LAUTERBACH
DEVELOPMENT TOOLS

ARC Debugger and Trace

MANUAL

ARC Debugger and Trace

ARC Debugger and Trace

Version 06-Jun-2024

History

16-Jun-23	Chapter 'Legacy MIPI60-C Connector' was removed.
17-Jan-23	Added SETUP.DIS command.
20-Jul-22	For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.
07-Jun-22	New command: JTAG.CONFIG.TckRun.
17-Feb-22	New command SYStem.Option.TRST.

Introduction

This document describes the processor-specific settings and features of the ARC in-circuit debugger.

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

Supported ARC Cores

The following ARC cores from Synopsys, Virage Logic or ARC International are supported:

- ARC Vector DSPs : EV7x, EV7xFS, VPX2/VPX3/VPX5, VPX2FS/VPX3FS/VPX5FS
- ARC-HS family : HS34, HS36, HS38, HS44, HS45D, HS46, HS46FS, HS47D, HS47DFS, HS48, HS48FS
- ARC-EM family : EM4, EM5D, EM6, EM7D, EM9D, EM11D, EM22FS
- ARC 700 core family : ARC710D, ARC725D, ARC750D, ARC770D
- ARC 600 core family : ARC601, ARC605, ARC610D, ARC652D, ARC630D, AS211SFX, AS221BD
- ARCtangent-A5 cores
- ARCtangent-A4 cores

Brief Overview of Documents for New Users

Architecture-independent information:

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

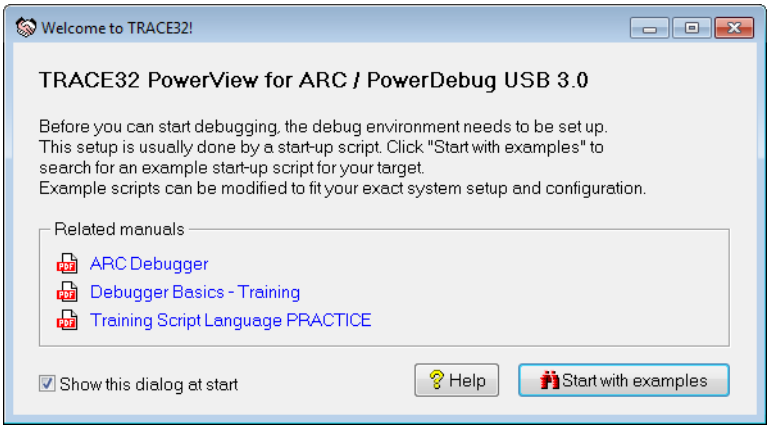
Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:

- Choose **Help** menu > **Processor Architecture Manual**.

- **"OS Awareness Manuals"** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

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



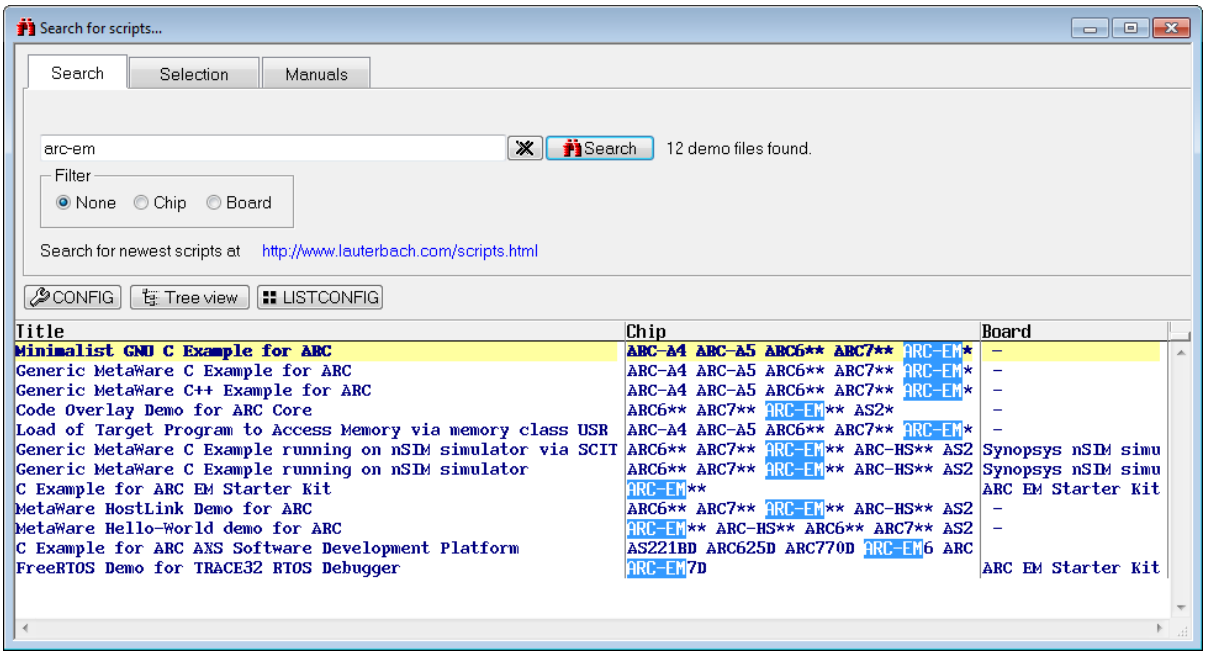
Demo and Start-up Scripts

Lauterbach provides ready-to-run PRACTICE start-up scripts and examples.

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/arc/ subfolder of the system directory of TRACE32.

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.

Troubleshooting

The LAUTERBACH debug module LA-7701 “DEBUG INTERFACE” is not supported.
You require a Power Debug module (e.g. LA-7702, LA-7704, LA-7705, LA-7707, LA-7708, LA-7690, LA-7699)

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.
- A FPGA which could hold an ARC Core like an ARCangel, is not loaded yet or does not contain an ARC Core with a debugging interface.
- There is a problem with the electrical connection between the debugger and the target - check if the JTAG connector 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.

Quick Start

This chapter should help you to prepare your debugger for ARC. Depending on your application not all steps might be necessary.

For some applications additional steps might be necessary, that are not described in this Quick Start section.

1. Prepare the Start

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**.

Power up your TRACE32 system.

Start the TRACE32 Debugger Software.

Power up your Target!

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

2. Select the CPU Type

For example:

```
SYStem.CPU ARC-HS
```

If you have a normal ARC core without the need of special configurations (e.g. position inside a JTAG daisy chain) you can also use the keyword AUTO. E.g.:

```
SYStem.CPU AUTO
```

3. Set the speed of the JTAG debug clock

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 might depend on the configuration of your FPGA design. The default clock frequency is 1 MHz.

4. Configure the JTAG debug accesses

If you have a single ARC core and you use standard JTAG (IEEE 1149.1) there is nothing you have to do in this step.

Configure a multi-core setup

If you have more than one CPU core connected to the same JTAG port, please tell the debugger how it should connect to the core you want to debug:

In case of a JTAG daisy chain use command **SYStem.DETECT SHOWChain** to scan the chain. The result is shown in a window. Double-click on the desired core to tell the debugger which core you'd like to debug.

To configure the position of your core in the JTAG daisy chain manually use commands **SYStem.CONFIG IRPOST**, **SYStem.CONFIG IRPRE**, **SYStem.CONFIG DRPOST** and **SYStem.CONFIG DRPRE**.

In case your CPU is designed in the mature ARC MADI multicore configuration, please use command **SYStem.CONFIG MADI** to specify the core you want to debug.

Configure Compact-JTAG

Some ARC cores must be debugged via a two-wire debug interface called Compact-JTAG, cJTAG or IEEE 1149.7.

Use command **SYStem.CONFIG DEBUGPORTTYPE CJTAG** to enable two wire mode.
Add command **SYStem.CONFIG CJTAGFLAGS 0x03** to skip TCA-scanning and to use TRACE32-pseudo-keeper to workaround problems with the cJTAG implementation of your core.

5. Enter Debug Mode

```
SYStem.Up                          ; Connect to ARC core, stop the core and
                                   jump to reset vector
```

This command resets the CPU, enters debug mode and jumps to the break address of the debugged core. After this command is executed, it is possible to access memory and registers.

6. Load your Application Program

When the core is prepared the code can be downloaded. This can be done with the command **Data.Load.<file_format> <file>**.

```
Data.Load.Elf <file>.elf           ; load application file
```

The options of the **Data.LOAD** command depend on the file format generated by the compiler. A detailed description of the **Data.LOAD** command is given in **"General Commands Reference"**.

7. Initialize Program Counter and Stackpointer (if required)

In a ready-to-run compiled ELF file, these settings are in the start-up code of the ELF file. In this case nothing has to be done. You can check the contents of Program Counter and Stack Pointer in the Register window, which provides the contents of all CPU Registers. Use CPU Registers in the CPU menu to open this window or use the command **Register**.

The Program Counter and the Stackpointer and all other registers can be set with the commands **Register.Set PC** *<value>* and **Register.Set SP** *<value>*. Here is an example of how to use these commands:

```
Register.Set PC 0xc000    ; Set the Program Counter to address 0xC000

Register.Set SP 0xbff     ; Set the Stack Pointer to address 0xbff

Register.Set PC main      ; Set the PC to a label (here: function main)
```

8. View the Source Code

Use the command **Data.List** to view the source code at the location of the Program Counter.

Now the quick start is done. If you were successful you can start to debug.

To reach the main() function use command **GO main**

10. Create a PRACTICE Script

LAUTERBACH recommends to prepare a PRACTICE script (*.cmm, ASCII file format) to be able to do all the necessary actions with only one command. Here is a typical start sequence:

```
WinClear                          ; Clear all windows

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

SYStem.CPU ARC700                 ; Use generic ARC700 core support.

System.JtagClock 5.MHz            ; Set JTAG clock speed.

SYStem.Up                         ; Reset the target and enter debug mode

Data.LOAD.Elf demo.elf            ; Load the application

Data.List                         ; Open disassembly window

Register.view                     ; Open register window

Var.Frame /Args /Locals           ; Show call stack

Var.Ref %HEX %DECIMAL             ; Auto-watch local variables

Break.Set 0x400                   ; Set software breakpoint on address
                                  ; 0x400

Break.Set main                    ; Set software breakpoint on address of
                                  ; main function.
```

For information about how to build a PRACTICE script file (*.cmm file), refer to **"Training Basic Debugging"** (training_debugger.pdf). There you can also find some information on basic actions with the debugger.

SETUP.DIS Disassembler configuration

Format:	**SETUP.DIS** [*<fields>* [*<bar>*]] [*<constants>*] [*<keywords>*]
<keywords>:	[**RegNames** | **Generic**] [**AddressOffset.auto** | **AddressOffset.Signed** | **AddressOffset.Unsinged**]

Sets **default values** for configuring the disassembler output of **newly opened windows**. Affected windows and commands are **List.Asm**, **Register.view**, and **Register.Set**.

The command does **not affect existing windows** containing disassembler output.

<fields>, *<bar>*, *<constants>*	For a description of the generic arguments, see **SETUP.DIS** in **general_ref_s.pdf**.
RegNames (default)	Use the *ABI* (application binary interface) naming scheme for the names of the ARC general purpose registers (e.g. “sp” instead of “r28” for the stack pointer.). This setting is equivalent with **SYStem.Option.RegNames ON**.
Generic	Use the *register number* (x0, x1, …, x31) naming scheme for the names of the ARC general purpose registers. (e.g. “r28” instead of “sp” for the stack pointer.). This setting is equivalent with **SYStem.Option.RegNames OFF**.
AddressOffset.auto (default)	Automatically choose a probably suitable format for the address offsets in load and store instructions. E.g.: For `LD <dst>,[<reg>,<offset>]` the *offset* is displayed as a signed number if the offset is smaller +/- 255 of if *reg* is gp/fp/sp/pcl, or as an unsigned hex-number otherwise.
AddressOff-set.Signed	Force the display of the address offsets in load and store instructions as signed. E.g.: For `LD <dst>,[<reg>,<offset>]` the *offset* is always displayed as a signed number.
AddressOff-set.Unsinged	Force the display of the address offsets in load and store instructions as unsigned. E.g.: For `LD <dst>,[<reg>,<offset>]` the *offset* is always displayed as a unsigned hexadecimal number.

SYStem.CONFIG.state — Display target configuration

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

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

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

<tab>	Opens the **SYStem.CONFIG.state** window on the specified tab. For tab descriptions, see below.
DebugPort	Informs the debugger about the debug connector type and the communication protocol it shall use.
Jtag	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.
MultiTap	Informs the debugger about the existence and type of a System/Chip Level Test Access Port. The debugger might need to control it in order to reconfigure the JTAG chain or to control power, clock, reset, and security of different chip components. For descriptions of the commands on the **MultiTap** tab, see **MultiTap**.
DAP	Informs the debugger about an ARM CoreSight Debug Access Port (DAP) and about how to control the DAP to access chip-internal memory busses (AHB, APB, AXI) or chip-internal JTAG interfaces.

COmponents	Informs the debugger about the existence and interconnection of on-chip **CoreSight** debug and trace modules and informs the debugger on which memory bus and at which base address the debugger can find the control registers of the modules. This is only relevant in case used ARC core is debugged over a **CoreSight DAP**. For descriptions of the commands on the **COmponents** tab, see **COmponents** in **"Arm Debugger"** (debugger_arm.pdf).

Format: **SYStem.CONFIG.***<sub_cmd>* *<parameter>*
SYStem.MultiCore.*<sub_cmd>* *<parameter>* (deprecated)

<sub_cmd>:
(DebugPort)

CONNECTOR [**MIPI34** | **MIPI20T**]
CORE *<core>* *<chip>*
CoreNumber *<number>*
DEBUGPORT [**DebugCable0** | **DebugCableA** | **DebugCableB**]
DEBUGPORTTYPE [**JTAG** | **SWD** | **CJTAG**]
Slave [**ON** | **OFF**]
TriState [**ON** | **OFF**]
MADI *<id>*
CJTAGFLAGS *<flags>*
CJTAGTCA *<value>*
SWDP [**ON** | **OFF**]
SWDPIdleHigh [**ON** | **OFF**]
SWDPTargetSel *<value>*

<sub_cmd>:
(JTAG)

DAPDRPOST *<bits>*
DAPDRPRE *<bits>*
DAPIRPOST *<bits>*
DAPIRPRE *<bits>*
DRPRE *<bits>*
DRPOST *<bits>*
IRPRE *<bits>*
IRPOST *<bits>*
Slave [**ON** | **OFF**]
TriState [**ON** | **OFF**]
TAPState *<state>*
TCKLevel [**0** | **1**]

<sub_cmd>:
(DAP)

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

<sub_cmd>:
(MultiTap)

MULTITAP [**NONE** | **PrimaryTAP** *<args>* | **JtagSEQuence.***<sub_cmd>*]

<parameter>: **(COmponents)**

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

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

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

<parameter>: (Components cont.)	**REP.ATBSource** *<source>* **REP.Base** *<address>* **REP.Name** *<string>* **REP.RESet** **REP.view** **TPIU.Base** *<address>* **TPIU.Name** *<string>* **TPIU.RESet** **TPIU.Type** [**CoreSight** | **Generic**] **TPIU.view**
<sub_cmd>: (misc)	**DEBUGTIMESCALE** *<multiplier>* **ADDRTICYCLES** *<dr>* *<ir>*

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

Some commands need a certain CPU type selection (**SYStem.CPU** *<type>*) for Lauterbach debug hardware to become active.

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**.

<parameters> describing the “DebugPort”

CJTAGFLAGS *<flags>*

Activates workarounds for incomplete or buggy cJTAG (IEEE 1149.7) implementations.
Bit 0: Disable scanning of cJTAG ID (TCA-scanning).
Bit 1: Target has no “keeper”. Use TRACE32 pseudo keeper.
Bit 2: Inverted meaning of SREDGE register.
Bit 3: Old command opcodes (cJTAG < 1.14).
Bit 4: APFC unlock required.
Bit 5: OAC required

Default: 0

CJTAGTCA *<value>*

Selects the TCA (TAP Controller Address) to address a device in a cJTAG (IEEE 1149.7) Star-2 configuration. The Star-2 configuration requires a unique TCA for each device on the debug port.

CONNECTOR [**MIPI34** | **MIPI20T**]

Specifies the connector “MIPI34” or “MIPI20T” on the target. This is mainly needed in order to notify the trace pin location.

This command is only available if the used Lauterbach debug cable supports different pin-outs. E.g. if a CombiProbe is used with a MIPI34 whisker.

Default: MIPI34.

CORE *<core> <chip>*

The command helps to identify cores which have 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 over the same PowerDebug.

All core which share the same debug resources should have the same *<chip>* number.

E.g.: If you SoC contains an ARConnect unit for inter-core communication and cross-triggering, all cores, which arc connected to the same ARConnect should have the same *<chip>* number. (Otherwise truly synchronous start and stop of the cores wouldn’t work)

This are the default settings of the command:
1st TRACE32 PowerView GUI: *<core>*=1 *<chip>*=1
2nd TRACE32 PowerView GUI: *<core>*=1 *<chip>*=2
...
n-th TRACE32 PowerView GUI: *<core>*=1 *<chip>*=*n*
This means that by default the cores are handled as unrelated.

CoreNumber *<number>*

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

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

In case you’re using a Lauterbach CombiProbe with two MIPI34 whiskers, this command allows to select the whisker cable, which should be used by the current TRACE32 PowerView GUI.

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.

- **JTAG**: Standard 5- or 4-pin JTAG (IEEE 1149.1)
- **CJTAG**: Compact 2-wire-JTAG (IEEE 1149.7)
- **SWD**: ARM Serial Wire Debug (requires that the ARC debug logic is connected to a CoreSight DAP inside the target SoC)
- **CJTAGSWD**: CJTAG in a mixed SWD/cJTAG configuration.

Default: JTAG

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

With this command you can change from the normal JTAG communication to Serial Wire Debug. SWD (Serial Wire Debug) uses just two signals instead of five. It works only if your target SoC contains a CoreSight DAP and the ARC core debug register are connected to that DAP (via APB or JTAG-AP)

If **SYStem.CONFIG.DEBUGPORTTYPE** is set to **CJTAGSWD**, this command selects if cJTAG or SWD should be used.

Default: OFF.

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

(default: OFF) If more than one debugger share the same JTAG port, all except one must have this option active. Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET).

Default: OFF for the first TRACE32 PowerView GUI connected a PowerDebug, ON for every further TRACE32 PowerView GUI connected to the same PowerDebug for AMP multicore debugging.

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: OFF.

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

TriState has to be set to ON 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. (Pull-down resistor on TCK is strongly recommended!)

Default: OFF.

MADI

Some chips with multiple ARC cores use the so-called Multiple ARCtangent Processor Debug Interface (MADI). MADI is a multiplexer which allows you to debug several ARC cores via one JTAG TAP.
While you select the TAP with IRPRE, IRPOST, DRPRE, DRPOST the MADI options tells the debugger which core connected to the MADI-TAP you want to debug.
Setting MADI to OFF means you don't have a MADI IP between your TAP and your core.
If your target system does not have MADI, MADI is set automatically to OFF.

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

With a 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.

To tell the debugger the exact position of your core’s Test Access Port controller (TAP) within a JTAG daisy-chain you’ll require the commands IRPRE, IRPOST, DRPRE, and DRPOST.

Most ARC cores are directly connected to JTAG Test Access Port controller (TAP), which is accessible directly to the debugger via JTAG. However, in case you’re ARC core is debugged via a CoreSight DAP you’ll need DAPDRPRE/POST and DAPIRPRE/POST below instead.

DRPRE *<bits>*
(default: 0) *<number>* of TAPs in the JTAG chain between the core of interest and the TDO signal of the debugger. If each core in the system contributes only one TAP to the JTAG chain, DRPRE is the number of cores between the core of interest and the TDO signal of the debugger.

DRPOST *<bits>*
(default: 0) *<number>* of TAPs in the JTAG chain between the TDI signal of the debugger and the core of interest. If each core in the system contributes only one TAP to the JTAG chain, DRPOST is the number of cores between the TDI signal of the debugger and the core of interest.

IRPRE *<bits>*
(default: 0) *<number>* of instruction register bits in the JTAG chain between the core of interest and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the core of interest and the TDO signal of the debugger.

IRPOST *<bits>*
(default: 0) *<number>* of instruction register bits in the JTAG chain between the TDI signal and the core of interest. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the core of interest.

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.SHOWChain** command.

If you JTAG daisy chain contains a CoreSight DAP and the DAP is accessible via JTAG the DAP's JTAG Test Access Port controller (TAP) may also be inside a JTAG daisy-chain. To tell the debugger the exact position DAP's TAP within the JTAG daisy-chain you'll require the commands DAPIRPRE, DAPIRPOST, DAPDRPRE, and DAPDRPOST. These settings are especially important if the CoreSight DAP is not just used to access memory, but also your ARC cores's debug registers are also accessed via the DAP.

DAPDRPOST *<bits>*
(default: 0) *<number>* of TAPs in the JTAG chain between the DAP and the TDO signal of the debugger.

DAPDRPRE *<bits>*
(default: 0) *<number>* of TAPs in the JTAG chain between the TDI signal of the debugger and the DAP.

DAPIRPOST *<bits>*
(default: 0) *<number>* of instruction register bits in the JTAG chain between the DAP and the TDO signal of the debugger. This is the sum of the instruction register length of all TAPs between the DAP and the TDO signal of the debugger.

DAPIRPRE *<bits>*
(default: 0) *<number>* of instruction register bits in the JTAG chain between the TDI signal and the DAP. This is the sum of the instruction register lengths of all TAPs between the TDI signal of the debugger and the DAP.

.

Slave [**ON** | **OFF**]
If more than one debugger share the same JTAG port, all except one must have this option active. Only one debugger - the "master" - is allowed to control the signals nTRST and nSRST (nRESET).

Default: OFF for the first TRACE32 PowerView GUI connected a PowerDebug, ON for every further TRACE32 PowerView GUI connected to the same PowerDebug for AMP multicore debugging.

TriState [**ON** | **OFF**]
TriState has to be set to ON 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. (Pull-down resistor on TCK is strongly recommended!)

Default: OFF.

TCKLevel *<level>*

Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target. (Pull-down resistor on TCK is strongly recommended!)

Default: 0.

TAPState *<state>*

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

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

Default: 7 = Select-DR-Scan.

MultiTap

MULTITAP [**NONE** | **PrimaryTAP** *<args>*]

For command descriptions, see **SYStem.CONFIG.MULTITAP**.

MULTITAP JtagSEQuence.*<sub_cmd>*

For command descriptions, see **SYStem.CONFIG.MULTITAP JtagSEQuence**.

<parameters> configuring a CoreSight Debug Access Port "DAP"

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

1. **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:. The interface to these buses is called Memory Access Port (MEM-AP).
   The debug registers of some cores are accessible via such a memory bus (mostly APB).
2. **Chip-internal JTAG interfaces**. This is 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

The following **SYStem.CONFIG** commands configure the port-number for the memory busses:

AHBACCESSPORT *<port>* — DAP access port number (0-255) which shall be used for "AHB:" access class. Default: *<port>*=0.

APBACCESSPORT *<port>* — DAP access port number (0-255) which shall be used for "APB:" access class. Default: *<port>*=1.

AXIACCESSPORT *<port>* — DAP access port number (0-255) which shall be used for "AXI:" access class. Default: port not available

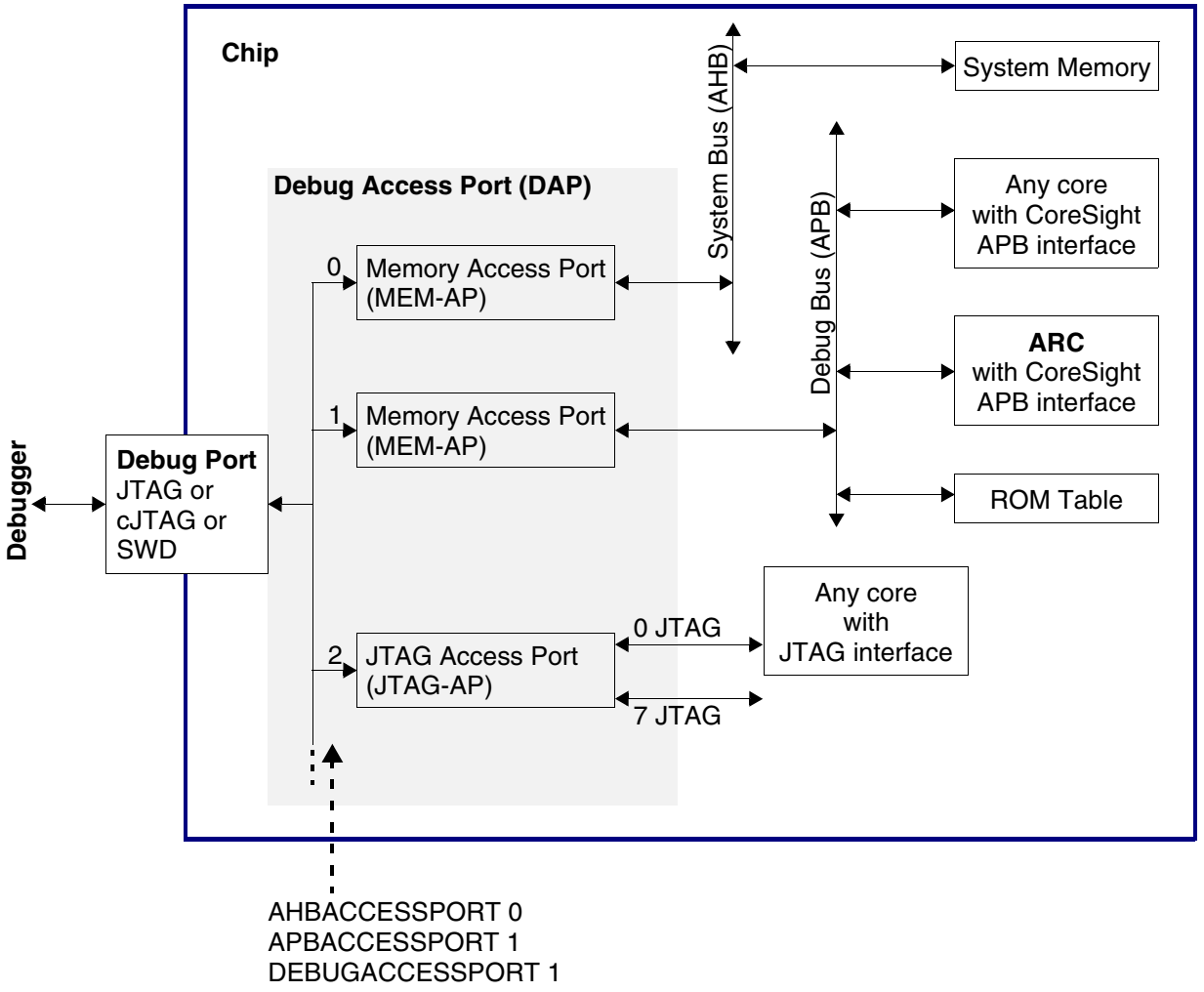
DEBUGACCESSPORT *<port>* — DAP access port number (0-255) where the debug register can be found (typically on APB). Used for "DAP:" access class. Default: *<port>*=1.

The following SYStem.CONFIG commands are required if your ARC core is connected to a chip-internal JTAG interface, which is controlled via the DAP:.

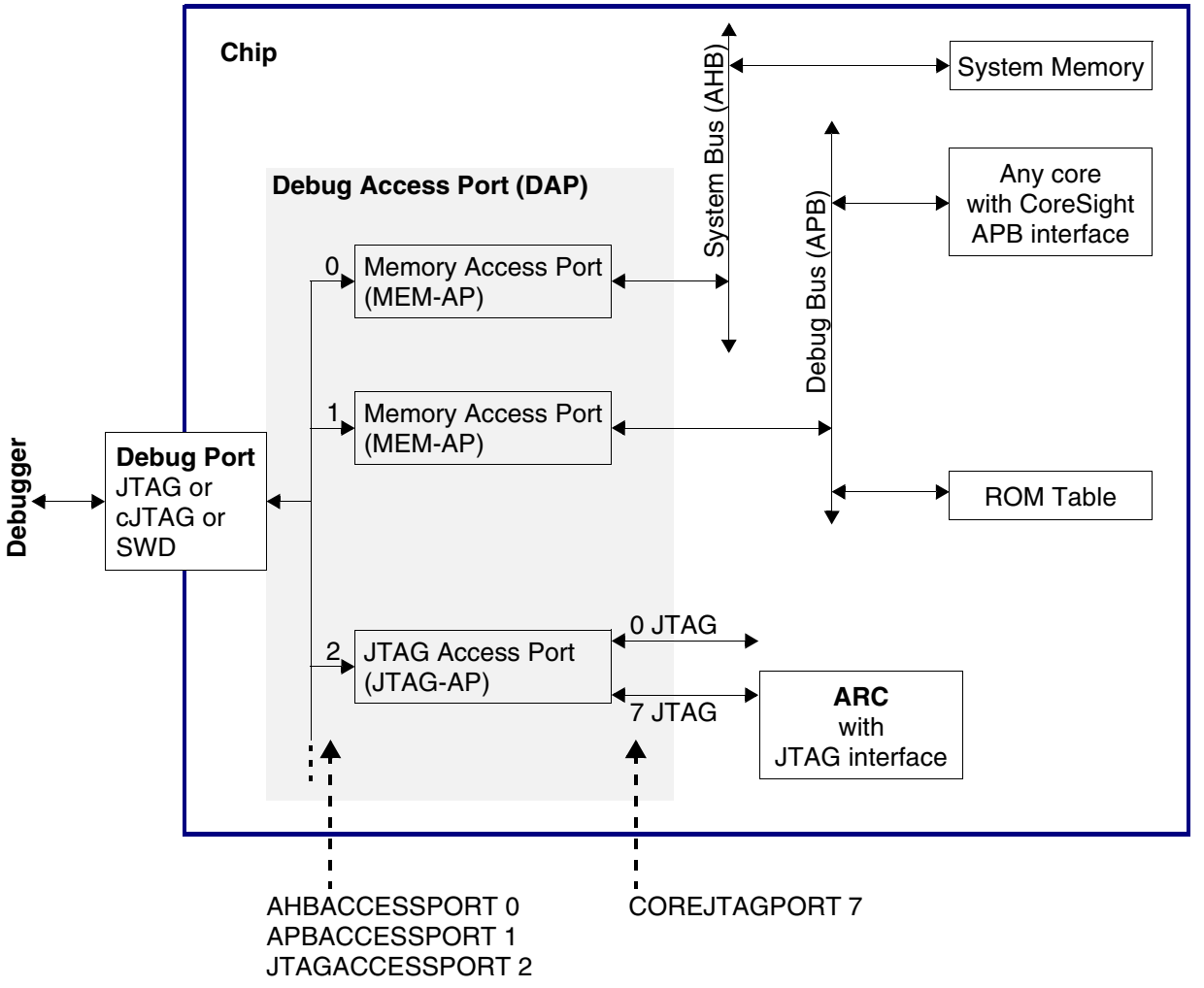
JTAGACCESSPORT *<port>* — DAP access port number (0-255) of the JTAG Access Port.

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

Example 1: ARC core debugged via CoreSight APB interface



Example 2: ARC core debugged via chip-internal JTAG interface controlled via the DAP:



<parameters> describing debug and trace "Components"

On the **COmponents** tab of 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.

… **.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.

… **.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.

... **.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.

… **.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.

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.
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.

ETB.Base *<address>*
ETB.Name *<string>*
ETB.NoFlush [**ON** \| **OFF**]
ETB.RESet
ETB.Size *<size>*
Embedded Trace Buffer (ETB) - ARM CoreSight module
Enables trace to be stored in a dedicated SRAM. The trace data will be read out through the debug port after the capturing has finished.

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

ETR.Base *<address>*
ETR.Name *<string>*
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.

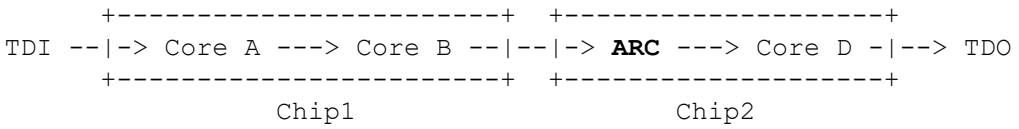
ETS.ATBSource *<source>*
ETS.Base *<address>*
ETS.Name *<string>*
ETS.RESet
Embedded Trace Streamer (ETS) - ARM CoreSight module

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.

TPIU.Base *<address>*
TPIU.Name *<string>*
TPIU.RESet
Trace Port Interface Unit (TPIU) - ARM CoreSight module
Trace sink sending the trace off-chip on a parallel trace port (chip pins).

Multicore with JTAG Daisy Chain



Instruction register length of

- Core A: 3 bit
- Core B: 5 bit
- Core D: 6 bit

```
SYStem.CONFIG.IRPRE 6             ; IR Core D
SYStem.CONFIG.IRPOST 8            ; IR Core A + B
SYStem.CONFIG.DRPRE 1             ; DR Core D
SYStem.CONFIG.DRPOST 2            ; DR Core A + B
SYStem.CONFIG.CORE 1. 2.          ; Core 1 in Chip 2
SYStem.Up
```

SMP multicore debugging of a quad-core ARC-HS

Setup for a ARC-HS quad core, which should debugged in SMP mode, which means that all 4 core are debugged via the same TRACE32 PowerView GUI. SMP is the right mode if all cores share the same memory and task. E.g. when Linux is running on the ARC quad core cluster.

In this example all ARC cores of the quad-core cluster have a separate JTAG-TAP, which is the most common configuration.

```
SYStem.CPU ARC-HS
SYStem.CONFIG CoreNumber 4
SYStem.CONFIG IRPRE  12.  8.  4.   0.
SYStem.CONFIG DRPRE   3.  2.  1.   0.
SYStem.CONFIG IRPOST  0.  8.  4.  12.
SYStem.CONFIG DRPOST  0.  1.  2.   3.
CORE.ASSIGN 1 2 3 4
SYStem.Up
```

```
      +-------------+
TDI --|---> MADI ---|--> TDO
      +----/-|-\----+
          /  |  \
         /   |   \
        /    |    \
    CoreA  CoreB  ARCtangent
```

```
SYStem.CONFIG MADI 2              ; Debug 3rd core attached to MADI
SYStem.Up
```

SYStem.CPU — Select CPU type

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

<cpu>:
AUTO |
ARCtangent-A4 |
ARCtangent-A5 |
ARC600 | **ARC601**
ARC700
ARC-EM | **ARC-EM-1r0** |
ARC-HS |
ARC-EV6x | **ARC-EV7x** | **ARC-VPX5** |

Default: AUTO.

Selects the processor type.

AUTO reads out the IDENTITY auxiliary register after a **SYStem.Up** or **SYStem.Mode Attach**, and sets the system CPU to the detected core accordingly.

Format:	**SYStem.JtagClock** [*<clockmode>*] *<frequency>*
<clockmode>:	**RTCK** | **ARTCK** | **CTCK** | **CRTCK** (*<clockmode>* only available if an ARM debug cable (e.g. LA-3750) is used.)
<frequency>:	**6 kHz** … **80 MHz** **1250000.** | **2500000.** | **5000000.** | **10000000.** (on obsolete ICD hardware) (*<frequency>* is optional if *<clockmode>* is set to RTCK)

Default frequency: 10 MHz.

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

<frequency> The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window.
Besides a decimal number like “100000.” short forms like “10kHz” or “15MHz” can also be used. The short forms imply a decimal value, although no “.” is used.

RTCK The JTAG clock is controlled by the RTCK signal (Returned TCK).
On some processor derivatives there is the need to synchronize the processor clock and the JTAG clock. In this case RTCK shall be selected. Synchronization is maintained, because the debugger does not progress to the next TCK edge until after an RTCK edge is received.
In case you have a processor derivative requiring a synchronization of the processor clock and the JTAG clock, but your target does not provide an RTCK signal, you need to select a fix JTAG clock below 2/3 of the processor clock.
When RTCK is selected, the frequency depends on the processor clock and on the propagation delays. The maximum reachable frequency is about 16 MHz.
Example: `SYStem.JtagClock RTCK`

ARTCK Accelerated method to control the JTAG clock by the RTCK signal (Accelerated Returned TCK).
Theoretically RTCK mode allows frequencies up to 2/3 of the processor clock.
For designs using a very low processor clock we offer a different mode (ARTCK) which might not work on all target systems. In ARTCK mode the debugger uses a fixed JTAG frequency for TCK, independent of the RTCK signal. TDI and TMS will be delayed by 1/2 TCK clock cycle. TDO will be sampled with RTCK

CTCK With this option higher JTAG speeds can be reached.
The TDO signal will be sampled by a signal which derives from TCK, but which is timely compensated regarding the debugger-internal driver propagation delays (Compensation by TCK). This feature can be used with a debug cable versions 3b or newer. If it is selected, although the debug cable is not suitable, a fix JTAG clock will be selected instead (minimum of 10 MHz and selected clock).

CRTCK With this option higher JTAG speeds can be reached, if your target provides RTCK.
The TDO signal will be sampled by the RTCK signal. This compensates the debugger-internal driver propagation delays, the delays on the cable and on the target (Compensation by RTCK). This feature requires that the target provides an RTCK signal. In contrast to the **RTCK** option, the TCK is always output with the selected, fixed frequency.

The modes RTCK, ARTCK, CRTCK can only be used if the target provides an RTCK signal.
Furthermore the modes are only available with the 20-pin Debug Cable LA-3750 (or LA-3750A in an ARM Debug Cable).

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

Default: OFF.

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

The process can also be automated, see **SYStem.CONFIG TriState**.
It must be ensured that the state of the 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.

SYStem.MemAccess Select run-time memory access method

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

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

Denied — Memory access during program execution to target is disabled.

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

This option declares if an **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.

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

If specific windows that display memory or variables should be updated while the program is running, select the memory class prefix **E:** or the format option **%E**.

```
Data.dump ED:0x100

Data.List EP:main

Var.View %E first
```

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
Attach
Up
Go
Prepare

Down The debug adapter gets tristated.
The state of the CPU remains unchanged. Debug mode is not active.
In this mode the target behaves as if the debugger is not connected.

If **SYStem.Option.EnReset.ON** is set to **ON** and **SYStem.CONFIG.Slave** is set to **OFF** the debugger will drive a low-active reset pulse on the nRESET (nSRST) on the JTAG connector.

NoDebug The debug adapter gets tristated.
The state of the CPU remains unchanged. Debug mode is not active.
In this mode the target behaves as if the debugger is not connected.

Attach Initializes the debug interface and connects to core while program remains running.

After this command the user program can be stopped with the break command or by any other break condition (e.g a breakpoints).

Up Initializes the debug interface, enters debug mode, stops the core and initializes several registers to their reset value. The debugger sets the program counter to the reset address of the core.

If **SYStem.Option.EnReset.ON** is set to **ON** and **SYStem.CONFIG.Slave** is set to **OFF** the debugger will drive a low-active reset pulse on the nRESET (nSRST) on the JTAG connector.

Go Start code execution from reset vector.
Actually the debugger performs the same actions than on **SYStem.Mode.Up** followed by **Go.direct**.

Prepare

Allow memory access without checking the go-stated of the core.

On **SoCs with CoreSight DAP** the debugger connects to the DAP and allows accesses to the APB, AHB and/or AXI bus - without communicating with the ARC core.

On **SoCs with direct JTAG connection** to the ARC core, memory can be accessed via the ARC core in prepare-mode.

StandBy

Not available for ARC.
(This mode is used to start debugging from power-on. The debugger will wait until power-on is detected, then initialize the debug interface and connect to core.)

NOTE:	**SYStem.Down** is an abbreviation for **SYStem.Mode Down**. **SYStem.Up** is an abbreviation for **SYStem.Mode Up**. **SYStem.Attach** is an abbreviation for **SYStem.Mode Attach**.

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

Set target-specific options, e.g. **SYStem.Option.Endianness** or **SYStem.Option.IMASKHLL**.
See the description of the available options below.

SYStem.Option.AHBHPROT Select AHB-AP HPROT bits

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

Default: 0

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

This option is only meaningful, if the chip contains a CoreSight DAP.

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

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

Default: OFF.

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

This option is only meaningful, if the chip contains a CoreSight DAP.

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

<value>:
DeviceSYStem
NonCacheableSYStem
ReadAllocateNonShareable
ReadAllocateInnerShareable
ReadAllocateOuterShareable
WriteAllocateNonShareable
WriteAllocateInnerShareable
WriteAllocateOuterShareable
ReadWriteAllocateNonShareable
ReadWriteAllocateInnerShareable
ReadWriteAllocateOuterShareable

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

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

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.

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

This option is only meaningful, if the chip contains a CoreSight DAP.

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

Default: 0

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

This option is only meaningful, if the chip contains a CoreSight DAP.

SYStem.Option.CorePowerDetection Set methods to detect core power

Format: **SYStem.Option.CorePowerDetection** *<method>*

<method>: **JtagSEQuence** *<seq_name>* | **none**

Sets and configures methods to detect the power of a core.

The core power is detected when **SYStem.Mode Up** is active or is entered. If a core is not powered, the debugger stays in system mode “Up” but displays the state “running (no power)” in the TRACE32 state line.

At the moment only the method **JtagSEQuence** is available.

JtagSEQuence *<seq_name>*	Enables the detection of the core power via a specified JTAG sequence. The specified JTAG sequence is periodically executed by the debug driver. You can create a JTAG sequence with the command **JTAG.SEQuence.Create**. The debug driver assumes that the core is powered when the JTAG sequence returns zero in the variable **Result0**. In case of an SMP system, use the environment variable **PhysicalCORE** within your JTAG sequence.
JtagSequence none	Disables the detection of the core power via a JTAG sequence.

Example:

```
SYStem.RESet ; resets SYStem settings (unlocks all used JTAG sequences)
SYStem.CPU ARC-HS

; create JTAG sequence for power detection
JTAG.SEQuence.Delete myCorePowerCheck   ; delete old sequence
JTAG.SEQuence.Create myCorePowerCheck   ; create new sequence
JTAG.SEQuence.Add , PrePostRelative  +4.  -4.  +1.  -1.
JTAG.SEQuence.Add , RawShift          4.  0x03  0x00
JTAG.SEQuence.Add , ShiftIrAndExit    4.  0x07
JTAG.SEQuence.Add , RawShift          4.  0x03  0x00
JTAG.SEQuence.Add , ShiftDrAndExit   16.  0x00  Result0
JTAG.SEQuence.Add , RawShift          2.  0x01  0x00
JTAG.SEQuence.Add , ASSIGN   Result0 =  ~ Result0 &  0x0001

; use the new JTAG sequence for detecting the core power
SYStem.Option.CorePowerDetection.JtagSEQuence myCorePowerCheck

; connect to all cores of the chip
SYStem.Mode Attach
```

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

Default: ON.

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

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

Use case:

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

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

This option is only meaningful, if the chip contains a CoreSight DAP.

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

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

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

This option is only meaningful, if the chip contains a CoreSight DAP.

SYStem.Option.DAPSYSPWRUPREQ Force system power in DAP

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

Default: ON.

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

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

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

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

This option is only meaningful, if the chip contains a CoreSight DAP.

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

Default: OFF.

Bug fix for derivatives which do not return the correct pattern on a DAP (Arm CoreSight Debug Access Port) instruction register (IR) scan. When activated, the returned pattern will not be checked by the debugger.

This option is only meaningful, if the chip contains a CoreSight DAP.

SYStem.Option.DCFLUSH Invalidate/flush data-cache for modified memory

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

Default: ON.

If the target memory is modified via the debugger, this option ensures that the data cache (and the 2nd level cache (SLC) if available) gets invalidated before the target CPU is restarted. Furthermore, the data cache gets flushed when the CPU stops.

If the option is disabled, the debugger checks for every target memory access if the data is cached. If so the debugger reads the data from the cache on a read access or writes the data separately to both target memory and the cache on a write access.

The disabled option allows to do small modifications in the target memory without loosing the content of the cache, while the enabled option ensures that the physical memory contains the latest changes by the CPU after it stops, allows faster memory accesses by the debugger, and guarantees that there are no artifacts left in the cache when re-starting the core.

SYStem.Option.DEBUGPORTOptions Options for debug port handling

Format: **SYStem.Option.DEBUGPORTOptions** *<option>*

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

Default: SWITCHTOSWD.TryAll, SWDTRSTKEEP.DEFault.

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

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

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

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

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

This option is only meaningful, if the chip contains a CoreSight DAP.

SYStem.Option.detectOTrace Disable auto-detection of on-chip trace

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

Default: OFF.

When connecting the debugger to the ARC core via commands **SYStem.Mode Attach** or **SYStem.Mode Up** the debugger tries to detect if the ARC on-chip trace (SmaRT) by reading auxiliary register 255 (AUX:0xFF).
For some reason some rare core implementations without SmaRT seem to have a fatal side-effect on AUX:0xFF. For these cores use this option to avoid the read of AUX:0xFF during **SYStem.Mode Attach** or **SYStem.Mode Up**.

SYStem.Option.Endianness — Set the target endianness

Format: **SYStem.Option.Endianness** [**Big** | **Little** | **AUTO**]

Default: AUTO.

This option selects the target byte ordering mechanism (endianness). It effects the way data is read from or written to the target CPU.

In AUTO mode the debugger sets the endianness corresponding to the “ARC Build Registers”, when the debugger is attached to the target. AUTO mode is not available for ARCtangent-A4 cores.

Consider that the compiler, the ARC core and the debugger should all use the same endianness.

SYStem.Option.EnReset — Allow the debugger to drive nRESET (nSRST)

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

Default: OFF.

If this option is set to ON, the debugger will drive a low-active reset pulse on the nRESET (nSRST) line on the JTAG connector on **SYStem.Up** and **SYStem.Down**.

From the view of the core, it is not necessary that nRESET (nSRST) becomes active at the start of a debug session (**SYStem.Up**), but there may be other logic on the target which requires a reset.

If **SYStem.CONFIG.Slave** or **SYStem.Opiton.IntelSOC** is set to **ON**, the debugger will never drive the nRESET (nSRST), independently from **SYStem.Option.EnReset**.

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

Default: AUTO.

This option controls how software breakpoints are set to a running ARC core:

ON The debugger tries to set a software breakpoint while the CPU is running, if **SYStem.MemAccess** is set to **CPU**.

OFF To set a software breakpoint, the debugger tries to stop the CPU temporarily, if **SYStem.CpuAccess** is set to **ENABLED**.

AUTO To set a software breakpoint, the debugger stops the CPU temporarily if the CPU has an Instruction Cache (requires **SYStem.CpuAccess** set to **ENABLED**) otherwise the debugger tries to set a software breakpoint while the CPU is running (requires **SYStem.MemAccess** set to **CPU**).

SYStem.Option.ICFLUSH Invalidate instruction-cache for modified memory

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

Default: ON.

If the target memory is modified via the debugger, this option ensures that the instruction cache (and the 2nd level cache (SLC) if available) gets invalidated before the target CPU is restarted.

If the option is disabled, the debugger tries to write any modification on the target memory also separately to the instruction cache.

The disabled option allows to do small modifications in the target memory without loosing the content of the cache, while the enabled option allows faster memory and guarantees that there are no artiffacts left in the cache when re-starting the CPU.

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 **IMASKASM**.

SYStem.Option.IntelSOC Core is part of Intel® SoC

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

Default: OFF.

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

Requires that the ARC debugger is slave in a multicore setup. The master of the multicore setup must be "TRACE32 for x86" with **SYStem.Option.CLTAPOnly** enabled.

<soc_id> An integer ID used by TRACE32 to identify a specific core in an SOC if there is more than one core of the same type. This ID is platform specific. For more details, see **"Slave Core Debugging"** in Intel® x86/x64 Debugger, page 34 (debugger_x86.pdf).
Default: 0.

SYStem.Option.LimmBreakPoints Software breakpoints with extra NOPs

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

Default: OFF.

Any ARC instruction set allows instructions with so-called Long Immediate Data (LIMM). These instructions have a total length of 6 or 8 bytes. When setting a software breakpoint the instruction at the address of the software breakpoints gets replaced by a BRK or BRK_S instruction. The BRK instruction has a length of 4 byte and the BRK_S has a length of 2 bytes. When **SYStem.Option.LimmBreakPoints** is set to ON the remaining 2 or 4 bytes of a LIMM instruction are overwritten with NOP_S instructions when setting a software breakpoint on them.

This option helps to workaround a buggy implementation of an ARC core.

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.OVERLAY — Enable overlay support

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

Default: OFF.

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

OFF — Disables support for code overlays.

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

Example:

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

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

Default: ON.

This option just effects the way core registers are displayed e.g. in the **Register.view** window or in disassembled memory. If the option is enabled some core registers are displayed by their trivial names describing the registers function e.g. “blink” for core register 31. When disabled the systematic name is used corresponding tho the register number e.g. “r31” for core register 31.

SYStem.Option.PowerDetection Choose method to detect the target power

Format: **SYStem.Option.PowerDetection** *<method>*

<method>: **VTREF**
PowerGood

Default: VTREF.

Selects the method how the debugger detects if the target system is powered.

VTREF The debugger uses the VTREF line on the debug **connector** to check if the target system is powered.

PowerGood The debugger uses the PWRGOOD line on the debug **connector** to check if the target system is powered.
This setting is only available if the used debug probe supports the PowerGood signal.

Format: **SYStem.Option.ResetDetection** *<method>*

<method>: **Sem0** | **Sem1** | **Sem2** | **Sem3**
RAbit
nSRST
None

Default: Sem1.

Selects the method how an external target reset can be detected by the debugger.

Sem0 … **Sem3** Detects a reset if corresponding semaphore bit in the SEMAPHORE auxiliary register (AUX:0x01) is set to zero. This option is not available on ARC700 and ARC-EM cores, since these cores do not have a SEMAPHORE register. (This method detects “core resets”.)

RAbit Detects a reset by checking the RA-bit in the JTAG status register or DEBUG auxiliary Register (AUX:0x05). This option is only available for ARC cores with JTAG version 2 or higher. (This detects core resets.) (This method detects “core resets”.)

nSRST Detects a reset if nSRTS line on the debug **connector** is pulled low. (This method detects a “chip resets” or a complete “target resets”.) Furthermore by enabling this option the debugger will actively pull down the nSRST line while in system-down state and when going to system-up state and.

None Detection of external resets is disabled.

SYStem.Option.TIMEOUT Define maximum time for core response

Format: **SYStem.Option.TIMEOUT** *<time>*

Default: 1000.ms

After each JTAG transaction the debugger has to wait until the ARC core acknowledges the successful transaction.

With this option you can specify how long the debugger waits until the debugger has to assume that the core does no longer respond. You have to use this option only if you what to debug a unusual slow core.

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

Default: ON.

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

SYStem.POWER Control target power

Format: **SYStem.POWER** [**ON** | **OFF** | **CYCLE**]

This command requires a MIPI60 debug probe (e.g. CombiProbe with MIPI60 whisker).

If supported by the target and the used debug probe, this command turns the target power ON (if off) or OFF (if on), or does a power CYCLE (if on) via the PowerGood signal (pin#42 on MIPI60 connector).

SYStem.state Show SYStem settings window

Format: **SYStem.state**

Opens a window which enables you to view and modify CPU specific system settings.

On-chip Breakpoints/Actionpoints

"On-chip Breakpoints" and "Actionpoints" are two names for the same thing: A mechanism provided by the on-chip debug logic to stop the core when an instruction is fetched form a specific address or data is read from or written to a specific memory location. This enables you to set breakpoints even if your not able to modify the code on the fly e.g. in a Read Only Memory.

"Actionpoints" is the name used by Synopsys in the ARC manuals, while "On-chip Breakpoints" is the generic name used by Lauterbach. In the rest of the documentation we'll speak only about "On-chip Breakpoints".

An ARC core can have 2, 4, 8 or none on-chip breakpoints. The debugger detects the number of available breakpoints after you've connected to your target CPU with **SYStem.Up** or **SYStem.Mode Attach**. To find out how many on-chip breakpoints are available execute **PER.view, "Build"** and check the value at "AP_BUILD".

Using On-chip Breakpoints

See chapters **Break** and **On-chip Breakpoints** in the "General Commands Reference Guide B". .

When a read or write breakpoint triggers, any ARC CPU stops with an additional delay after the instructions, which causes the trigger.
The delay is 1 cycle for ARC700 and 3 cycles for ARC600. For memory reads there is an extra delay corresponding to the memory latency.
(However program breakpoints always stop before executing the instruction.)

On ARC600 you can set on-chip breakpoints only when the core is stopped. You can set **SYStem.CpuAccess** to **Enable** to allow the debugger to stop and restart the core to set on-chip breakpoints.

On ARC700 you can set on-chip breakpoints also while the core is running, when you've set **SYStem.MemAccess** to **CPU**.

Breakpoints in a ROM Area

With the command **MAP.BOnchip** *<range>* it is possible to tell the debugger where you have ROM / FLASH on the target. If a breakpoint is set into a location mapped as BOnchip, it gets automatically implemented as an on-chip breakpoint.

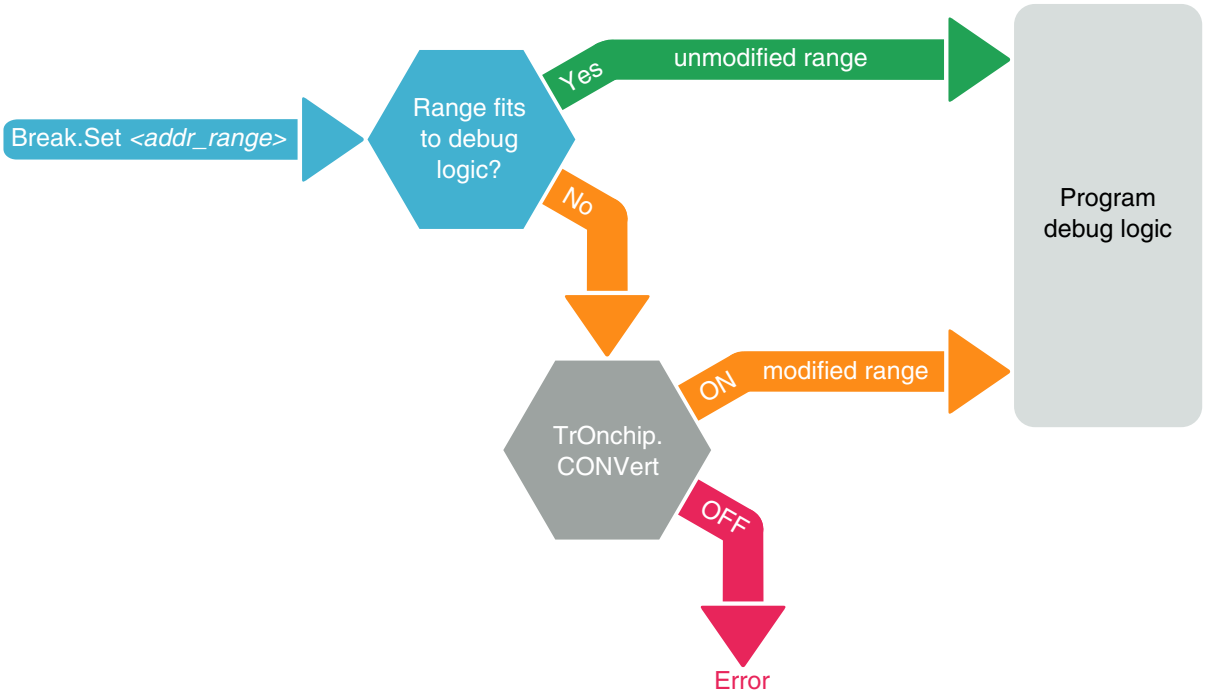
Limitations

Due to limitations in the ARC core logic, some common features for on-chip breakpoint are not available.

- ARC600 and ARC700 cores do not provide resources to set on-chip breakpoints for arbitrary address or data ranges. Instead they use bit masks. If a given range can't be programmed with a bit mask, the next larger range will be used, if **TrOnchip.CONVert** is active.
  You can check the address ranges actually set by the debugger inside the **Break.List /Onchip** window.
- While normal read breakpoints are available, which stop the core on the read of a given address, so-called "read *data* breakpoints" area not available. So you can't stop the core, when specific data is read from a given address.
  ("Write data breakpoints" are available.)
- On ARC700 you can use "Write data breakpoints" together with address ranges only for 32-bit wide data.
- For ARC600 using on-chip program breakpoints together with instruction data is not supported, since the on-chip logic of an ARC600 does not align the fetched instruction before comparing it to the value, which make this feature useless.
- On ARC600 you can't set on-chip breakpoints, while the core is running.

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

Controls for all on-chip read/write breakpoints whether the debugger is allowed to change the user-defined address range of a breakpoint (see **Break.Set** *<address_range>* in the figure below).



The debug logic of a processor may be implemented in one of the following three ways:

1. The debug logic does not allow to set range breakpoints, but only single address breakpoints. Consequently the debugger cannot set range breakpoints and returns an error message.
2. The debugger can set any user-defined range breakpoint because the debug logic accepts this range breakpoint.
3. The debug logic accepts only certain range breakpoints. The debugger calculates the range that comes closest to the user-defined breakpoint range (see “modified range” in the figure above).

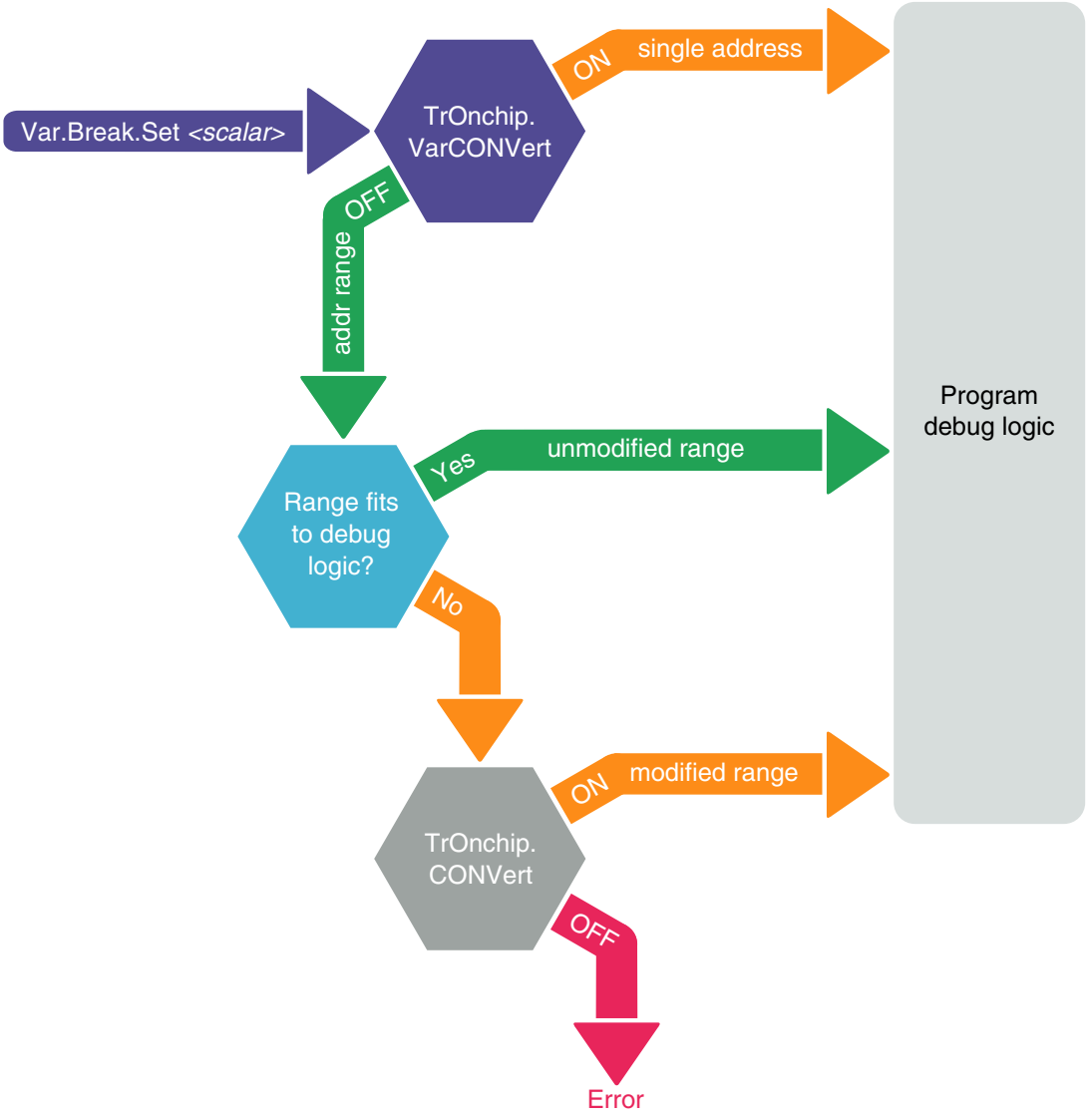
The **TrOnchip.CONVert** command covers case 3. For case 3) the user may decide whether the debugger is allowed to change the user-defined address range of a breakpoint or not by setting **TrOnchip.CONVert** to **ON** or **OFF**.

ON (default)	If **TrOnchip.Convert** is set to **ON** and a breakpoint is set to a range which cannot be exactly implemented, this range is automatically extended to the next possible range. In most cases, the breakpoint now marks a wider address range (see “modified range” in the figure above).
OFF	If **TrOnchip.Convert** is set to **OFF**, the debugger will only accept breakpoints which exactly fit to the debug logic (see “unmodified range” in the figure above). If the user enters an address range that does not fit to the debug logic, an error will be returned by the debugger.

In the **Break.List** window, you can view the requested address range for all breakpoints, whereas in the **Break.List /Onchip** window you can view the actual address range used for the on-chip breakpoints.

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

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



ON	If **TrOnchip.VarCONVert** is set to **ON** and a breakpoint is set to a scalar variable (int, float, double), then the breakpoint is set only to the start address of the scalar variable. • Allocates only one single on-chip breakpoint resource. • Program will not stop on accesses to the variable's address space.
OFF (default)	If **TrOnchip.VarCONVert** is set to **OFF** and a breakpoint is set to a scalar variable (int, float, double), then the breakpoint is set to the entire address range that stores the scalar variable value. • The program execution stops also on any unintentional accesses to the variable's address space. • Allocates **up to two** on-chip breakpoint resources for a single range breakpoint. **NOTE**: The address range of the scalar variable may not fit to the debug logic and has to be converted by the debugger, see **TrOnchip.CONVert**.

In the **Break.List** window, you can view the requested address range for all breakpoints, whereas in the **Break.List /Onchip** window you can view the actual address range used for the on-chip breakpoints.

TrOnchip.OnchipBP Number of on-chip breakpoints used by debugger

Format: **TrOnchip.OnchipBP** [*<number>* | **AUTO**]

Default: AUTO.

An ARC core has between 0 and 8 on-chip breakpoint resources (Called "Actionpoints" in the ARC core documentation). These resources are normally completely controlled by the debugger and are modified e.g. when you set on-chip breakpoints e.g. via **Break.Set** *<address>* **/Onchip /Write.**

Sometimes you might want to control the breakpoint resources (AUX:0x220--0x237) or parts of it by you own. With **TrOnchip.OnchipBP** you can tell the debugger how many on-chip breakpoint registers the debugger may control, leaving the rest of them untouched.

E.g.: If you have an ARC core with 4 on-chip breakpoints but you want control one breakpoint by your own, set **TrOnchip.OnchipBP** to 3. The registers you can control then by your own are those of the fourth breakpoint (AUX:0x229--0x22b).

NOTE:	This option is only for advanced users which have a good knowledge of the Actionpoint Auxiliary Registers described in the ARC600 Ancillary Components Reference or the ARC700 System Components Reference.

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

Format: **TrOnchip.RESet**

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

TrOnchip.state — Display on-chip trigger window

Format: **TrOnchip.state**

Opens the **TrOnchip.state** window.

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**.

CPU specific Tables in MMU.DUMP <table>

ITLB	Displays the contents of the Instruction Translation Lookaside Buffer.
DTLB	Displays the contents of the Data Translation Lookaside Buffer.
TLB	Displays the contents of the Translation Lookaside Buffer.
TLB0	Displays the contents of the Translation Lookaside Buffer 0.
STLB	Displays the contents of the STLB.

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**.

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**] *<cpu_specific_tables>*

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.

MMU.Init — Invalidate TLB entries

Format: **MMU.Init TLB** | **STLB**

Invalidates all entries of the given TLB.

MMU.Set — Set an MMU TLB entry

Format: **MMU.Set** *<tlb> <index> <pd0> <pd1>*

<tlb> **TLB** | **STLB**

Sets the specified MMU TLB entry.

JTAG.CONFIG Electrical characteristics of MIPI-60 debug signals

Using the **JTAG.CONFIG** command group, you can change electrical characteristics of MIPI-60 debug signals to account for target irregularities.

Availability of these commands is dependent on the used Lauterbach debug probe (e.g. available with a CombiProbe with MIPI60-Cv2 whisker).

JTAG.CONFIG.DRiVer Set slew rate of JTAG signals

Format:	**JTAG.CONFIG.DRiVer.**<*signal*> **Fast** | **Slow** [**/**<*whisker*>]
<*signal*>:	**all** | **TCK** | **TCK0** | **TCK1** | **TMS** | **TDI** | **nTRST** | **nPREQ**
<*whisker*>:	**WhiskerA** | **WhiskerB** | **WhiskerC** | **WhiskerD**

Selects whether to use a series inductor to slow the slew rate of output signals.

all	Set rate for all relevant signals.
TCK **TCK0** **TCK1** **TMS** **TDI** **nTRST** **nPREQ**	Set rate only for selected signal.
FAST	Use direct drive of selected signals.
SLOW	Insert inductor on drive of selected signals to limit voltage change rate.

NOTE:	With a CombiProbe and MIPI60-Cv2 whisker only **WhiskerA** is available. With a CombiProbe and MIPI60-C(v1) whisker this configuration command is not available.

Format:	**JTAG.CONFIG.PowerDownTriState ON** | **OFF** [*/<whisker>*]
<whisker>:	**WhiskerA** | **WhiskerB** | **WhiskerC** | **WhiskerD**

Default: ON.

Enables or disables the automatic setting of all signals to tristate when a power down state of the target is detected.

JTAG.CONFIG.TckRun Free-running TCK mode

[build 143356 - DVD 09/2022]

Format:	**command.subcommand OFF** | **TCK0** | **TCK1** [*/<whisker>*]
<whisker>:	**WhiskerA** | **WhiskerB** | **WhiskerC** | **WhiskerD**

Default: OFF.

Enables free-running TCK mode for the respective TCK signal.

JTAG.CONFIG.TDOEdge Select TCK edge

Format:	**JTAG.CONFIG.TDOEdge Rising** | **Falling** [*/<whisker>*]
<whisker>:	**WhiskerA** | **WhiskerB** | **WhiskerC** | **WhiskerD**

Default: RISING

Selects which edge of TCK signal is used for reading TDO.

Format:	**JTAG.CONFIG.Voltage.HooKTHreshold.***<signal>* *<voltage>* [**/***<whisker>*]
<signal>:	**all** | **Hook0** | **Hook6** | **Hook8** | **Hook9**
<whisker>:	**WhiskerA** | **WhiskerB** | **WhiskerC** | **WhiskerD**

Default: JTAG.CONFIG.Voltage.HooKTHreshold.all 0.6.

Sets voltage threshold to use for determining active state for selected Hook signals.

all	Set threshold for all Hook input signals.
Hook0 **Hook6** **Hook8** **Hook9**	Set threshold for selected Hook input signal only. (Not available for MIPI60-C(v1) whisker.)
<voltage>	Float value in volts to use as threshold.

NOTE:	With a CombiProbe and any MIPI60 whisker only **WhiskerA** is available. With a CombiProbe and MIPI60-C(v1) whisker only the signal **all** is possible

JTAG.CONFIG.Voltage.THreshold Set JTAG threshold voltages

Format:	**JTAG.CONFIG.Voltage.THreshold.***<signal>* *<level>* [**/***<whisker>*]
<signal>:	**all** | **TDO** | **PRDY**
<level>:	*<voltage>* | **AUTO**
<whisker>:	**WhiskerA** | **WhiskerB** | **WhiskerC** | **WhiskerD**

Default: JTAG.CONFIG.Voltage.THreshold.all AUTO.

Set voltage threshold to use for determining active state for selected JTAG signals.

all	Set threshold for TDO and PRDY. (Sets also voltage threshold for VTREF on a MIPI60-C(v1) whisker.)
TDO **PRDY**	Set threshold for only selected signal. (Not available for MIPI60-C(v1) whisker.)
AUTO	Use threshold derived from reference voltage.
<voltage>	Value in volts to use as threshold.

NOTE:	With a CombiProbe and any MIPI60 whisker only **WhiskerA** is available. With a CombiProbe and MIPI60-C(v1) whisker only signal **all** is possible

JTAG.CONFIG.Voltage.REFerence Voltage level of signals send to target

Format: **JTAG.CONFIG.Voltage.REFerence** *<voltage>* | **AUTO**

Default: JTAG.CONFIG.Voltage.REFerence AUTO.

Selects voltage level with which all signals from the debug probe to the target system are driven.

Setting a too high voltage level may damage you target hardware!
Don't use this command unless you know, which voltage levels can be handled by your CPU device.

<voltage>	Use specified value in volts as reference voltage.
AUTO	Output voltage set by measuring reference voltage supplied from target system.

NOTE:	With a CombiProbe and MIPI60-Cv2 whisker only **WhiskerA** is available. With a CombiProbe and MIPI60-C(v1) whisker this command is not available.

NEXUS.AuxTM — Enable auxiliary register trace messages

Format: **NEXUS.AuxTM** [**OFF** | **Read** | **Write** | **ReadWrite**]

Globally enables data trace for accesses to the core's auxiliary register.

This feature is only available for DesignWare ARC Trace with producer-type "full".

OFF	No data trace for auxiliary register (default).
Read	Data trace messages for read accesses on auxiliary registers (load instructions).
Write	Data trace messages for write accesses on auxiliary registers (store instructions).
ReadWrite	Data trace messages for read or write accesses on auxiliary registers (load and store instructions).

NEXUS.BTM — Enable program trace messaging

Format: **NEXUS.BTM** [**ON** | **OFF**]

Control for NEXUS program trace messaging.

ON (default)	Program trace messaging enabled.
OFF	Program trace messaging disabled.

Format: **NEXUS.CLOCK** *<frequency>*
NEXUS.CLOCK *<frequency0><frequency1>...* (SMP tracing only)

Sets core clock to calculate elapsed time based on the cycle count information, which is emitted when **NEXUS.TImeMode** is set to a mode using “NexusTimeStamps”.

This command is an alias to command **Trace.CLOCK**.

NEXUS.DataSuppress Suppress data flow on likely FIFO overflow

Format: **NEXUS.DataSuppress** [**ON** | **OFF**]

Default: OFF.

Allows DesignWare ARC Trace to suppress any data trace if a chip internal FIFO overflow is likely to happen.

This feature is only available when DesignWare ARC Trace was build with a CoreSight compatible (ATB) offload interface.

NEXUS.DDR Enable NEXUS double data rate mode

Format: **NEXUS.DDR** [**ON** | **OFF**]

Default: OFF.

Sets trace port and NEXUS adapter to operate in DDR (double data rate) mode, which means that trace data is emitted on the Nexus auxiliary port on both rising and falling edge of the trace clock.

This feature is only available when DesignWare ARC Trace was build with a Nexus auxiliary port (Nexus offchip-trace).

Format: **NEXUS.DSM** [**ON** | **OFF**]

Enables debug status messages, which get admitted when the trace starts and stops.

ON	Debug status messages enabled (default & recommended).
OFF	Debug status messages disabled.

NEXUS.DTM Enable data trace messages

Format: **NEXUS.DTM** [**OFF** | **Read** | **Write** | **ReadWrite**]

Globally enables data trace for memory accesses performed by the core.

This feature is not available for DesignWare ARC Trace with producer-type “small”.

OFF	No data trace for memory accesses (default).
Read	Data trace messages for read accesses on the memory (load instructions).
Write	Data trace messages for write accesses on the memory(store instructions).
ReadWrite	Data trace messages for read or write accesses on the memory (load and store instructions).

NEXUS.FILTER.ACompLimit Trace address filters used by debugger

Format: **NEXUS.FILTER.ACompLimit** *<number>* | **AUTO**

Default: AUTO.

Depending on its build configuration DesignWare ARC Trace has between 0 and 8 on-chip address filter resources. These resources are normally completely controlled by the debugger and are modified when you configure a trace filter via the **Break.Set** commands.

Example:

```
Break.Set <address-range> / TraceEnable
```

Sometimes you might want to control the address filter resources (or parts of it) by yourself.

With **NEXUS.FILTER.AComplimit** you can tell the debugger how many address filter the debugger may control, leaving the rest of them untouched.

Setting AUTO means, that the debugger may use all available address filters.

This is an advanced feature for users which have a very good knowledge of DesignWare ARC Trace.

NEXUS.FILTER.DCompLimit Number of trace data filter used by debugger

Format: **NEXUS.FILTER.DCompLimit** *<number>* | **AUTO**

Default: AUTO.

Depending on its build configuration DesignWare ARC Trace has between 0 and 2 on-chip data filter resources. These resources are normally completely controlled by the debugger and are modified when you configure a trace filter via the **Break.Set** commands.

Example:

```
Break.Set D:0x0--0xffffffff /TraceData /DATA.Long 0x42
```

Sometimes you might want to control the data filter resources (or parts of it) by yourself.

With **NEXUS.FILTER.DCompLimit** you can tell the debugger how many data filter the debugger may control, leaving the rest of them untouched.

Setting AUTO means, that the debugger may use all available data filters.

This is an advanced feature for users which have a very good knowledge of DesignWare ARC Trace.

NEXUS.HISToryTHreshold Control the conditional history threshold

Format: **NEXUS.HISToryTHreshold** [*<number>*]

Default: 0 (RFM is generated every 29th or 28th branches).

Advanced option to configure the Conditional History Threshold of the DesignWare ARC Trace, which specifies how often a Resource Full Message is emitted.

For more details look in the “DesignWare ARC Trace Databook” for the CHTH field of the DSEN register.

This feature is only available for DesignWare ARC Trace version 5 and higher.

This is an advanced feature for users which have a very good knowledge of DesignWare ARC Trace.

NEXUS.OFF Switch the NEXUS trace port off

Format: **NEXUS.OFF**

Disables the usage of DesignWare ARC Trace.

NEXUS.ON Switch the NEXUS trace port on

Format: **NEXUS.ON**

Enables the usages of DesignWare ARC Trace. All trace registers are configured by debugger.

Format: **NEXUS.PortMode 1/1 | 1/2 | 1/4 | 1/8**

Sets the NEXUS trace port frequency. For parallel NEXUS, the setting is the system clock divider. For Aurora NEXUS, the setting is a fixed bit clock which is independent of the system frequency.

NEXUS.Register Display NEXUS trace control registers

Format: **NEXUS.Register**

Opens a window which shows all registers related to DesignWare ARC Trace.

The registers are usually controlled by debugger and thus, manual changes get overwritten without notification.

NEXUS.RegTM Enable core register trace messages

Format: **NEXUS.RegTM** [**OFF** | **Write**]

Enables data trace for write accesses to core register.

This feature is only available for DesignWare ARC Trace with producer-type “full”.

OFF No data trace for auxiliary register (default).

Write Data trace messages for write accesses on auxiliary registers (store instructions).

NEXUS.RESet Reset NEXUS settings

Format: **NEXUS.RESet**

Resets all settings of the NEXUS command group to its default values.

NEXUS.RTTBUILD Define build configuration of used DesignWare trace

Format: **NEXUS.RTTBUILD** *<rtt_bcr>*

Advanced option to define the build configuration of the used DesignWare ARC Trace when decoding trace in the TRACE32 Instruction Set Simulator. The command is locked when the debugger is connected to the ARC core (**SYStem.Up()**==TRUE()).

For the meaning to the 32-bit value "rtt_bcr" look in the "DesignWare ARC Trace Databook" for the Build Configuration Register (RTT_BCR).

This is an advanced feature which is only required for belated trace decoding.

NEXUS.STALL Stall program execution when FIFO full

Format: **NEXUS.STALL** [**ON** | **OFF**]

Default: OFF.

Stalls the DesignWare ARC processor core when the output FIFO of the DesignWare ARC Trace is full.

If enabled, gaps in the program trace recording are avoided at the cost of a strong negative impact on the core performance.

NEXUS.state Display NEXUS port configuration dialog

Format: **NEXUS.state**

Displays a dialog window to configure the DesignWare ARC Trace.

NEXUS.SyncFrame Control SYNC frame insertion in ATB stream

Format: **NEXUS.SyncFrame** *<number>*

Default: 1024.

Advanced option to configure the amount of SYNC frames inserted in the CoreSight ATB stream. SYNC frames are required to extract the Nexus beats from the byte oriented ATB stream.

For more details look in the “DesignWare ARC Trace Databook” for the SYNCFR register.

This feature is only available when DesignWare ARC Trace was build a CoreSight compatible (ATB) offload interface.

This is an advanced feature for users which have a very good knowledge of DesignWare ARC Trace.

NEXUS.TImeMode Select method of time measurement

Format:	**NEXUS.TImeMode** *<mode>*
mode>:	**OFF** | **External** | **TimeStamps** | **NexusTimeStamps** | **NexusTimeStamps+External** | **NexusTimeStamps+ExternalTrack** | **NexusTimeStamps+TimeStamps**

The command **NEXUS.TImeMode** allows to choose the method, which is used to determine the elapsed time in the trace recording.

OFF No time measurement.

External Time measurement based on external timestamps added by the PowerTrace hardware. Only available with offchip-trace.
Pro: No extra bandwith is required for the timestamp. Allows timing correlation among cores simultaneously traced with offchip-trace.
Con: Timestamp might be imprecise due to delays caused by the trace infrastructure of the chip and due to dalays caused by the TRACE32 recording technology.

TimeStamps Time measurement based on separate timestamp messages fed by a global timestamp counter inside the chip. Command **NEXUS.TimeStampCLOCK** needs to be used to notify the debugger about the frequency of the global timestamp counter.
Pro: Better accuracy than mode "External". Allows timing correlation among cores sharing the same global timestamp.
Con: Requires some additional bandwidth on the trace port.Timestamp packets are not generated too often. Requires trace version 4 or higher.

NexusTimeStamps	Time measurement based on elapsed core clock cycles, which get emitted with every Nexus message. Command **NEXUS.CLOCK** needs to be used to notify the debugger about the frequency of the core clock. **Pro**: Provides accurate core clock cycle information. **Con**: Requires a lot more bandwidth on the trace port. Not suitable when core clock frequency changes dynamically. No timing correlation between cores.
NexusTimeStamps+External	Time measurement based on elapsed core clock cycles and external timestamps. Recommended if the core clock changes occasionally or if you can't determine the core clock frequency. **Pro**: Better accuracy than mode "External". Allows timing correlation among cores simultaneously traced with offchip-trace. **Con**: Requires a lot more bandwidth on the trace port.
NexusTimeStamp+ExternalTrack	Time measurement based on elapsed core clock cycles and external timestamps. Recommended if the core clock changes frequently. **Pro**: Better accuracy than mode "External". Allows timing correlation among cores simultaneously traced with offchip-trace. **Con**: Requires a lot more bandwidth on the trace port. Less accurate if the core clock frequency does not change.
NexusTimeStamps+TimeStamps	Time measurement based on elapsed core clock cycles and global onchip timestamps. **Pro**: Better accuracy than mode "TimeStamps". Allows timing correlation among cores sharing the same global timestamp. **Con**: Requires a lot more bandwidth on the trace port.

NEXUS.TimeStampCLOCK Specify frequency of the global timestamp

Format: **NEXUS.TimeStampCLOCK** *<frequency>*

If the trace infrastructure contains a global timestamp, TRACE32 needs to know its frequency, if the global timestamp is used to determine the elapsed time during the trace recoeding. This command only has an effect if **NEXUS.TImeMode** was set to “Timestamps” or “NexusTimeStamps+TimeStamps”.

This feature is only available for DesignWare ARC Trace version 4 and higher.

Format: **NEXUS.TraceID AUTO** | *<number>*

Default: AUTO.

The command **NEXUS.TraceID** sets the ATB-ID used by the DesignWare ARC Trace when emitting the trace stream to the CoreSight ATB.

Every trace stream must have a different ID inside the same CoreSight ATB network. It is especially important with AMP multicore configurations to ensure that every trace producer uses a different ID.

In SMP multicore configurations this command sets the ID of the first core, while the ID is incremented for each consecutive core in the same SMP cluster.

This feature is only available when DesignWare ARC Trace was build with a CoreSight compatible (ATB) offload interface.

NEXUS.WTM Enable watchpoint trace messages

Format: **NEXUS.WTM** [**ON** | **OFF**]

Enables the generation of Watchpoint Trace Messages, which are emitted when an actionpoint (onchip breakpont) triggers. Watchpoint Trace Messages are ignored by TRACE32.

ON Debug status messages enabled.

OFF Debug status messages disabled (default & recommended).

Normal 20-Pin Connector

The 20-pin connector is the typical connector used with ARC CPUs but also with quite a lot of other CPU families (like e.g. ARM Cortex). The Lauterbach debug cable LA-3750 is designed for this pin-out. The following drawing shows the top view to the male connector on the target board.

Signal	Pin	Pin	Signal
VTREF	1	2	VSUPPLY (not used)
TRST- (optional)	3	4	GND
TDI	5	6	GND
TMS|TMSC	7	8	GND
TCK|TCKC	9	10	GND
RTCK (optional)	11	12	GND
TDO	13	14	GND
SRST- (optional)	15	16	GND
EVTI (optional)	17	18	GND
EVTO (leave open)	19	20	GND

The meaning of the signals is as follows:

TCK | TCKC	to CPU	JTAG Clock It is recommended to put a pull-DOWN to GND on this signal.
TMS	to CPU	Standard JTAG TMS It is recommended to put a pull-UP to VTREF on this signal for standard 4-pin JTAG.
TMSC	to/from CPU	Compact JTAG TMSC Your chip should have a bus-hold on this line for compact JTAG.
TDI	to CPU	JTAG TDI It is recommended to put a pull-UP to VTREF on this signal. Only required for standard 4-pin JTAG / Optional for compact JTAG.
TDO	from CPU	JTAG TDO (No pull-up or pull-down is required.) Only required for standard 4-pin JTAG / Optional for compact JTAG.
TRST-	to CPU	JTAG Testport Reset (Optional signal) No pull-up or pull-down is required.
RTCK	from CPU	JTAG Return Clock (Optional signal) No pull-up or pull-down is required.
VTREF	from CPU	Reference voltage This voltage should indicate the nominal HIGH level for the JTAG and trace pins. So for example, if your signals have a voltage swing from 0 … 3.3 V, the VTREF pin should be connected to 3.3 V.

SRST-	to/from CPU	System Reset Signal. (Optional signal) If your board has a low active CPU reset signal, you can connect this low active reset signal to this pin. This enables the debugger to detect a CPU reset (see **SYStem.Option.ResetDetection.nSRST**). Furthermore the debugger can drive this pin to GND to reset the CPU (see **SYStem.Option.EnReset**). The debugger drives this pin as open-drain, so a pull-up is mandatory.
EVTI	to CPU	Nexus Event In. (optional), only used with DesignWare ARC Trace to force a Nexus Synchronization Message.
EVTO	from CPU	Nexus Event Out (optional). This pin is not used by ARC processors. (Leave open (N/C) if not used)

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

MIPI10 / MIPI20 / MIPI34 Connector

For the pin-out of the MIPI10, MIPI20, or MIPI34 connector see
https://www.lauterbach.com/adarmcombi.html.

You can connect to the MIPI10, MIPI20, or MIPI34 connector by either using a converter for the normal 20-pin debug cable, or by using a CombiProbe with a standard MIPI34 whisker (which is suitable for MIPI10, MIPI20 and MIPI34)

Converged MIPI60-Cv2 Connector

For the converged MIPI60 target pinout specified by Intel®see
"MIPI60-Cv2 Connector" (debugger_x86.pdf)
or **https://www.lauterbach.com/adcobintelx86.html**

To use this connector you usually need a Lauterbach CombiProbe with MIPI60-Cv2 whisker plus a separate ARC debug license. (LA-4590 + LA3750A)

XDP Connector

For the 60-pin XDP connector specified by Intel® see
"JTAG Connector" (debugger_x86.pdf)
or **https://www.lauterbach.com/adatom.html**

Do not use the 60-pin XDP connector for new designs.

Instead use the **"Converged MIPI60-Cv2 Connector"**, page 87.

The Lauterbach debug cable LA-3752 is designed for the 60-pin XDP pin-out. It has been out of production since 2017.

Trace Signals

Pin	Direction	Description
MKCO	from CPU	Nexus Message Clock-out (trace clock). This pin is mandatory and has to be on its dedicated position.
MSEOx	from CPU	Nexus Message Start/End Out. These signals are controlling the transitions between the state in the Nexus state machine. The signals are mandatory. With TRACE32 command **Analyzer.REMAP** you can move them to the location of any MSEO or MDO pin.
MDOx	from CPU	Depending on the build configuration of the ARC trace block, you need 4, 8, or 16 MDO lines. With TRACE32 command **Analyzer.REMAP** you can move them to the location of any MSEO or MDO pin.
VREF-TRACE	from CPU	This voltage should indicate the nominal HIGH level for the trace pins. So for example, if your signals have a voltage swing from 0 V … 3.3 V, the VREF-TRACE pin should be connected to 3.3 V. This pin is mandatory and have to be on its dedicated position.

Normal Nexus Auxiliary Port (Mictor 38)

Connector "TRACE A" only.

Signal	Pin	Pin	Signal
N/C	1	2	N/C
N/C	3	4	N/C
N/C	5	6	MCKO
EVTI (opt.)	7	8	EVTO (opt.)
SRST- (opt.)	9	10	N/C
TDO	11	12	VREF-TRACE
RTCK (opt.)	13	14	VREF-JTAG
TCK | TCKC	15	16	MDO7
TMS | TMSC	17	18	MDO6
TDI	19	20	MDO5
TRST- (opt.)	21	22	MDO4
MDO15	23	24	MDO3
MDO14	25	26	MDO2
MDO13	27	28	MDO1
MDO12	29	30	MDO0
MDO11	31	32	GND
MDO10	33	34	GND
MDO9	35	36	MSEO1
MDO8	37	38	MSEO0

Dual Eight-bit Nexus Auxiliary Port (Mictor 38)

Connector "TRACE A" only.

Signal	Pin	Pin	Signal
N/C	1	2	N/C
N/C	3	4	N/C
N/C	5	6	MCKO
EVTI (opt.)	7	8	EVTO (opt.)
SRST- (opt.)	9	10	N/C
TDO	11	12	VREF-TRACE
RTCK (opt.)	13	14	VREF-JTAG
TCK | TCKC	15	16	MDO7_A
TMS | TMSC	17	18	MDO6_A
TDI	19	20	MDO5_A
TRST- (opt.)	21	22	MDO4_A
MDO7_B	23	24	MDO3_A
MDO6_B	25	26	MDO2_A
MDO5_B	27	28	MDO1_A
MDO4_B	29	30	MDO0_A
MDO3_B	31	32	MSEO1_B
MDO2_B	33	34	MSEO0_B
MDO1_B	35	36	MSEO1_A
MDO0_B	37	38	MSEO0_A

Out Offload and CoreSight TPIU

For details please refer to **"ARM-ETM Trace"** (trace_arm_etm.pdf).