LAUTERBACH
DEVELOPMENT TOOLS

Simulator for ARC

MANUAL

Simulator for ARC

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .... 🗁

TRACE32 Instruction Set Simulators .... 🗁

Simulator for ARC **1**

History **4**

Introduction **4**

Supported ARC Cores 4

Brief Overview of Documents for New Users 5

Demo and Start-up Scripts 6

TRACE32 License **6**

Troubleshooting **7**

FAQ **7**

Quick Start **8**

Configure the Debugger to Use the ARCINT Interface **12**

Use T32Start 12

Modify an Existing Configuration File 13

Create a New Configuration File 13

ARCINT specific SYStem Commands **14**

SYStem.LIBrary	Set path to debug driver of simulator	14
SYStem.PROPertieS	Control properties of the used simulator (usually nSIM)	15
SYStem.PROPertieS.ADD	Add a property to configure the simulator	15
SYStem.PROPertieS.Delete	Remove a property to configure the simulator	15
SYStem.PROPertieS.List	Show all property sets to configure the simulator	16
SYStem.PROPertieS.Modify	Change property set to configure simulator	16

Access Classes **17**

CPU specific SETUP Command **18**

SETUP.DIS	Disassembler configuration	18

CPU specific SYStem Commands **19**

SYStem.CPU	Select CPU type	19
SYStem.MemAccess	Select run-time memory access method	20
SYStem.Mode	Select target reset mode	21
SYStem.Option	Set a target-specific option	22

SYStem.Option.detectOTrace Disable auto-detection of on-chip trace 22
SYStem.Option.Endianness Set the target endianness 22
SYStem.Option.HotBreakPoints Set breakp. when CPU is running 23
SYStem.Option.IMASKASM Disable interrupts while single stepping 23
SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping 24
SYStem.Option.LimmBreakPoints Software breakpoints with extra NOPs 24
SYStem.Option.MMUSPACES Separate address spaces by space IDs 25
SYStem.Option.OVERLAY Enable overlay support 26
SYStem.Option.RegNames Enable trivial names for core registers 26
SYStem.Option.TIMEOUT Define maximum time for core response 27
SYStem.state Show SYStem settings window 27

On-chip Breakpoints/Actionpoints **28**
Using On-chip Breakpoints 28
Breakpoints in a ROM Area 28
Limitations 29
TrOnchip.CONVert Allow extension of address range of breakpoint 30
TrOnchip.VarCONVert Convert breakpoints on scalar variables 32
TrOnchip.OnchipBP Number of on-chip breakpoints used by debugger 33
TrOnchip.RESet Set on-chip trigger to default state 34
TrOnchip.state Display on-chip trigger window 34
TrOnchip.MCD MCD API specific on-chip triggers 35
TrOnchip.MCD.McdBreakPoints Set onchip breakpoint via MCD API 35
TrOnchip.MCD.CoreHalted Workaround to detect core-halt via MCD API 35
TrOnchip.MCD.CoreRunning Workaround to start/step core via MCD API 36

CPU specific MMU Commands **37**
MMU.DUMP Page wise display of MMU translation table 37
MMU.List Compact display of MMU translation table 39
MMU.SCAN Load MMU table from CPU 40
MMU.Init Invalidate TLB entries 41
MMU.Set Set an MMU TLB entry 41

Simulator for ARC

Version 06-Jun-2024

History

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.

Introduction

This manual describes how to debug a simulator connected to the Lauterbach debugger TRACE32 via the ARCINT interface such as the **nSIM simulator** from Synopsys.

At the time of release of this document, a TRACE32 internal instruction set simulator was not yet available. As soon as a TRACE32 internal instruction set simulator is available, it will be also described in this manual.

For using TRACE32 with a virtual prototype such as the Synopsys Virtualizer, please see the **"Virtual Targets User's Guide"** (virtual_targets.pdf).

- This manual is for debugging **nSIM**
- This manual is for debugging any target via the **ARCINT interface**
- This is *not* for the TRACE32 internal instruction set simulator
- This is *not* for debugging virtual prototypes
- This is *not* for debugging via the MCD interface

Supported ARC Cores

The following ARC cores from Synopsys are supported:

- ARC-HS family : HS34, HS36, HS38, HS47D
- ARC-EM family : EM4, EM6, EM5D, EM7D, EM9D, EM11D
- ARC 700 core family : ARC710D, ARC725D, ARC750D, ARC770D
- ARC 600 core family : ARC601, ARC605, ARC610D, ARC652D, ARC630D, AS211SFX, AS221BD

- ARCtangent-A5 cores (see note below)
- ARCtangent-A4 cores (see note below)

If you require support of any other ARC core, please contact support@lauterbach.com

!	Legacy ARCtangent-A4 and A5 cores are not supported by the nSIM simulator from Synopsys.

Brief Overview of Documents for New Users

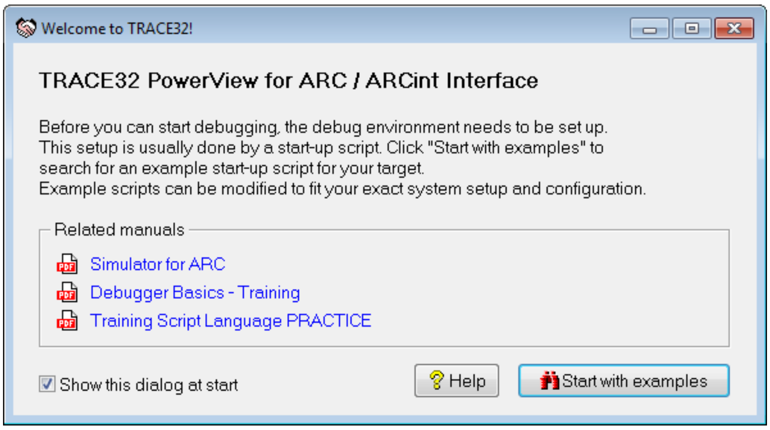
Architecture-independent information:

- **"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**):



TRACE32 PowerView for ARC / ARCint Interface

Before you can start debugging, the debug environment needs to be set up.
This setup is usually done by a start-up script. Click "Start with examples" to search for an example start-up script for your target.
Example scripts can be modified to fit your exact system setup and configuration.

Related manuals
Simulator for ARC
Debugger Basics - Training
Training Script Language PRACTICE
☑ Show this dialog at start
Help
Start with examples

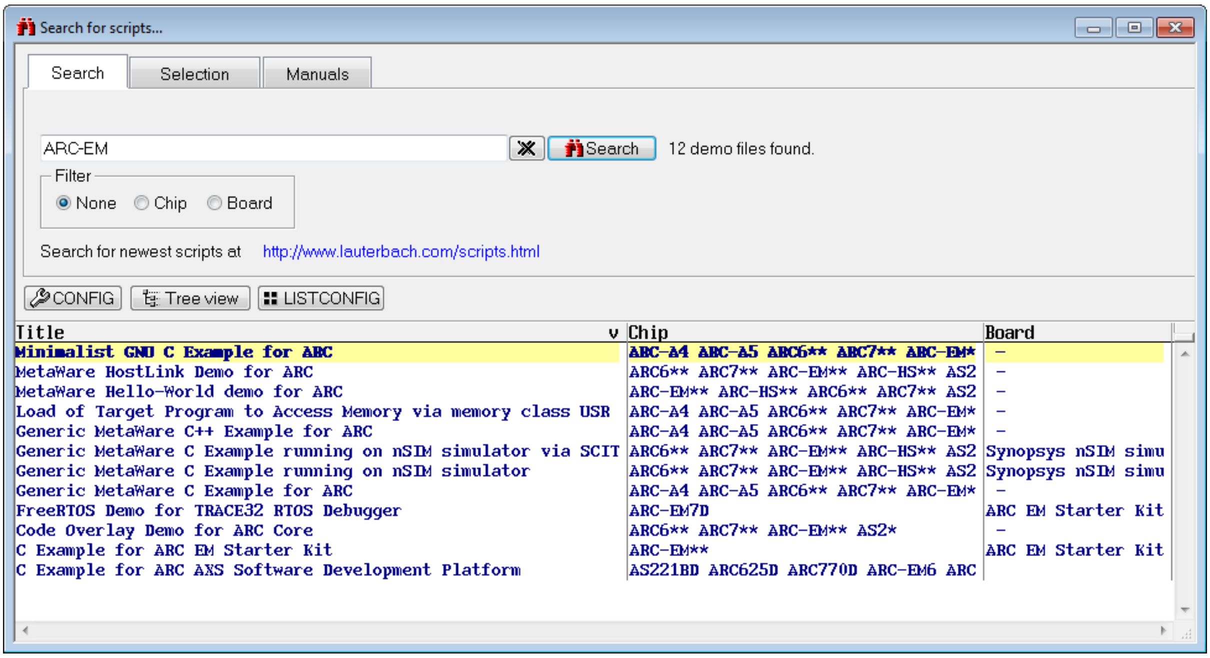
Demo and Start-up Scripts

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.



ARC-EM

12 demo files found.

Filter: [✓ None] Chip Board

Search for newest scripts at http://www.lauterbach.com/scripts.html

Title	Chip	Board
Minimalist GNU C Example for ARC	ARC-A4 ARC-A5 ARC6** ARC7** ARC-EM*	-
MetaWare HostLink Demo for ARC	ARC6** ARC7** ARC-EM** ARC-HS** AS2	-
MetaWare Hello-World demo for ARC	ARC-EM** ARC-HS** ARC6** ARC7** AS2	-
Load of Target Program to Access Memory via memory class USR	ARC-A4 ARC-A5 ARC6** ARC7** ARC-EM*	-
Generic MetaWare C++ Example for ARC	ARC-A4 ARC-A5 ARC6** ARC7** ARC-EM*	-
Generic MetaWare C Example running on nSIM simulator via SCIT	ARC6** ARC7** ARC-EM** ARC-HS** AS2	Synopsys nSIM simu
Generic MetaWare C Example running on nSIM simulator	ARC6** ARC7** ARC-EM** ARC-HS** AS2	Synopsys nSIM simu
Generic MetaWare C Example for ARC	ARC-A4 ARC-A5 ARC6** ARC7** ARC-EM*	-
FreeRTOS Demo for TRACE32 RTOS Debugger	ARC-EM7D	ARC EM Starter Kit
Code Overlay Demo for ARC Core	ARC6** ARC7** ARC-EM** AS2*	-
C Example for ARC EM Starter Kit	ARC-EM**	ARC EM Starter Kit
C Example for ARC AXS Software Development Platform	AS221BD ARC625D ARC770D ARC-EM6 ARC	

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

TRACE32 License

To use TRACE32 for debugging via the ARCINT interface (e.g. to debug nSIM) you require a **TRACE32 front-end license for ARC**.

Please order the **LA-8903L** floating license.

Contact your local Lauterbach representative for questions regarding licenses and orders.

Independently of the licenses for the TRACE32 debugger you will need a license for the external simulator you like to debug via the ARCINT interface. E.g.: Get in touch with Synopsys for a license of the nSIM simulator.

Troubleshooting

No information available until yet.

FAQ

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

Quick Start

This chapter should give you a quick overview to start the debugging of an ARC core simulator by an SIM simulator.

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

1. Configure the Debugger to Use the ARCINT Interface

Before starting the TRACE32 software (t32marc.exe) you have to configure the type of the interface used by the debugger.

To use the ARCINT interface you have to set `PBI=ARCINT` in the TRACE32 configuration file (config.t32).

For more details on how to configure TRACE32 for ARCINT, see **“Configure the Debugger to Use the ARCINT Interface”**, page 12.

Start TRACE32 for ARC (t32marc.exe).

2. Set the Path to the ARCINT Interface Library

TRACE32 communicates with the ARCINT compatible debug backend over a shared library file (*.dll / *.so).

Use command **SYStem.LIBrary *** to set up the path to the shared library provided with your debug backend.

E.g. in case of a direct connection to nSIM use the command

```
SYStem.LIBrary "C:\ARC\ARC_2016-06\nSIM\nSIM_64\lib\libsim.dll"
```

⚠	Please ensure to use the 64-bit version of TRACE32 (C:\T32\bin\windows64\t32marc.exe) if you use a shared library compiled for a 64-bit host operating system, while you have to use the 32-bit version of TRACE32 (C:\T32\bin\windows\t32marc.exe) if you use a shared library compiled for 32-bit host operating system.

By default SYStem.LIBrary points to `%NSIM_HOME%\lib\libsim.dll` in case you are using the 32-bit version of TRACE32 while it points to `%NSIM_HOME%\..\lib_64\libsim.dll`. in case you are using the 64-bit version of TRACE32.
Thus, in case you are using nSIM you usually don’t have to use the command SYStem.LIBrary explicitly.

3. Configure your Simulator

Most simulators have to be configure to know what hardware they actually should simulate.
E.g. you have to tell nSIM which ARC core with which configuration it should provide.

Configure your simulator’s properties via the command:

```
SYStem.PROPertieS.ADD "<prop_name>=<prop_value>"
```

Look up you simulators manual for suitable property names and values.

E.g. in case of debugging an ARC-EM simulated by nSIM you can configure the simulator like this:

```
SYStem.PROPertieS.ADD "nsim_isa_family=av2em"
SYStem.PROPertieS.ADD "nsim_isa_core=1"
SYStem.PROPertieS.ADD "nsim_isa_code_density_option=2"
SYStem.PROPertieS.ADD "nsim_isa_shift_option=2"
SYStem.PROPertieS.ADD "nsim_isa_num_actionpoints=8"
SYStem.PROPertieS.ADD "nsim_isa_aps_feature=1"
```

In case you have a TCF of your ARC core configuration (and using nSIM) you can do the whole configuration with the following single command:

```
SYStem.PROPertieS.ADD "nsim_tcf=<path_to_your_tcf_file>"
```

4. Select the CPU Type

```
SYStem.CPU AUTO
```

AUTO works usually fine with nSIM.

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 the command **Go main**

9. 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

RESet                                   ; Reset all setting set before

PRIVATE &nSIM
&nSimHome=OS.ENV(NSIM_HOME)             ; get base directory of nSIM
IF SOFTWARE.64BIT()
  &nSimHome="&(nSIM)_64"

SYStem.LIBrary "&nSIM/libsim.dll"       ; Set path to nSIM DLL

SYStem.PROPertieS.ADD "nsim_tcf" \      ; Configure nSIM via TCF file
  "&nSIM/etc/tcf/templates/arc610d.tcf"

SYStem.CPU AUTO                         ; Enable CPU auto-detection

SYStem.Up                               ; Connect to simulated CPU

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

Data.List                               ; View sourcecode/disassembly *)

Register.view                           ; View core register          *)

Var.Frame /Args /Locals                 ; View call stack             *)

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

Break.Set 0x400                         ; Set breakpoint on fixed addr.

Break.Set main                          ; Set breakpoint on main() func.
```

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

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.

Configure the Debugger to Use the ARCINT Interface

Before starting the TRACE32 software (t32marc.exe) you have to configure the type of the interface used by the debugger.

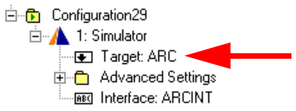
Choose one of the following options:

- Use T32Start (Windows only)
- Modify an existing configuration file
- Create a new configuration file

Use T32Start

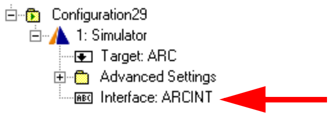
By the release of this document T32Start was only available for Microsoft Windows. Linux, and Mac users have to create or modify a configuration file manually.

1. Start T32Start (t32start.exe)
2. Click on **Configuration Tree** (top of the shown tree) and click on **Add > Configuration**
3. Click on the new created configuration and click on **Add > Simulator**
4. Open the Simulator entry inside your new configuration (by double clicking on it), select **Target** and change the target from **ARM/XScale/Janus** to **ARC**.



```
Configuration29
  1: Simulator
    Target: ARC
    Advanced Settings
    Interface: ARCINT
```

5. Open entry **Simulator** entry inside your new configuration (by double clicking on it), select “Interface” and change the interface from **SIM** to **ARCINT**.



```
Configuration29
  1: Simulator
    Target: ARC
    Advanced Settings
    Interface: ARCINT
```

6. Select again **Simulator** inside your new configuration and click **Start**. TRACE32 for ARC should open.

For more on the configuration tool T32Start, see **“T32Start”** (app_t32start.pdf).

Modify an Existing Configuration File

1. First, open the configuration file you are already using to start TRACE32 for ARC in a text editor. If you haven't specified a special file on the command line than it's most likely the file 'config.t32' in you TRACE32 system directory. E.g. `C:\T32\config.t32`
   You can also use the TRACE32 command **PRINT OS.PresentConfigurationFile()** to display the configuration file used by an active TRACE32 PowerView GUI.

2. Go to the line starting with **`PBI=`** and change it to **`PBI=ARCINT`**.

Example of existing config.t32 for using PowerDebug via USB	**Example of new config.t32 for debugging nSIM via ARCINT**
`OS=`	`OS=`
`SYS=C:\T32`	`SYS=C:\T32`
`HELP=C:\T32\pdf`	`HELP=C:\T32\pdf`
`PBI=`	**`PBI=ARCINT`**
`USB`	
`PRINTER=WINDOWS`	`PRINTER=WINDOWS`

3. Start (or restart) TRACE32

Create a New Configuration File

1. Open a text editor and create a file with the following content:

Windows	**Linux**
`OS=`	`OS=`
`ID=T32_ARCINT_01`	`ID=T32_ARCINT_01`
`SYS=C:\T32`	`SYS=/opt/t32`
`HELP=C:\T32\pdf`	`HELP=/opt/t32/pdf`
`PBI=ARCINT`	`PBI=ARCINT`
`PRINTER=WINDOWS`	`PRINTER=PS`

2. Save the file in your TRACE32 system directory with the name configARCINT.t32

3. Launch TRACE32 for ARC (t32marc.exe) with the command line argument **-c** <*path of configARCINT.t32*>
   E.g.: `C:\T32\bin\windows64\t32marc.exe -c C:\T32\configARCINT.t32`

ARCINT specific SYStem Commands

SYStem.LIBrary — Set path to debug driver of simulator

Format:	**SYStem.LIBrary** *<file>*

TRACE32 communicates with the ARCINT compatible debug backend over a shared library file (*.dll / *.so). The command **SYStem.LIBrary** sets the path to the shared library provided with your debug backend. Use this command only before executing **SYStem.Mode Up**

E.g. in case of a direct connection to nSIM running on a 64-bit version of Windows use the command

```
SYStem.LIBrary "C:\ARC\ARC_2016-06\nSIM\nSIM_64\lib\libsim.dll"
```

E.g. in case use are using a DLL to access a SCIT backend (by Snyopsys) use command

```
SYStem.LIBrary "C:\ARC\ARC_2016-06\MetaWare\arc\bin\arccli.dll"
```

Please ensure to
use the 64-bit version of TRACE32 (C:\T32\bin\windows64\t32marc.exe)
if you use a shared library compiled for a 64-bit host operating system,
while you have to
use the 32-bit version of TRACE32 (C:\T32\bin\windows\t32marc.exe)
if you use a shared library compiled for 32-bit host operating system.

By default SYStem.LIBrary points to `%NSIM_HOME%\lib\libsim.dll` in case you are using the 32-bit version of TRACE32,m while it points to `%NSIM_HOME%\..\lib_64\libsim.dll` in case you are using the 64-bit version of TRACE32.

Thus, in case you are using nSIM you usually don't have to use the command **SYStem.LIBrary** explicitly.

SYStem.PROPertieS Control properties of the used simulator (usually nSIM)

SYStem.PROPertieS.ADD Add a property to configure the simulator

Format 1: **SYStem.PROPertieS.ADD** "*<property_name>*" "*<value>*"

Format 2: **SYStem.PROPertieS.ADD** "*<property_name>*=*<value>*"

Most simulators have to be configure to know what hardware they actually should simulate.
E.g. you have to tell nSIM which ARC core with which configuration it should provide.

The command **COMmand.PROPertieS.ADD** tells TRACE32 to configure the simulator with the given property when connecting to it. Use this command only before executing **SYStem.Mode Up**

Look up you simulators manual for suitable property names and values.

E.g. in case of debugging an ARC-EM simulated by nSIM you can configure the simulator like this:

```
SYStem.PROPertieS.ADD "nsim_isa_family=av2em"
SYStem.PROPertieS.ADD "nsim_isa_core=1"
SYStem.PROPertieS.ADD "nsim_isa_code_density_option=2"
SYStem.PROPertieS.ADD "nsim_isa_shift_option=2"
SYStem.PROPertieS.ADD "nsim_isa_num_actionpoints=8"
SYStem.PROPertieS.ADD "nsim_isa_aps_feature=1"
```

In case you have a TCF of your ARC core configuration (and using nSIM) you can do the whole configuration with the following single command:

```
SYStem.PROPertieS.ADD "nsim_tcf=<path_to_your_tcf_file>"
```

SYStem.PROPertieS.Delete Remove a property to configure the simulator

Format: **SYStem.PROPertieS.Delete** "*<property_name>*"

Removes a previously added property which would otherwise configure the simulator when connecting to it.
Use this command only before executing **SYStem.Mode Up**

SYStem.PROPertieS.List Show all property sets to configure the simulator

Format:	**SYStem.PROPertieS.List**

Opens a window which shows all the properties which have been added in TRACE32 to configure the external simulator (which gets debugged via ARCINT).

SYStem.PROPertieS.Modify Change property set to configure simulator

Format 1:	**SYStem.PROPertieS.Modify** **"***<property_name>***"** **"***<value>***"**
Format 2:	**SYStem.PROPertieS.Modify** **"***<property_name>***=***<value>***"**

Changes the value of a previously added property which gets used to configure the simulator when connecting to it. Use this command only before executing **SYStem.Mode Up**

Look up your simulators manual for suitable property names and values.

Access Classes

Access Class	Description
P:	Program Memory.
D:	Data Memory.
C:	Program or Data Memory (unspecified)
AUX:	Auxiliary Register Space. Accesses to this memory class allows you to read and write Auxiliary Registers and read Build Registers.
A: AP:, AD:	Absolute addressing (physical address) on SoCs with Memory Management Unit (MMU)
E	Access memory while the CPU is running. See **SYStem.CpuAccess** and **SYStem.MemAccess**. Any memory class can be prefixed with **E**, if the memory class supports access while the CPU is running.
USR:	Access to special memory via user-defined target program. See **Data.USRACCESS**.
OVS:	On systems with code overlays, OVS allows to access in-active overlay segments located at the memory where it gets loaded from (storage) via the addresses where it gets executed. See **sYmbol.OVERLAY**.
JSEQ:	Access data via JTAG sequences registered with **JTAG.SEQuence.MemAccess.ADD**
VM:	Virtual Memory (memory on the debug system).

Currently the cache is bypassed with any access. Thus, the debugger updates the memory from the cache before any memory gets accessed. After each memory access the debugger invalidates the cache.

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.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.

SYStem.MemAccess — Select run-time memory access method

Format:	**SYStem.MemAccess Enable | Denied | StopAndGo** **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
```

SYStem.Mode

Format:	**SYStem.Mode** <*mode*> **SYStem.Down** **(alias for SYStem.Mode Down)** **SYStem.Up** **(alias for SYStem.Mode Up)**
<*mode*>:	**Down** **Up**

Down	The debugger disconnects from the backend. 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.
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.

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

SYStem.Option — Set a target-specific option

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.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.HotBreakPoints — Set breakp. when CPU is running

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.IMASKASM — Disable interrupts while single stepping

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

Default: OFF.

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

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

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

Default: OFF.

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

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.

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>
```

SYStem.Option.RegNames — Enable trivial names for core registers

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.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.

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.

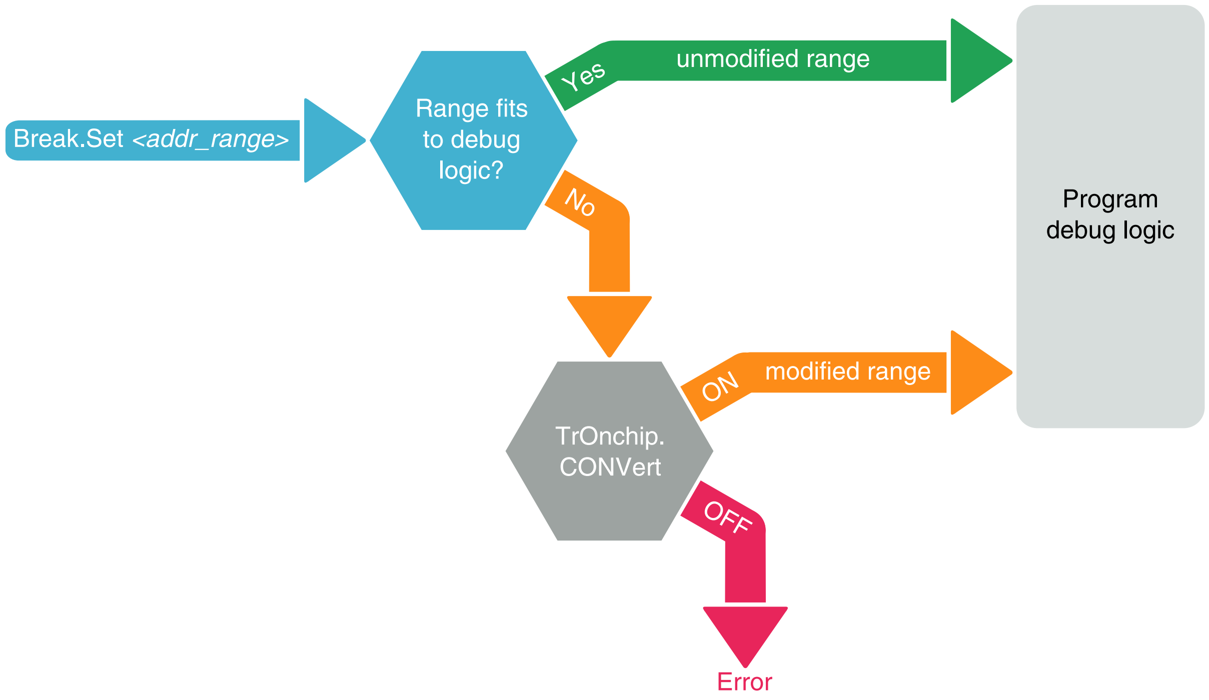
Limitations

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).

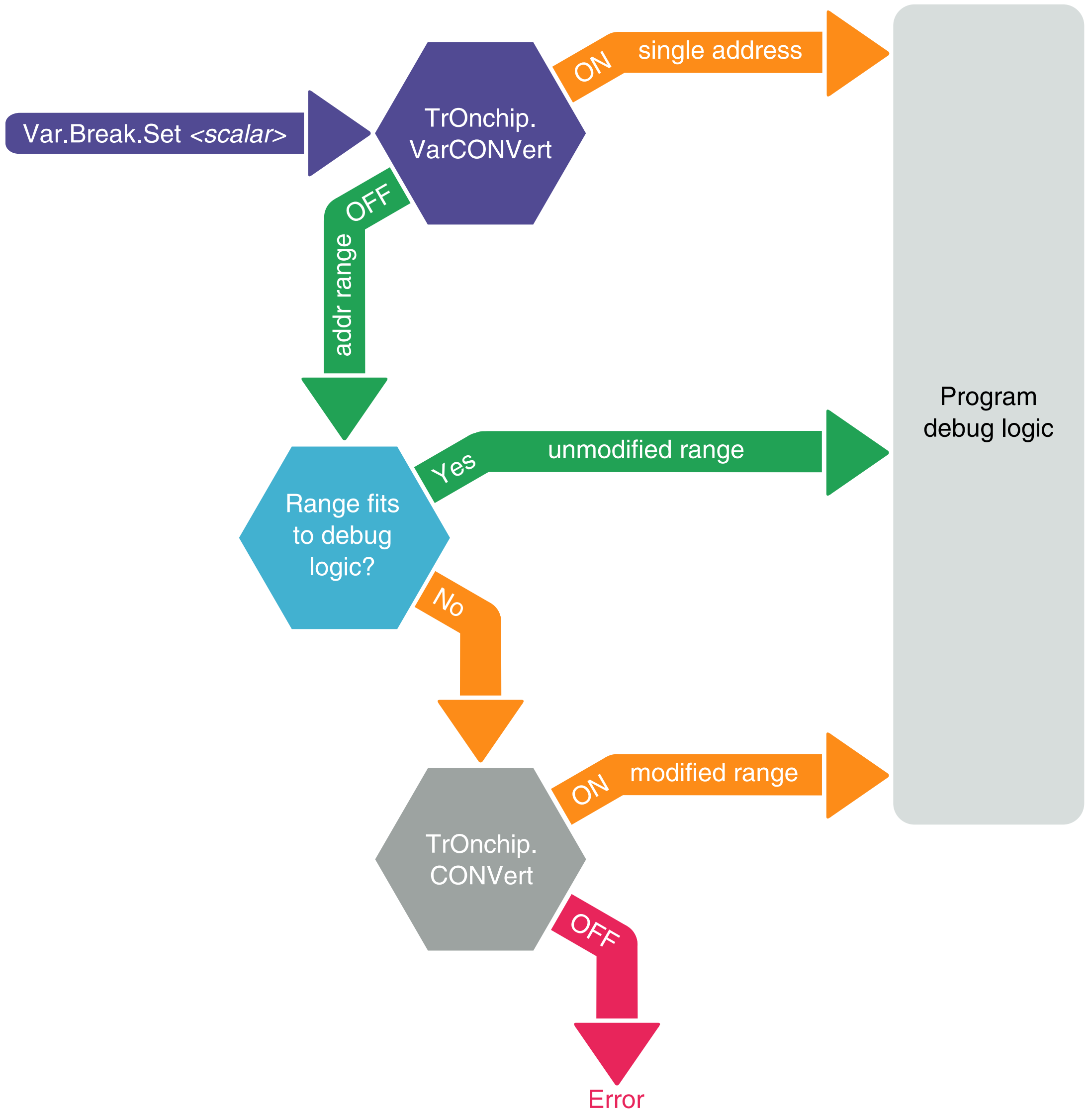
The **TrOnchip.CONVert** command covers case 3. For case 3) the user may decide whether the debugger is allowed to change the user-defined address range of a breakpoint or not by setting **TrOnchip.CONVert** to **ON** or **OFF**.

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.

TrOnchip.MCD.McdBreakPoints Set onchip breakpoint via MCD API

Format:	**TrOnchip.MCD.McdBreakPoints** [**ON** | **OFF**]

Default: ON

This command is only available when debugging via the MCD interface.

When debugging via the MCD interface Onchip breakpoints are set by default via the dedicated API function. By setting TrOnchip.MCD.McdBreakPoints to OFF, onchip breakpoints are set instead via the ARC auxiliary registers to control Actionpoints.

You might want to try TrOnchip.MCD.McdBreakPoints OFF, if onchip breakpoints do not work as expected. Otherwise leave the setting as ON.

If you set TrOnchip.MCD.McdBreakPoints to OFF, you might have to set both **TrOnchip.MCD.CoreHalted** and **TrOnchip.MCD.CoreRunning** to ON, to ensure that the debugger detects a stop on an onchip breakpoint and is able to restart the core from there.

TrOnchip.MCD.CoreHalted Workaround to detect core-halt via MCD API

Format:	**TrOnchip.MCD.CoreHalted** [**ON** | **OFF**]

Default: OFF

This command is only available when debugging via the MCD interface.

By setting TrOnchip.MCD.CoreHalted to ON, TRACE32 will enable trigger resource "CoreHalted". This is required on some MCD implementations, to detect at any time, that the core is halted.

You might want to try TrOnchip.MCD.CoreHalted ON, if TRACE32 does not show your core as stopped, although you know that it has stopped. Otherwise leave the setting as OFF.

Format:	**TrOnchip.MCD.CoreRunning** [**ON** | **OFF**]

Default: OFF

This command is only available when debugging via the MCD interface.

By setting TrOnchip.MCD.CoreRunning to ON, TRACE32 will explicitly clear the HALT bit of the ARC STATUS32 register when the core should be started (e.g. via command **Go.direct**) or stepped (e.g. via command **Step.Asm**).

You might want to try TrOnchip.MCD.CoreRunning ON, if starting or stepping your ARC core does not work as expected. Otherwise leave the setting as OFF.

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.