG.AXIAP1.Base DP:0x80004000
Meaning: The control register block of the AXI access ports starts at address 0x80004000.

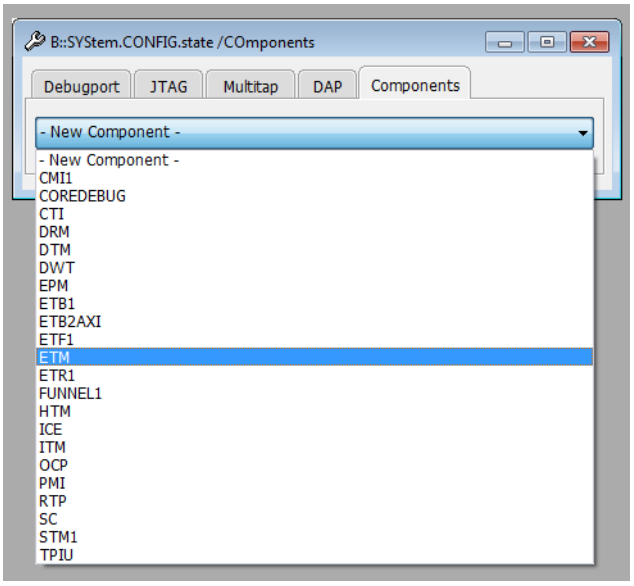
JTAGAPn.Base *<address>*

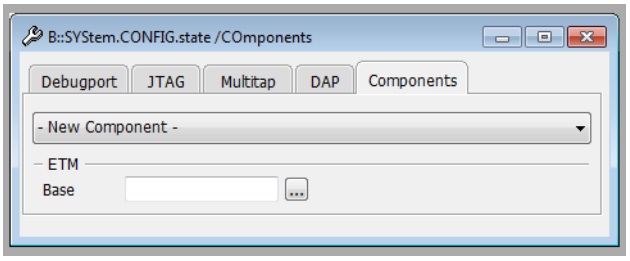
This command informs the debugger about the start address of the register block of the “JTAGAPn:” access port. And this way it notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus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

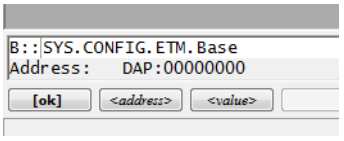
On the **Components** tab in the **SYStem.CONFIG.state** window, you can comfortably add the debug and trace components your chip includes and which you intend to use with the debugger's help.







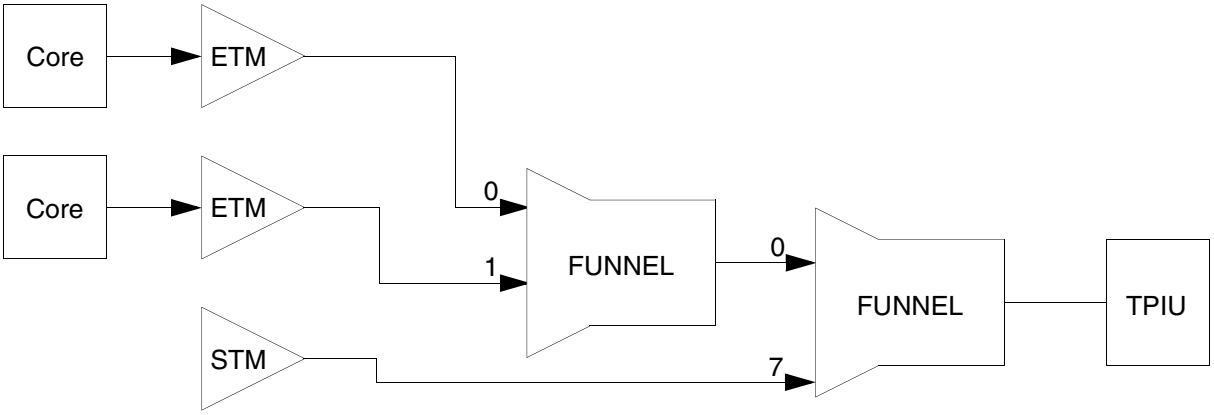
Each configuration can be done by a command in a script file as well. Then you do not need to enter everything again on the next debug session. If you press the button with the three dots you get the corresponding command in the command line where you can view and maybe copy it into a script file.



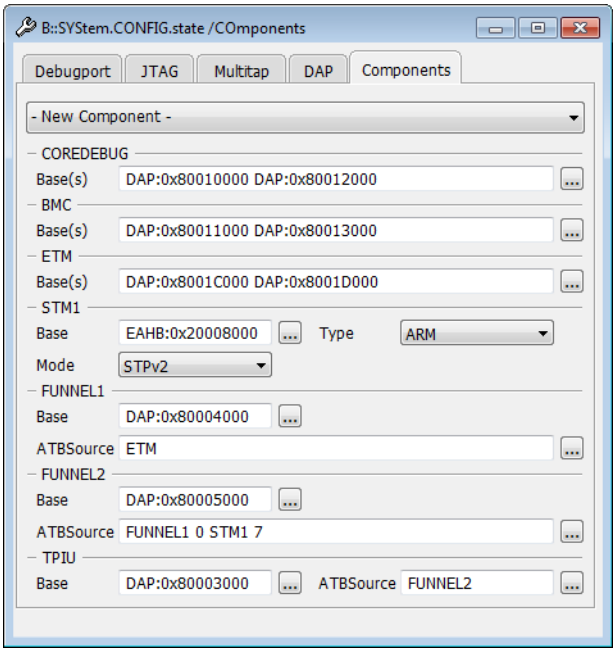
You can have several of the following components: CMI, ETB, ETF, ETR, FUNNEL, STM.
Example: FUNNEL1, FUNNEL2, FUNNEL3,...

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

Example:




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



… **.ATBSource** *<source>*

Specify for components collecting trace information from where the trace data are coming from. This way you inform the debugger about the interconnection of different trace components on a common trace bus.

You need to specify the "... .Base *<address>*" or other attributes that define the amount of existing peripheral modules before you can describe the interconnection by "... .ATBSource *<source>*".

A CoreSight trace FUNNEL has eight input ports (port 0-7) to combine the data of various trace sources to a common trace stream. Therefore you can enter instead of a single source a list of sources and input port numbers.

Example: 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.

CTICH01: Channel 0 and 1 of the CTM are used to distribute start/stop events from and to the CTIs. Armv8 only.
CTICH23: 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.

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

ETS.ATBSource *<source>*
ETS.Base *<address>*
ETS.RESet
Embedded Trace Streamer (ETS) - Arm CoreSight module

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

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

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".

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

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

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

… **PORT** *<port>*

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

Example: SYStem.CONFIG STMFUNNEL2PORT 3

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

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

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

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

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.

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>*
STMETBFUNNELPORT *<port>*	**FUNNEL4.ATBSource STM1** *<port>* (1)
STMFUNNEL2PORT *<port>*	**FUNNEL2.ATBSource STM1** *<port>* (1)
STMFUNNELPORT *<port>*	**FUNNEL1.ATBSource STM1** *<port>* (1)
STMTPIUFUNNELPORT *<port>*	**FUNNEL3.ATBSource STM1** *<port>* (1)
TPIUBASE *<address>*	**TPIU.Base** *<address>*
TPIUFUNNELBASE *<address>*	**FUNNEL3.Base** *<address>*
view	**state**

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

Format: **SYStem.CONFIG.MSA** *<address>*

Default: 0xffffffff (disabled).

Only available for HexagonV5 and later.

When specifying the MSA register address, L2-address translation by the on-chip system-level MMU is enabled. See your SoC documentation for the MSA register address.

The debugger knows from the TLB or page table entry whether a resolved address is to be translated once more by the System MMU. In case the debugger or OS Awareness cannot resolve this second translation the SoC can be instructed to do it by setting bit 2 in this register.

The user can manually trigger this translation using memory class SD:. This assumes that the core MMU translation has already been performed.

SYStem.CPU

Select CPU type

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

<cpu>: **HexagonV2** |
HexagonV3 | **HexagonV3L** | **HexagonV3M** | **HexagonV3C** |
HexagonV4 | **HexagonV4L** | **HexagonV4M** | **HexagonV4C** |
HexagonV5 | **HexagonV55** | **HexagonV56** |
HexagonV60

Default: HexagonV2.

Select your CPU.

Format: **SYStem.JtagClock** *<frequency>*
SYStem.BdmClock (deprecated)

<frequency>: **1.0 kHz**…**80.0 MHz**

Default: 10.0 MHz.

This command selects the frequency of the JTAG clock, which is used to communicate with the Hexagon core. The maximum reachable frequency is target dependent, the default should be fine. If no unit is specified, Hz is assumed.
The frequency modes RTCK, ARTCK, CTCK and CRTCK are not supported by Hexagon.

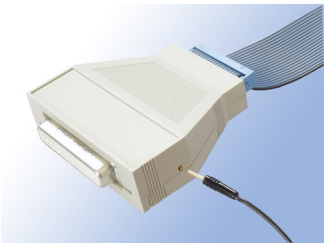
SYStem.LOCK Tristate the JTAG port

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

Default: OFF.

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

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



There is a single cable contact on the casing of the debug cable which can be used to detect if the JTAG connector of the debugger is tristated. If tristated also this signal is tristated, otherwise it is pulled low.

Format:	**SYStem.MemAccess** *<mode>* **SYStem.ACCESS** (deprecated)
<mode>:	**Enable** | **Denied** | **DAP** | **StopAndGo**

Default: Enable.

This option declares if and how a non-intrusive memory access can take place while the CPU is executing code. Although the CPU is not halted, run-time memory access creates an additional load on the processor's internal data bus. Default access mode is CPU. The currently selected run-time memory access mode is printed in the state line.

The run-time memory access has to be activated for each window by using the memory class **E:** (e.g. **Data.dump ED:0x000000**) or by using the format option **%E** (e.g. **Var.View %E var1**).

Enable **CPU** (deprecated)	The debugger performs memory accesses via a dedicated CPU interface. Hexagon has no such CPU interface.
Denied	Non-intrusive memory access is disabled while the CPU is executing code. Instead intrusive accesses can be configured with **SYStem.CpuAccess**.
DAP	A run-time memory access is done via the Debug Access Port (DAP). This is only possible if a DAP is available on the SoC and if the memory bus is connected to it. Not possible on the instruction set simulator.
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.Option.MMUSPACES Separate address spaces by space IDs

Format:	**SYStem.Option.MMUSPACES** [**ON** | **OFF**] **SYStem.Option.MMUspaces** [**ON** | **OFF**] (deprecated) **SYStem.Option.MMU** [**ON** | **OFF**] (deprecated)

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

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

NOTE:	**SYStem.Option.MMUSPACES** should not be set to **ON** if only one translation table is used on the target. If a debug session requires space IDs, you must observe the following sequence of steps: 1. Activate **SYStem.Option.MMUSPACES**. 2. Load the symbols with **Data.LOAD**. Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

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

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

SYStem.Option.MACHINESPACES Address extension for guest OSes

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

Default: OFF

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

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

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

- Addresses are extended with an identifier called machine ID. The machine ID clearly specifies to which host or guest machine the address belongs.

  The host machine always uses machine ID 0. Guests have a machine ID larger than 0. TRACE32 currently supports machine IDs up to 30.

- The debugger address translation (**MMU** and **TRANSlation** command groups) can be individually configured for each virtual machine.

- Individual symbol sets can be loaded for each virtual machine.

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
Go
Attach
StandBy
Up

Down Stops the JTAG communication to the core.

NoDebug Releases the core from reset without establishing a JTAG connection to the core. The debug connector is tristated and behaves as if it was not connected. The core will start running.

Go Establishes a JTAG connection, initializes the debug interface. The core is reset and will start running from the reset vector.

Attach User program remains running (no reset) and the debug interface is initialized. After this command the user program can be stopped with the break command or if a break condition occurs.

StandBy The target processor waits for target power and clock. When both are active it starts as fast as possible. This functionality is not available on all targets.

Up Establishes a JTAG connection, initializes the debug interface. The core is reset and will start running from the reset vector. If the application has enabled debugging, the DSP will halt at the initial software breakpoint, if set by the application.

Default: Down.

When debugging is prohibited or the Hexagon core is not correctly configured, establishing the debug connection fails and the debugger will revert into the Down state. One of the following error messages will be printed, and the **AREA** window shows some additional information. See **Hexagon Security** for more information on the Hexagon debug modes.

Emulator debug port secured	Debugging is either permitted by the application or untrusted debugging has been selected (not supported by TRACE32).
Emulator debug port locked	The Hexagon core was not enabled or configured by the Arm core. Either it is still in power down mode (all Hexagon versions) or in reset (Hexagon v4 and newer).
Emulator debug port fail	Setting the debug mode by the application failed. The debugger can communicate with the debug system but gets no further access.

The “Emulate” LED on the debug module is ON when the debug mode is active and the CPU is running.

When **SYStem.Option.SRST** is OFF, only a Hexagon internal reset is performed. The reset has no impact on other cores in the same chip (depending on the design). This internal reset is also performed when **SYStem.CONFIG.Slave** is enabled.

SYStem.Option CPU specific commands

The SYStem.Options are a class of Architecture and CPU specific commands.

SYStem.Option.BUGFIX Workaround for single-stepping an RTE instruction

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

Default: OFF (Hexagon v1: ON).

Workaround for single-stepping an RTE instruction on Hexagon v1. Do not enable on other Hexagon variants.

SYStem.Option.DCFREEZE Do not invalidate cache

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

Default: ON.

Instruct the debugger whether to provide data cache coherency on a memory access or not. The behavior is different for the various Hexagon variants:

Hexagon v1, v2 and v3 behave as follows when performing a memory access via memory classes D:, DC: and L2:

- If enabled and L1 data cache is enabled, the debugger will disable L1 caches. This assures cache coherency for L1 cache state and content.
- If disabled, all accesses are performed via caches as far as they are enabled. This usually allocates new lines in the cache and so alters the cache states.

In case of any issues with cache content, try to disable DCFREEZE for Hexagon v1, v2 and v3.

Hexagon v4 behaves as follows when performing a memory access via memory classes P:, IC:, D:, DC: and L2:

- If enabled, the debugger will perform a non-destructive read access regarding the L1 data cache by using the memw_phys() instruction.
- If disabled, the debugger will perform a destructive read access which will result in newly allocated cache lines in L1 data cache.

For Hexagon v4, enabling DCFREEZE is recommended.

SYStem.Option.CLADE Enable debugger support for CLADE

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

Default: OFF.

When enabled, TRACE32 will handle RAM regions compressed using CLADE.

SYStem.Option.CLADEDICT Load CLADE dictionary from file

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

Loads the clade dictionary from a file.

SYStem.Option.CLADEPARAM Define the CLADE address parameters

Format: **SYStem.Option.CLADEPARAM** *<high> <low> <main> <excp_high> <excp_low> <excp_low_small>* […]

Inform TRACE32 about the location of the compressed memory regions in memory.

SYStem.Option.CLADEREAD Use hardware for reading compressed RAM

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

Default: ON.

When enabled, TRACE32 will read compressed memory using CLADE hardware. When disabled, TRACE32 will use the built-in CLADE library.

SYStem.Option.ICFLUSH Flush instruction cache at “Go” or “Step”

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

Default: ON.

In case the instruction memory is modified, Instruction Cache(s) have to be kept coherent with RAM, otherwise modifications (e.g. Software Breakpoints) may not take effect. There are two strategies how the invalidation is performed:

- If ICFLUSH is enabled, the entire L1 instruction cache is unconditionally invalidated before program execution is resumed (executing a **Go** or **Step** command). The entire L2 cache is also invalidated in the following cases:

  Hexagon v2: in case L2 cache is configured.

  Hexagon v3: in case L2 cache is configured and in write-through mode

  Hexagon v4: never

  In case L2 cache is configured but can not be invalidated completely, e.g. because it is in copy-back mode or in case of Hexagon v4, L2 cache coherence is maintained as for the ICFLUSH disabled case.

- If ICFLUSH is disabled, the affected lines in L1 instruction cache are invalidated in the instant of the memory modification. The affected L2 cache lines are also invalidated, but note that this is performed by invalidation of the corresponding L1 data cache lines.

In case ICFLUSH is disabled, the instruction cache is only invalidated when writing via memory class P: or IC:.

Note that Hexagon v1 has no L2 cache implemented.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

If enabled, the debug core will disable all interrupts for the CPU when single stepping assembler instructions. No hardware interrupt will be executed during single-step operations. When you execute a Go command, the hardware interrupts will be enabled again, according to the system control registers.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the debug core will disable all interrupts for the CPU during HLL single-step operations. When you execute a Go command, the hardware interrupts will be enabled again, according to the system control registers. This option should be used in conjunction with **SYStem.Option.IMASKASM**.

SYStem.Option.ISDBSoftBreakFix Workaround for prefetch issue

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

Default: OFF.

NOTE: This workaround is only for early HexagonV60 devices only used by Qualcomm internally. Only enable this option when your device is affected by this issue.

A bug in the prefetcher causes ISDB to lock when a breakpoint is hit. The debugger is then not able to communicate with Hexagon any more. This bug affects onchip and software breakpoints.

The debugger's workaround is to write an ISYNC instruction right after the breakpoint instruction. This has the following restrictions:

- Setting onchip breakpoints is not possible. Use software breakpoints instead.
- Setting software breakpoints at consecutive addresses is not possible.
- The instruction followed by the software breakpoint must not be the target of a jump. The user has to ensure this when setting the breakpoint.

SYStem.Option.PC PC parking position

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

Default: 0xffffffff

If *<base>* is programmed to a different value than `0xffffffff`, the PC will parked at the given address while the thread is breaked. This is a recommended workaround for several silicon issues.

Select a valid physical address which has no jump or call instruction. It is recommended not to park in the middle of a multi-instruction packet.

Format: **SYStem.RESetOut**

This command resets the DSP via the debug registers in ISDB. Only the DSP will reset, not the debug port or the target system. This function only works when the CPU is in **SYStem.Mode Up**.

SYStem.Option.REVision Define default value for REV register

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

Default: 0x00000000

Allows to specify a default value for the REV register. Applications evaluate this register to get information about the Hexagon version and implemented features.

This command is available in the TRACE32 simulator only. The command becomes effective after the next **SYStem.Mode Up** or **Register.Init** and will be permanent. Use the **Register.Set** REV to change the REV register temporary and immediately.

SYStem.Option.SRST Reset via SRST line

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

Default: OFF.

When enabled (ON), the debugger also resets the core via the nSRST line of the JTAG connector. It also holds the core in reset while in **SYStem.Mode Down**. When disabled, only Hexagon specific resets will be generated if possible.

On some evaluation boards where the nSRST line is also connected to other cores and chips a reset via nSRST will have unwanted effects.

SYStem.StuffInstruction.Assemble Execute assembly stuff instruction

[build 139867 - DVD 02/2022]

Format: **SYStem.StuffInstruction.Assemble 0x0** *<address> <assembly>*

Execute a stuff instruction (given in assembly) without modifying the instruction memory. The instruction will be executed in the thread's current mode. The instruction may not assemble to more bytes than the stuff instruction register can hold. The address is a hint to the assembler that is required for address relative operations (e.g., jump). For Hexagon, this can always be "r(pc)" to enable relative jumps. For operations that require no base address, this can be set to 0. Note that instructions are truncated if the assembled opcode is longer than the stuff instruction register.

SYStem.StuffInstruction.Opcode Execute opcode stuff instruction

[build 139867 - DVD 02/2022]

Format: **SYStem.StuffInstruction.Opcode** *<opcode>*

Execute a stuff instruction (given as opcode) without modifying the instruction memory. The instruction will be executed in the thread's current mode.

SYStem.Option.TCMBase Base address of the TCM

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

Default: 0x00000000 (disabled)

Only required for Hexagon v2 and v3, ignored for all other Hexagon versions.

The location of the Tightly Coupled Memory (TCM) depends on the device. As the L2 cache is implemented in the TCM, the debugger needs to know where to look for it. Disabling the TCM (setting the base address to 0x00000000) disables the L2 cache support of TRACE32.

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

Default: OFF

This is a workaround necessary for some early Hexagon v5 devices. When the debugger accesses memory while the MMU is OFF, the TLB might get corrupted. Enabling this option ensures that the debugger invalidates all TLB entries when disabling the MMU. When enabling again, all entries are restored.

SYStem.state

Display SYStem.state window

Format: **SYStem.state**

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

MMU.DUMP Page wise display of MMU translation table

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

Displays the contents of the CPU specific MMU translation table.

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

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

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

TLB	Displays the contents of the Translation Lookaside Buffer.
MAP	Displays the contents of the MAP table.
VTLB	Displays the contents of a virtual TLB (VTLB) translation table. If **SYStem.Option.MMUSPACES** is set to OFF, the list of VTLB entries is sorted ascending by address only. If **SYStem.Option.MMUSPACES** is set to ON, the list of VTLB entries is sorted by the ASID first and then by the address.

MMU.List — Compact display of MMU translation table

Format:	**MMU.List** *<table>* [*<range>* | *<address>* | *<range>* *<root>* | *<address>* *<root>*] **MMU.***<table>***.List** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**

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

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

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.

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

MMU.MAP Translations from ELF file memory load map

MMU.MAP.dump Display addresses from ELF file memory load map

Format: **MMU.MAP.dump**

Shows the logical and associated physical addresses of the program sections loaded from an ELF file. This command is an alias for **sYmbol.List.MAP**.

MMU.MAP.SCAN Load MMU table from ELF file

Format: **MMU.MAP.SCAN**

Loads the logical and associated physical addresses of the program sections loaded from an ELF file to the debugger-internal translation table.The loaded address translation can be viewed with **TRANSlation.List**.

See also **MMU.MAP.dump**.

Format:	**MMU.SCAN** *<table>* [*<range>* *<address>*] **MMU.***<table>***.SCAN** (deprecated)
<table>:	**PageTable** **KernelPageTable** **TaskPageTable** *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0** **ALL** [**Clear**] **TLB** | **VTLB** | **MAP**

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

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

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

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID: loads the translation table of the specified process • else, this command loads the table the CPU currently uses for MMU translation.

KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.
ALL [**Clear**]	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**. **Clear:** This option allows to clear the static translations list before reading it from all page translation tables.

CPU specific tables in MMU.SCAN <table>

TLB	Loads the translation table from the CPU to the debugger-internal translation table.
MAP	Loads the logical and associated physical addresses of the program sections loaded from an ELF file to the debugger-internal translation table.
VTLB	Loads the translations from a virtual TLB (VTLB) translation table to the debugger-internal translation table. If **SYStem.Option.MMUSPACES** is set to **OFF**, the VTLB entries will be scanned, disregarding their ASID field. If **SYStem.Option.MMUSPACES** is set to **ON**, the list of VTLB entries will be scanned and their ASID field will be copied to the space ID of the new static translation entry.

Format: **MMU.Set** [**TLB** [<*index*> [<*tlb0*> [<*tlb1*>]]]]

TLB Writes data to the processor's TLB.

index Index (entry) in TLB table

tlb0
tlb1 Data of the TLB entries

MMU.TLB Scan or dump MMU TLB entries

Format: **MMU.TLB.**<*sub_cmd*>

<*sub_cmd*>: **dump** | **SCAN**

dump Displays a table of all entries from the CPU TLB table.

SCAN Loads the TLB table entries from the CPU to the debugger-internal MMU table.

Format:	**MMU.VTLB.**<*sub_cmd*>
<*sub_cmd*>:	**dump** | **SCAN**

dump Displays the entries of a virtual TLB (VTLB) translation table.
See **MMU.DUMP.VTLB**.

SCAN Loads the translations from a virtual TLB (VTLB) translation table to the debugger-internal translation table.
See **MMU.SCAN.VTLB**.

CPU specific BenchMarkCounter Commands

The Hexagon-ETM contains six counters which can record various events related to DSP performance.

The benchmark counters can be read at run-time. The counter values are periodically packed into so-called profiling packets and sent out through the trace port.

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<n> Specify event count

Format:	**BMC.COUNTER***<n>* *<event>*
<event>:	**DCMISS** | **DCCCONFLICT** | **ICMISS** | **ICSTALL** | **ITLBMISS** | **DTLBMISS** | **STALLS**

Specifies which event is counted by an event counter.

DCMISS	data cache misses
DCCONFLICT	data cache conflicts
ICMISS	instruction cache misses
ICSTALL	instruction cache stall-cycles
ITLBMISS	itlb misses
DTLBMISS	dtlb misses
STALLS	all stall cycles

```
BMC.COUNTER1 ICSTALL
```

Format: **BMC.CyclePeriod** *<cycles>*

Specifies the number of clock cycles for which an event is counted before it is sent out through the trace port.

```
BMC.CyclePeriod 500.                ; count the specified events for
                                    ; 500 clock cycles,
                                    ; pack the result in a profiling
                                    ; packet and sent it out
```

BMC.OFF Disable benchmark counters

Format: **BMC.OFF**

Disable TRACE32 BMC functionality.

BMC.ON Enable benchmark counters

Format: **BMC.ON**

Enable TRACE32 BMC functionality.

BMC.SELect Select counter for statistic analysis

Format: **BMC.SELect** *<counter>*

<counter>: **COUNTER0** | **COUNTER1** | …

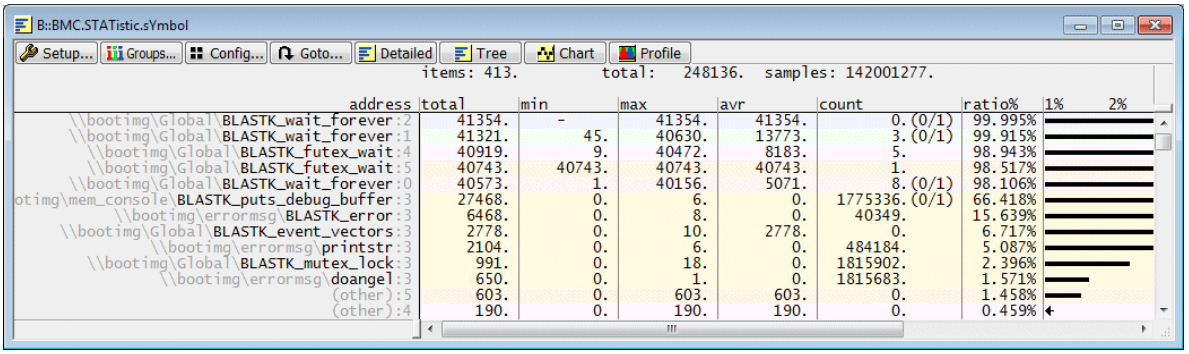
The exported event counter values can be combined with the exported instruction flow in order to get a clearer understanding of the program behavior. The command **BMC.SELect** allows to specify which counter is combined with the instruction flow to get a statistical evaluation.

```
BMC.COUNTER0 STALL                      ; counter0 counts all stall cycles

…

BMC.SELect COUNTER0                     ; counter0 is selected for the
                                        ; combination with the instruction
                                        ; flow

BMC.STATistic.sYmbol                    ; assign the stall cycles to the
                                        ; executed functions/symbol
                                        ; regions
```



BMC.SPDM — Specify profiler control

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

OFF (default)	The ETM region counter controls the accumulation/packet generation.
ON	The SPDM controls the profiler accumulation/packet generation.

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

OFF (default)	The specified events are counted for all hardware thread.
ON	A single event is counted separately each hardware thread.

TrOnchip.ContextID Extend on-chip breakpoint/trace filter by TID

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

OFF (default)	Stop the program execution at on-chip breakpoint if the address matches. Trace filters and triggers become active if the address matches.
ON	Stop the program execution at on-chip breakpoint if the address and the TID matches. Trace filters and triggers become active if the address and the TID matches. Precondition is that the RTOS operates the TID.

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

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

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

TrOnchip.RESet — Reset “TrOnchip” settings

Format: **TrOnchip.RESet**

Reset **TrOnchip** settings to default.

TrOnchip.StepException — Single stepping of exceptions and interrupts

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

OFF (default)	Execute exception in real-time while single stepping. Execute interrupts in real-time while single stepping (only if **SYStem.Option.IMASKASM** is **OFF**).
ON	Enable single stepping for exceptions and interrupts (only if **SYStem.Option.IMASKASM** is **OFF**).

TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

Mechanical Description of the 20-pin Debug Cable

The connector for Hexagon is identical to the connector for Arm cores as defined by Arm:

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

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.

- The input and output signals are connected to a supply translating transceiver. Therefore the ICD can work in a voltage range of 0.4 … 5.0 V (5.25 V).
- VTREF is used as a sense line for the target voltage.
- nTRST, TDI, TMS, TCK are driven by the supply translating transceiver. In normal operation mode this driver is enabled, but it can be disabled to give another tool access to the JTAG port. In environments where multiple tools can access the JTAG port, it is required that there is a pull-up or pull-down resistor at TCK. This is to ensure that TCK maintains its level during a handover between different tools.
- RTCK is the return test clock signal from the target JTAG port. This signal can be used to synchronize JTAG clock with the processor clock (see **SYStem.JtagClock**). Hexagon does not support RTCK.
- TDO is an ICD input. It is connected to the supply translating transceiver.
- nSRST (=nRESET) is used by the debugger to reset the target CPU or to detect a reset on the target. It is driven by an open collector buffer. A pull-up resistor is included in the ICD connector. The debugger will only assert a pulse on nSRST when the SYStem.UP, the SYStem.Mode Go or the SYStem.RESetOUT command is executed.
- EDBGRQ is driven by the supply translating transceiver. This line is optional. It allows to halt the program execution by an external trigger signal.
- DBGACK is an ICD input. It is connected to the supply translating transceiver. A pull-down resistor is included in the ICD connector. This line is optional. It allows exact runtime measurement and exact triggering of other devices on a program execution halt.
- N/C (= Vsupply) is not connected in the ICD. This pin is used by debuggers of other manufacturers for supply voltage input. The ICD is self-powered.

There is an additional plug in the connector on the debug cable to the debug interface. This signal is tristated if the JTAG connector is tristated by the debugger and it is pulled low otherwise. This signal is normally not required, but can be used to detect the tristate state if more than one debug tools are connected to the same JTAG port.