

TriCore Debugger and Trace





Release 02.2025

MANUAL

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents	
ICD In-Circuit Debugger	
Processor Architecture Manuals	
TriCore	
TriCore Debugger and Trace	1
History	8
Safety Precautions	9
Introduction	10
Brief Overview of Documents for New Users	10
Available Tools	11
Debugger	11
Software-only Debugger for XCP	11
On-chip Trace	11
Serial Off-chip Trace (AGBT)	12
Parallel Off-chip Trace	12
Co-Processor Debugging (PCP/GTM)	12
Co-Processor Debugging (HSM)	12
Multicore Debugging and Tracing	13
Software Installation	13
Configuration	14
System Overview	14
Related Documents	14
Demo and Start-up Scripts	15
OCDS Levels	15
Debugging	16
Single-Core Debugging (AUDO)	17
Single-Core Debugging - Quick Start	17
Multicore Debugging (AURIX)	19
SMP Debugging - Quick Start	19
AMP Debugging - Quick Start	21
AMP vs. SMP vs. iAMP	23
Selecting the right AURIX CPU	24
Understanding Multicore Startup by Application Code	24

About Ambiguous Symbols	25
Access Classes	26
Breakpoints	27
Software Breakpoints	27
On-chip Breakpoints	27
MAP.BOnchip Command	28
Advanced Breakpoints	28
Single Stepping	29
Assembler Level	29
HLL Level	29
Flash	30
Onchip Triggers (TrOnchip Window)	32
BenchMarkCounter	33
Example: Measuring Instructions and Stalls per Clock Cycle	34
Example: A-to-B Mode (single shot)	35
Example: A-to-B Mode (average)	36
Example: Record Counters Periodically	37
Watchpins	38
AUDO Devices	38
AURIX Devices	38
Accessing Cached Memory Areas and Cache Inspection	42
AUDO Devices	42
AURIX Devices	43
Parallel Usage of a 3rd-Party Tool	47
Physical Sharing of the Debug Port	47
Debugging an Application with the Memory Protection Unit Enabled	49
TriCore v1.6 and Later	49
TriCore v1.3.1 and Earlier	49
Debugging with MPU Enabled in RAM	49
Debugging with MPU Enabled in FLASH	50
Debugging through Resets and Power Cycles	51
Soft Resets	51
Hard Resets	52
Power Cycles	53
Suspending the System Timers of TC3xx	55
Suspending the Watchdogs	56
Code Overlays	56
Data Overlays	56
Cerberus Access Protection	56
Target Code Execution	57
Internal Break Bus (JTAG)	57
Troubleshooting	58
SYSem.Up Errors	58

Debugging Optimized Code	58
FAQ	59
Tracing	60
On-chip Trace (OCDS-L3)	60
Quick Start for Tracing with On-chip Trace (OCDS-L3)	60
Supported Features	61
Trace Control	62
Trace Evaluation	62
Impact of the Debugger on FPI Bus Tracing	62
Trace Control Using Break.Set or Var.Break.Set	63
CPU specific BMC Commands	70
BMC.SELect	Select counter for statistic analysis 70
BMC.<counter>.ATOB	Control A-to-B mode 70
BMC.<counter>.TRIGMODE	BMC trigger mode 70
BMC.<counter>.TRIGVAL	BMC trigger value 71
CPU specific SYStem.CONFIG Commands	72
SYStem.CONFIG.state	Display target configuration 72
SYStem.CONFIG	Configure debugger according to target topology 73
Daisy-Chain Example	75
TapStates	76
SYStem.CONFIG.CORE	Assign core to TRACE32 instance 77
SYStem.CONFIG.BreakPIN	Define mapping of break pins 78
SYStem.CONFIG.CAN	Configure CAN interface 79
SYStem.CONFIG.CAN.BaseCLOCK	Base clock for CAN interface 79
SYStem.CONFIG.CAN.NominalBRP	Set CAN nominal baud rate prescaler 80
SYStem.CONFIG.CAN.NominalTSEG1	Set CAN nominal Phase_seg1 80
SYStem.CONFIG.CAN.NominalTSEG2	Set CAN nominal Phase_seg2 80
SYStem.CONFIG.CAN.NominalSJW	Set CAN nominal SJW parameter 80
SYStem.CONFIG.CAN.DataBRP	Set CAN data baud rate prescaler 81
SYStem.CONFIG.CAN.DataTSEG1	Set CAN data Phase_seg1 81
SYStem.CONFIG.CAN.DataTSEG2	Set CAN data Phase_seg2 81
SYStem.CONFIG.CAN.DataSJW	Set CAN data SJW 82
SYStem.CONFIG.DAP	Configure DAP interface 83
SYStem.CONFIG.DAP.BreakPIN	Define mapping of break pins 83
SYStem.CONFIG.DAP.CRC6	Enable CRC6 mode 83
SYStem.CONFIG.DAP.DAPENable	Enable DAP mode on PORST 83
SYStem.CONFIG.DAP.SISP	Configure SISP setting 84
SYStem.CONFIG.DAP.USERn	Configure and set USER pins 84
SYStem.CONFIG.DEBUGPORT	Select target interface 86
SYStem.CONFIG.DEBUGPORTTYPE	Set debug cable interface mode 86
SYStem.CONFIG.DXCM	Configure DXCM 88
SYStem.CONFIG.DXCM.TXID	Control frame message ID 88

SYStem.CONFIG.DXCM.TXIDE	Control frame format	88
SYStem.CONFIG.DXCM.TXFDF	Control frame format	89
SYStem.CONFIG.DXCM.TXBRS	Control the use of baud rate switching	89
SYStem.CONFIG.DXCM.RXID	Set ID for frames from target	89
SYStem.CONFIG.DXCM.RXIDE	Expect extended frames from target	89
SYStem.CONFIG.DXCPL	Configure DXCPL	90
SYStem.CONFIG.DXCPL.Timing	Configure SPD timing for DXCPL	90
SYStem.CONFIG.EXTWDTDIS	Control external watchdog	90
SYStem.CONFIG.PortSHaRing	Control sharing of debug port with other tool	91
SYStem.CPU	Select CPU	92
SYStem.JtagClock	Set the JTAG frequency	93
SYStem.LOCK	Tristate the JTAG port	94
SYStem.MemAccess	Select run-time memory access method	95
SYStem.Mode	Establish the communication with the CPU	96
CPU and Architecture specific SYStem.Option Commands		98
SYStem.Option.BREAKFIX	Enable workaround for asynchronous breaking	98
SYStem.Option.CBSACCEN<x>	Cerberus access protection	99
SYStem.Option.DCFREEZE	Do not modify cache structure	100
SYStem.Option.DCREAD	Control cache behavior of reads	100
SYStem.Option.DSYNC	Force data synchronization	101
SYStem.Option.DOWNMODE	Behavior of SYStem.Mode Down	102
SYStem.Option.DUALPORT	Implicitly use run-time memory access	102
SYStem.Option.DataTrace	Enable data tracing	103
SYStem.Option.EndInitProtectionOverride	Override ENDINIT protection	103
SYStem.Option.HeartBeat	Bug fix to avoid FPI bus conflict	104
SYStem.Option.HoldReset	Reset duration	104
SYStem.Option.HSMRESTART	Restart HSM on connect	105
SYStem.Option.ICFLUSH	Flush instruction cache at 'Go' or 'Step'	105
SYStem.Option.IMASKASM	Disable interrupts while single stepping	105
SYStem.Option.IMASKHLL	Disable interrupts while HLL single stepping	106
SYStem.Option.JTAGENSEQ	Use JTAG initialization sequence	106
SYStem.Option.KEYCODE	Set debug interface password	107
SYStem.Option.KEYCODEWarnNotAccepted	Set warning level	107
SYStem.Option.LBIST	LBIST gap handling	107
SYStem.Option.MACHINESPACES	Address extension for guest OSes	108
SYStem.Option.MAPCACHE	Map cache automatically	109
SYStem.Option.OCDSELOW	Set OCDS line to low	110
SYStem.Option.OVC	Enable OVERLAY memory access	110
SYStem.Option.OVERLAY	Enable overlay support	111
SYStem.Option.PERSTOP	Enable global peripheral suspend	112
SYStem.Option.PMILBFIX	Enable PMI line buffer invalidation workaround	113
SYStem.Option.PostResetDELAY	Delay after RESET is released	114
SYStem.Option.ReadOnly	Block all write accesses	114

SYStem.Option.RESetBehavior	Set behavior when a reset occurs	115
SYStem.Option.ResetDetection	Set how hard resets are detected	115
SYStem.Option.ResetMode	Select reset method	116
SYStem.Option.RESetTMS	State of TMS line at reset	116
SYStem.Option.RUNRESTOREDELAY	Delay of restore after reset	116
SYStem.Option.SLOWRESET	Long timeout for resets	117
SYStem.Option.SOFTLONG	Set 32 bit software breakpoints	117
SYStem.Option.SSWWAIT	Emulate SSWWAIT	117
SYStem.Option.STEPOONCHIP	Step with onchip breakpoints	118
SYStem.Option.STEPSOFT	Step with software breakpoints	118
SYStem.Option.TB1766FIX	Bug fix for some TC1766 TriBoards	119
SYStem.Option.UNLOCKTIME	Timeout for debug port unlock	119
SYStem.Option.WDTFIX	Disables the watchdog on SYStem.Up	119
SYStem.Option.WDTSUS	Link the watchdog timer to the suspend bus	120
SYStem.RESetOut	In-target reset	120
SYStem.state	Open SYStem.state window	120
CPU specific TrOnchip Commands		121
TrOnchip.BreakBusN.BreakIN	Configure break pin of 'BreakBus N'	121
TrOnchip.BreakBusN.BreakOUT	Configure break pin of 'BreakBus N'	121
TrOnchip.BreakIN.<target>	Connect break <target> to BreakBus	122
TrOnchip.BreakOUT.<source>	Connect break <source> to BreakBus	122
<source>		123
HaLTEN		123
TrOnchip.CONVert	Adjust range breakpoint in on-chip resource	124
TrOnchip.CountX	Event X counter value	124
TrOnchip.CountY	Event Y counter value	124
TrOnchip.EXTeRnal	Configure TriCore break on BreakBus event	125
TrOnchip.PERSTOPOUT	Route suspend signal to pin	125
TrOnchip.RESet	Reset settings for the on-chip trigger unit	125
TrOnchip.SoftWare	Configure 'TriCore' break on debug instruction	126
TrOnchip.SusSWitch	Enable or disable suspend switch	126
TrOnchip.SusSWitch.FORCE	Force generation of suspend signal	126
TrOnchip.SusSWitch.Mode	Set suspend switch mode	127
TrOnchip.SusTarget	Connect special targets to the suspend bus	127
TrOnchip.SYNCHRONOUS	Switches mode for data breakpoints	127
TrOnchip.TDelay	Trace trigger delay	128
TrOnchip.TExtMode	Mode for external trigger input	128
TrOnchip.TExtPol	Polarity of external trigger input	128
TrOnchip.TMode	Trace mode	128
TrOnchip.TR0	Specify trigger event 0	129
TrOnchip.TR1	Specify trigger event 1	130
TrOnchip.state	Show on-chip trigger window	130
TrOnchip.WatchPin	Route core trigger to pin	131

TrOnchip.X	Select trigger source X	131
TrOnchip.Y	Select trigger source Y	132
Technical Data		133
Trace Connector		133
Technical Data for Debugger		133
Mechanical Dimensions		133
Appendix		134
Parallel Off-chip Trace - OCDS-L2 Flow Trace (Analyzer)		134
Overview		134
Quick Start for Tracing with OCDS-L2 Trace (Analyzer)		134
Supported Features		135
Version History		136
Timestamp Accuracy		136
Concurrent Usage of OCDS-L2 Off-chip Trace and OCDS-L3 On-chip Trace		137
Simple Trace Control		137
Trace Break Signals (OCDS-L2)		137
Trace Examples		138
Troubleshooting for OCDS-L2 Trace		140
No Data in Trace.List Visible		140
Error Diagnosis		140
Searching for Errors		141
Error Messages		142

History

- 24-May-2024 Updated commands [SYStem.Option.IMASKASM](#) and [SYStem.Option.IMASKHLL](#).
- 28-Aug-2023 The chapter 'Simple Trace Control' has been renamed to '[Trace Control Using Break.Set or Var.Break.Set](#)' and updated.
- 07-Nov-2022 Added chapters "[Code Overlays](#)", "[Data Overlays](#)", "[iAMP Debugging](#)".
- 28-Oct-2022 Updated command [TrOnchip.SusTarget](#).
- 07-Sept-2022 New commands: [SYStem.Option.RUNRESTOREDELAY](#), and [SYStem.Option.UNLOCKTIME](#).
- 07-Sept-2022 Updated description of command: [SYStem.Option.EndInitProtectionOverride](#).

Caution:	<p>To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.</p> <p>Recommendation for the software start:</p> <ul style="list-style-type: none">• Disconnect the debug cable from the target while the target power is off.• Connect the host system, the TRACE32 hardware and the debug cable.• Power ON the TRACE32 hardware.• Start the TRACE32 software to load the debugger firmware.• Connect the debug cable to the target.• Switch the target power ON.• Configure your debugger e.g. via a start-up script. <p>Power down:</p> <ul style="list-style-type: none">• Switch off the target power.• Disconnect the debug cable from the target.• Close the TRACE32 software.• Power OFF the TRACE32 hardware.
-----------------	---

Introduction

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

Please note that only the **Processor Architecture Manual** (the document you are currently reading) is specific to the core architecture. All other parts of the online help are general and independent of any core architecture. Therefore, if you have questions related to the core architecture, the **Processor Architecture Manual** should be your primary reference.

Brief Overview of Documents for New Users

Architecture-independent information:

- **“Debugger Tutorial”** (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **“General Commands”** (general_ref_<x>.pdf): Alphabetic list of debug commands.
- **“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.

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**.
- **“XCP Debug Back-End”** (backend_xcp.pdf): This manual describes how to debug a target over a 3rd-party tool using the XCP protocol.

Available Tools

This chapter gives an overview of available Lauterbach tools for the TriCore architecture.

Debugger

Debugging an Infineon TriCore device requires a Lauterbach Debug Cable together with a Lauterbach Debug Module.

To connect to the target the following Debug Cables can be used:

- Debugger for TriCore Automotive (DEBUG-TRICORE-AUTO or DEBUG-MPC-TC-AUTO)
- Debugger for TriCore (Standard) (OCDS-TRICORE)



The Debug Cable comes with a license for debugging. Detailed information is available in chapter [“Debug Cables and CombiProbe Whiskers”](#) in Application Note Debug Cable TriCore, page 22 (app_tricore_ocds.pdf). Lauterbach also offers various [“Adapter 16-pin 100 mil to 50 mil”](#) in Application Note Debug Cable TriCore, page 45 (app_tricore_ocds.pdf).

Furthermore it is required to use a Debug Module from the POWER series, e.g.

- POWER DEBUG INTERFACE / USB 3
- POWER DEBUG INTERFACE / USB 2
- POWER DEBUG PRO
- POWER TRACE / ETHERNET
- POWER TRACE II / POWER TRACE III

The DEBUG INTERFACE (LA-7701) cannot be used for debugging TriCore.

Software-only Debugger for XCP

TRACE32 supports debugging over a 3rd-party tool using the XCP protocol. For details see [Parallel Usage of a 3rd-Party Tool](#) and [“XCP Debug Back-End”](#) (backend_xcp.pdf).

On-chip Trace

On-chip tracing requires no extra Lauterbach hardware, it can be configured and read out with a regular [Debugger](#). The trace related features are enabled with the Trace License for TriCore ED (TriCore-MCDS). Note that only TriCore Emulation Devices (ED) provide an onchip trace buffer.

Serial Off-chip Trace (AGBT)

Lauterbach offers an off-chip trace solutions for AURIX devices equipped with a serial off-chip trace port (Aurora Giga-Bit Trace, AGBT).

Tracing requires either the PREPROCESSOR SERIAL licensed for the TriCore AGBT and a POWER TRACE module,



or a POWER TRACE SERIAL licensed for the TriCore AGBT. The TRACE32 online help provides a [“PowerTrace Serial User’s Guide”](#) (serialtrace_user.pdf), please refer to this manual if you are interested in details about PowerTrace Serial.



Parallel Off-chip Trace

For devices of the AUDO-NG family, a parallel off-chip trace is available. For more information, see chapter [“Parallel Off-chip Trace - OCDS-L2 Flow Trace \(Analyzer\)”](#), page 134.

Co-Processor Debugging (PCP/GTM)

Debugging the Peripheral Control Processor (PCP) or Generic Timer Module (GTM) comes free of charge with the TriCore debug license, i.e. an additional license is not required.

Details are available in the [“PCP Debugger Reference”](#) (debugger_pcp.pdf) and [“GTM Debugger and Trace”](#) (debugger_gtm.pdf) manuals.

Co-Processor Debugging (HSM)

Debugging the Hardware Security Module (HSM) requires a Cortex-M license in addition to the TriCore debug license.

Information, documentation, demo files and support for HSM debugging are available for TRACE32 users who have a valid HSM-related NDA with Infineon. To receive the HSM Support Package, please contact the Lauterbach support team. Lauterbach will send the HSM Support Package after written approval from Infineon.

Multicore Debugging and Tracing

Lauterbach offers multicore debugging and tracing solutions (particularly for AURIX devices), which can be done in two different setups: Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing (AMP). For details, see [“AMP vs. SMP vs. iAMP”](#), page 23.

Multicore debugging can be activated with the License for Multicore Debugging (MULTICORE) in case you only have a single TriCore license.

AURIX devices can be traced with an [On-chip Trace](#) or a [Serial Off-chip Trace \(AGBT\)](#) if a connector is available on the target.

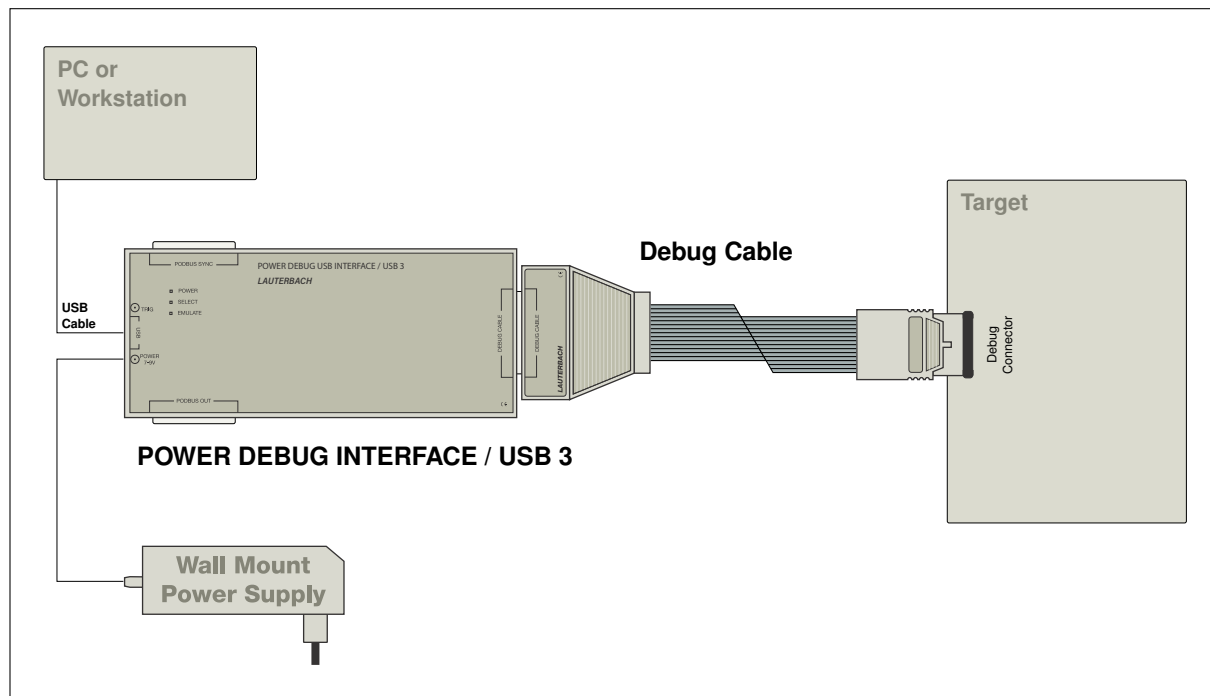
Software Installation

Please follow chapter [“Software Installation”](#) in TRACE32 Installation Guide, page 20 (installation.pdf) on how to install the TRACE32 software:

- An installer is available for a complete TRACE32 installation under Windows. See [“MS Windows”](#) in TRACE32 Installation Guide, page 21 (installation.pdf).
- For a complete installation of TRACE32 under Linux, see [“PC_LINUX”](#) in TRACE32 Installation Guide, page 23 (installation.pdf).

System Overview

This figure shows an example of how to connect the TRACE32 hardware to your PC and your target board.



Related Documents

- [“Onchip/NOR FLASH Programming User’s Guide”](#) (norflash.pdf): Onchip FLASH and off-chip NOR FLASH programming.
- [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf): All about the TriCore debug cables and available accessories.
- [“Training Basic SMP Debugging”](#) (training_debugger_smp.pdf): SMP debugging.
- [“MCDS User’s Guide”](#) (mcds_user.pdf): Basic and advanced information on the MCDS and the Infineon Emulation Devices.
- [“PCP Debugger Reference”](#) (debugger_pcp.pdf): Debugging and tracing the Peripheral Control Processor (PCP).
- [“GTM Debugger and Trace”](#) (debugger_gtm.pdf): Debugging and tracing the Generic Timer Module (GTM).
- [“XCP Debug Back-End”](#) (backend_xcp.pdf): Debugging over a 3rd-party tool using the XCP protocol.

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for all TriCore chips.

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/tricore/` subfolder of the system directory of TRACE32.

OCDS Levels

OCDS (On-chip Debug Solution) is the on-chip logic that implements the debug, trace and calibration functionality. Infineon defines three OCDS levels:

- *OCDS-L1* refers to the basic debug functionality (execution control, register and memory access). The 16-pin JTAG connector is often called OCDS-L1 connector.
- *OCDS-L2* refers to the (deprecated) parallel off-chip trace. The 40-pin AMP connector and the 60-pin SAMTEC connector for the parallel trace is called OCDS-L2 connector.
- *OCDS-L3* is the trace, trigger and calibration functionality in the MCDS. Although defined by Infineon the term is rarely used. Instead the term MCDS is used. MCDS is accessed by the debugger via the debug port (JTAG or DAP), the connector for the AGBT high-speed serial trace is called AGBT connector.

Today the differentiation between the OCDS levels is not very common any more. OCDS refers to the debugging features and MCDS to the trace- and calibration features. Calibration is not supported by TRACE32, but TRACE32 cooperates with third-party tools performing calibration.

Debugging, as described in this chapter, includes:

- Core debugging (run control, breakpoints)
- Memory and register access
- Trace configuration
- Additional debug features (performance counter, run-time memory access, ...)

It is performed via classic **JTAG** or alternatively via **DAP**.

This chapter covers the following topics:

- **“Single-Core Debugging (AUDO)”**, page 17
- **“Multicore Debugging (AURIX)”**, page 19
- **“Access Classes”**, page 26
- **“Flash”**, page 30
- **“Onchip Triggers (TrOnchip Window)”**, page 32
- **“Internal Break Bus (JTAG)”**, page 57
- **“Suspending Peripherals”**, page 55
- **“Assembler and Disassembler Options”**, page 46
- **“Parallel Usage of a 3rd-Party Tool”**, page 47
- **“Internal Break Bus (JTAG)”**, page 57
- **“Troubleshooting”**, page 58

Single-Core Debugging (AUDO)

This chapter introduces a typical start-up sequence for single-core debugging. Depending on your application not all steps might be necessary. The example shown uses a TC1766 B-Step on an Infineon TriBoard-TC1766.

For other chips and more examples, see the chapter [Demo and Start-up Scripts](#).

Single-Core Debugging - Quick Start

1. Prepare the start.



To prevent damage please take care to follow this sequence all the time you are preparing a 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 for TriCore.
- Power up your target.

2. Inform TRACE32 which chip you want to debug.

```
SYStem.CPU TC1766
```

3. Establish the communication between the debugger and your target chip.

```
SYStem.Up
```

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

4. Load your application program.

```
Data.LOAD.Elf myprog.elf
```

The sub-command of the **Data.LOAD** command (here, Elf) depends on the file format generated by the compiler. A detailed description of the **Data.LOAD** command is given in “[General Commands Reference](#)”.

If your application runs from FLASH, FLASH programming has to be enabled before the **Data.LOAD** command is used. Scripts that enable FLASH programming can be found in `~/demo/tricore/flash`.

5. Write a start-up script.

It is recommended to prepare a start-up script that performs the start-up sequence of the commands. This sequence can be written to a PRACTICE script file (*.cmm, ASCII format) and executed with the command **DO** <file>.

Here is a typical start sequence:

```
WinCLEAR                                ; clear all windows

SYStem.RESet

SYStem.CPU TC1766                        ; select CPU

SYStem.Up                               ; reset the target and enter debug
                                         ; mode

Data.LOAD Elf myprog.elf                ; load the application

List.auto                               ; open a source listing          *)

Register.view /SpotLight                 ; open register window          *)

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

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

PER.view                               ; open window with peripheral
                                         ; register                      *)

Break.Set main                           ; set breakpoint to function main
```

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

This chapter covers the following topics:

- [“SMP Debugging - Quick Start”](#), page 19
- [“AMP Debugging - Quick Start”](#), page 21
- [“AMP vs. SMP vs. iAMP”](#), page 23
- [“Selecting the right AURIX CPU”](#), page 24
- [“Understanding Multicore Startup by Application Code”](#), page 24
- [“About Ambiguous Symbols”](#), page 25

SMP Debugging - Quick Start

1. Inform TRACE32 which chip you want to debug.

```
SYStem.CPU TC275TE
```

2. Inform TRACE32 which cores form the SMP system.

Select the cores you want to debug. Do this with the [CORE.ASSIGN](#) command before running the [SYStem.Up](#) command, e.g.:

```
CORE.ASSIGN 1. 2. 3. ; assign cores to an SMP  
; system
```

3. Establish the communication between the debugger and all cores of the SMP system.

```
SYStem.Up
```

This command resets the chip and enters debug mode. After reset only TC 1.6.1 CPU0 is ready, TC 1.6.1 CPU1 and TC 1.6.1 CPU2 have to be initialized by code running on TC 1.6.1 CPU0.

4. Convenient debugging

The command [CORE.select](#) allows to switch between cores (also visible and changeable via the [“Status Bar”](#) in PowerView User’s Guide, page 29 (ide_user.pdf)). Many commands also offer the option [/CORE <core>](#), e.g.:

```
CORE.select 0. ; select core 0  
Register.view ; view registers of  
; currently selected core  
; i.e. core 0  
Register.view /CORE 1. ; view registers of core  
; 1
```

5. Summary

Refer to the SMP demo scripts for details, e.g.:

```
PSTEP tc275te_smp_demo_multisieve.cmm
```

The above script above can be found in `~/demo/tricore/hardware/triboard-tc2x5/tc275te`.

Copy the demo script and adjust it to your needs to get started with your own application in SMP mode.

Master-Slave Script Concept

For AMP it is recommended to write one start-up script, which is intended to run on the “master” GUI controlling the “slave” GUIs via [InterCom](#) commands.

Hint: Setting up user-defined commands with [ON CMD](#) improves readability, e.g.:

```
; startup_script.cmm running on GUI0:
&addressGUI2="127.0.0.1:20002"                ; address and port of
                                                ; GUI2
ON CMD CORE2 GOSUB                             ; define command "CORE2"
(
    LOCAL &params
    ENTRY %Line &params
    InterCom.execute &addressGUI2 &params      ; execute command on
                                                ; remote GUI
    RETURN
)
CORE2 PRINT "executed on core 2"              ; use the user-defined
                                                ; command: text will be
                                                ; printed on GUI2
```

1. Start the TRACE32 PowerView GUIs

To set up an AMP multicore debugging scenario, multiple TRACE32 instances (GUIs) need to be started. Please make sure [InterCom](#) is enabled and the “[Configuration File](#)” in TRACE32 Installation Guide, page 35 (installation.pdf) contains:

```
CORE=<integer>                                ; e.g. CORE=1
IC=NETASSIST                                  ; enable InterCom
PORT=<port_number>                           ; e.g. PORT=20000
```

2. Set up the Multicore Environment

After starting the TRACE32 PowerView GUIs, each instance assumes to be connected to a separate chip by default. Mounting the cores into the same chip makes TRACE32 aware of the resources to be shared between the cores. This is especially important for multicore synchronization and shared resources like the on- and off-chip trace (MCDS, AGBT).

Before bringing the system up, use the [SYSTEM.CONFIG.CORE](#) command on each GUI to mount all cores into one chip, e.g.:

```
CORE0 SYStem.CONFIG CORE 1. 1.                ; GUI0: core 0 in chip 1
CORE1 SYStem.CONFIG CORE 2. 1.                ; GUI1: core 1 in chip 1
CORE2 SYStem.CONFIG CORE 3. 1.                ; GUI2: core 2 in chip 1
```

3. Synchronize Go / Step / Break

The **SYnch** command allows for start stop synchronization between multiple GUIs.

4. Summary

All steps described above are included in the AMP multisieve demo scripts, e.g.:

```
; step through AMP demo script
PSTEP tc275te_amp_demo_multisieve.cmm
```

Copy the demo script and adjust it to your needs to get started debugging your own application in AMP mode.

The above script above can be found in `~/demo/tricore/hardware/triboard-tc2x5/tc275te`.

iAMP Debugging

iAMP allows to have multiple cores in one GUI but to have different symbols for each core. For details, see **“Application Note for iAMP Debugging”** (app_iamp.pdf).

AMP vs. SMP vs. iAMP

For multicore debugging and tracing, TRACE32 supports three different setups:

- Symmetric Multiprocessing (SMP)
 - Homogeneous (e.g. three TriCore CPUs)
- Asymmetric Multiprocessing (AMP)
 - Homogeneous AMP (e.g. three TriCore CPUs)
 - Heterogeneous AMP (e.g. TriCore, GTM, HSM CPUs)
- integrated Asymmetric Multiprocessing (iAMP)
 - Homogeneous (e.g. three TriCore CPUs)

The following table gives an overview of the differences between SMP and AMP:

SMP	AMP	iAMP
One GUI	Multiple GUIs	One GUI
Controls all cores at the same time.	Control cores individually.	Controls all cores at the same time.
Cores are automatically synchronized, i.e. all cores will run at the same time and halt at the same time.	GUIs can be synchronized (Go/Step/Break/SystemMode), see SYnch / InterCom	Cores are automatically synchronized, i.e. all cores will run at the same time and halt at the same time.
Same onchip breakpoints on each core	Individual onchip breakpoints per core	Onchip breakpoints per machine (group of cores)
Recommended for RTOS (esp. with dynamic core/thread assignment)	Required for heterogenous systems, e.g. debugging TriCore concurrently with GTM core	Can be used in case of different RTOS on different cores,
More: “SMP Debugging - Quick Start” , page 19	More: “AMP Debugging - Quick Start” , page 21	More: “Application Note for iAMP Debugging” (app_iamp.pdf)

The debug concept is explained in www.lauterbach.com/publications/debugging_amp_smp_systems.pdf and [“Application Note for iAMP Debugging”](#) (app_iamp.pdf).

To get started with a new AURIX target, we recommend to start with our [“Demo and Start-up Scripts”](#), page 15. (~~/demo/tricore/hardware/):

```
PSTEP ~~/demo/tricore/hardware/*.cmm ; step through a demo
; script
```

Selecting the right AURIX CPU

An AURIX CPU can be selected with the **SYStem.CPU** command as usual. Please note that for all TC27x devices there are two variants available per device. Devices with the `-Astep` suffix reflect the major silicon stepping “A”. When using a device with another major stepping, select a CPU without the `-Astep` suffix.

The following table gives some examples:

<code>SYStem.CPU TC275TE-Astep</code>	<code>; e.g. TC277TE-64F200S AA EES</code>
<code>SYStem.CPU TC275TF</code>	<code>; e.g. TC275TF-64F200W BC EES</code>
<code>SYStem.CPU TC275TP</code>	<code>; e.g. TC275TP-64F200W CA EES</code>
<code>SYStem.DETECT.CPU</code>	<code>; Detect the device automatically</code>
<code>PRINT CHIP.STEPping()</code>	<code>; Stepping is available with a ; function (for scripting)</code>

The connected device can be detected automatically with **SYStem.DETECT.CPU**.

Understanding Multicore Startup by Application Code

After bringing the system up (**SYStem.Up**), all cores are halted at their respective reset vector.

When starting program execution (**Go**), at first only core 0 will start running application code (your application). The other “slave” cores are yet not initialized, i.e. it is not possible to start them as long as their Program Counter is at the reset vector. TRACE32 indicates this with the special state “running (reset)” or “stopped at reset vector” in the [state line](#).

The other “slave” cores will switch to a special state “running (reset)”, waiting to be initialized and started by core 0. When the debugger detects that the slave cores were initialized and started by core 0, it will switch to “running” state in the [state line](#).

Debugging and tracing is possible right from the reset vector.

About Ambiguous Symbols

Each TriCore core of an AURIX chip is equipped with directly attached memories: Program Scratch Pad Ram (PSPR) and Data Scratch Pad Ram (DSPR), accessible by itself via core-local and global addressing, accessible by other cores via global addressing.

When accessing its own memory, accessing via global addresses has no (!) impact on access speed.

When having different code and therefore different symbols on each core, core-local addressing can easily lead to ambiguous symbols. Imagine a function “foo” at address P:0xC0000100 on core 0 and a completely different function “bar” at address P:0xC0000100 on core 1.

Using global addressing avoids such ambiguities, in the above example “foo” would be at address P:0x70100100 and “bar” at P:0x60100100. The following table illustrates the differences:

e.g. PSPR on TC275TE	Global addressing	Core-local addressing
Core 0	P:0x70100000	P:0xC0000000
Core 1	P:0x60100000	P:0xC0000000
Core 2	P:0x50100000	P:0xC0000000
Summary	Unambiguous addresses	Ambiguous addresses (!)

In general, we strongly recommend to use global addressing only, avoid core-local-addressing whenever possible. You might have to modify the linker script file in your compiler tool chain.

Access Classes

For background information about the term [access class](#), see [“TRACE32 Concepts”](#) (trace32_concepts.pdf).

For TriCore, the following memory access classes are available:

Memory Access Class	Description
P	Program (memory as seen by CPU's instruction fetch)
D	Data
EEC	Emulation Memory on EEC Only available on TriCore Emulation Devices for accessing the Emulation Extension Chip.
IC	Instruction Cache
DC	Data Cache (memory as seen by CPU's data access)
NC	No Cache (access with caching inhibited)

P : and D : display the same memory, the difference is in handling the symbols.

Prepending an E as attribute to the memory class will make memory accesses possible even when the CPU is running. See [SYStem.MemAccess](#) for more information.

[SYStem.Option.DUALPORT](#) will enable the runtime update of almost all windows so memory class attribute E is not necessary. Although the core is not stopped for accessing the memory this can slow down program execution since the CPU and debugger both access the on-chip buses.

With [SYStem.Option.DCREAD ON](#) all data memory accesses (D :) will be redirected cached data memory accesses (DC :). For details, see [“Accessing Cached Memory Areas and Cache Inspection”](#), page 42.

Breakpoints

TRACE32 uses two techniques to implement breakpoints: Software breakpoints and on-chip breakpoints.

Software Breakpoints

Software breakpoints are only available for program breakpoints. If a program breakpoint is set to an instruction, the original instruction at the breakpoint location is patched by a break code. This patching is the reason why software breakpoints are usually only used in RAM areas.

Software breakpoints in FLASH memory are also supported (see [FLASH.AUTO](#) in “[General Commands Reference Guide F](#)” (general_ref_f.pdf), chapter [FLASH](#)), but their use is not recommended since they drastically reduce the FLASH's lifetime.

On-chip Breakpoints

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

On-chip Breakpoints	Instruction Breakpoints	Read / Write Breakpoints	Data Breakpoints
(up to AUDO-FG) up to 4 instruction, up to 4 read/write	 up to 4 single or up to 2 ranges	 up to 4 single or up to 2 ranges	—
(AUDO-MAX, AURIX) up to 8	 up to 8 single address or up to 4 ranges	 up to 8 single address or up to 4 ranges	—

You can view your currently set breakpoints with the command [Break.List](#).

If no more on-chip breakpoints are available, you will get an error message when trying to set one.

Intrusive Breakpoints

TRACE32 PowerView uses so-called **intrusive breakpoint** to implement a breakpoint, if the core in use does not provide the means to set a complex breakpoint.

```
; stop the program execution at the 1000. call of the function sieve
Break.Set sieve /Program /COUNT 1000.

; stop the program execution if the byte 0x0 is written
; to the variable flags[3]
Var.Break.Set flags[3] /Write /DATA.Byte 0x0
```

Note	Only one intrusive breakpoints can be set on read/write breakpoints.
-------------	--

MAP.BOnchip Command

Previously it was necessary to inform the debugger about the location of read-only memory on the target. This allows the debugger to automatically use on-chip breakpoints. Meanwhile, the debugger automatically sets Onchip breakpoints when read-only memory is detected. Today the command **MAP.BOnchip** *<address_range>* is rarely used.

Advanced Breakpoints

When using an Infineon Emulation Device, the MCDS part of the Emulation Extension Chip allows to set advanced breakpoints.

Single Stepping

OCDS does not offer a single-stepping feature so this has to be emulated by the debugger. We differentiate between single-stepping on assembler level and single-stepping on HLL level.

Assembler Level

For single-stepping on assembler level, use the **Step.Asm** command. An assembler level single-step is performed by default in **mixed** or **assembler mode**.

Use **SYStem.Option.IMASKASM** to control whether interrupts should be enabled during single-stepping.

On-chip Breakpoint Ranges (Default)

By default, the TriCore uses on-chip breakpoint ranges for implementing single-stepping. This allows the detection of exceptions (interrupts, traps) or any abnormal program execution.

SYStem.Option.STEPSOFT and **SYStem.Option.STEPONCHIP** both have to be disabled (OFF) for activating the default behavior.

Software Breakpoints

With **SYStem.Option.STEPSOFT** one or two software breakpoint instructions are used for single-stepping. This is required when the Memory Protection Unit shall be used for memory protection. Note that software breakpoints can also be used in FLASH memory. A special setup is required for this to work.

As side effect, the interrupt handler may run silently in the background without the user noticing. So data required for the current function may change while stepping, thus resulting in a different behavior.

On-chip Breakpoint

With **SYStem.Option.STEPONCHIP** an on-chip breakpoint with Break-After-Make is used for single-stepping.

This is mainly used as a workaround for a silicon bug (CPU_TC.115) which prevents a proper suspend of an already taken interrupt. As a result, the interrupt handler will run silently in the background without the user noticing. So data required for the current function may change while stepping, thus resulting in a different behavior.

In case **SYStem.Option.STEPSOFT** is enabled at the same time, **SYStem.Option.STEPONCHIP** will have no effect.

HLL Level

For single-stepping on HLL level, use the **Step.Hll** command. An HLL level single-step is performed by default in **HLL mode**.

Use **SYStem.Option.IMASKHLL** to control whether interrupts should be enabled during single-stepping.

There are no options to configure the behavior, the debugger will always choose the correct implementation depending on the kind of memory.

Programming the flash memory is explained in detail in chapter “**FLASH**” in General Commands Reference Guide F, page 25 (general_ref_f.pdf). This chapter will only give a short overview.

TriCore devices can be efficiently programmed with target controlled flash algorithms. They require a flash declaration, which can be found in the `~/demo/tricore/flash/` directory of the TRACE32 installation.

After the flash is declared, TRACE32 offers three approaches:

- **FLASH.Erase** and **FLASH.Program**
- **FLASH.Auto**
- **FLASH.ReProgram** (recommended)

Flashing an application typically follows these steps:

1. Prepare flash programming (declarations)
 - Using **FLASH.Create** commands
 - Or using provided scripts in `~/demo/tricore/flash/`
2. Enable flash programming using **FLASH.ReProgram ALL**.
3. Load an application using **Data.LOAD** *<file>*.
4. Finally program the flash using **FLASH.ReProgram off**.

All these steps, including device-specific declarations, are provided in the flash demo scripts. Use

```
PSTEP ~/demo/tricore/flash/tc*.cmm ; run flash script
```

for a comprehensive step-by-step procedure.

General help for flashing is available in chapter “**FLASH**” in General Commands Reference Guide F, page 25 (general_ref_f.pdf).



AURIX devices have several security features. Be careful not to lock yourself out when flashing a device!

- **Boot Mode Headers (BMHD):**
Do not reboot or unpower your device in case all BMHD (Boot Mode Headers) do not contain valid information. This is normally the case after having erased the internal program flash or loading an object or binary file without a valid BMHD.
- **Hardware Security Module (HSM):**
Do not enable HSM boot when no valid HSM code is present. This will lock your device permanently. See also “**Flashing the Hardware Security Module (HSM)**”, page 32.
- **User Configuration Blocks (UCB):**
Pay special attention when modifying the UCB. An invalid or erroneous content will lock your device permanently. This also happens in case the confirmation code is neither “unlocked” nor “confirmed”.

The onchip flash on AURIX devices has three states: 0, 1 and Bus Error.

By default, an erased flash sector returns a Bus Error on read access, indicated as “???????” in a **Data.dump** window. It is caused by an ECC error in the chip, which can be changed with the `FLASH0_MARP.TRAPDIS` and `FLASH0_MARD.TRAPDIS` register fields.

If you prefer to have an erased flash set to zero, simply run **Data.Set** `<range> %Long 0x0` before loading your application, e.g.:

```
FLASH.ReProgram 0xA0080000--          ; Activate flash region
                    0xA009FFFF /Erase   ; for programming

Data.Set 0xA0080000--0xA009FFFF %Long 0x0    ; set to zero

Data.LOAD.Elf myapp.elf                ; load ELF file

FLASH.ReProgram off                    ; Erase and program
                                         ; modified region
```

To make sure that another core is not intervening while flashing, it is recommended to switch to single-core debugging before flashing an application. All required steps are demonstrated in the flash script, e.g. in `~/demo/tricore/flash/tc27x.cmm`:

```
DO ~/demo/tricore/flash/tc2*.cmm      ; run flash script
```

After flashing the application, a multicore debug scenario can be started as usual.

Hint: Use the option `/NoCODE` of the **Data.LOAD** command to load only the symbols (without downloading code), e.g.:

```
Data.LOAD.elf my_application.elf /NoCODE ; Load just the symbolic
                                           ; information of an al-
                                           ; ready flashed appli-
                                           ; cation
```

Flashing the Hardware Security Module (HSM)

When flashing the HSM, use the same precautions as described in **“Flashing and Debugging AURIX Devices”**, page 31.

The HSM can be flashed with the scripts `~/demo/tricore/flash/tc2*-hsm.cmm`:

```
DO ~/demo/tricore/flash/tc2*-hsm.cmm ; run flash script
```



Do not enable HSM boot when no valid HSM code is present. This will lock your device permanently. See the Infineon documentation and contact your Infineon FAE for more information on HSM.

Onchip Triggers (TrOnchip Window)

Onchip triggers can be influenced with the **TrOnchip.view** window. It affects primarily **“On-chip Breakpoints”**, page 27, and allows to control the Multi-Core Break Switch (MCBS) providing two internal Break Buses and a Suspend Switch.

See the **TrOnchip** commands for more information.

BenchMarkCounter

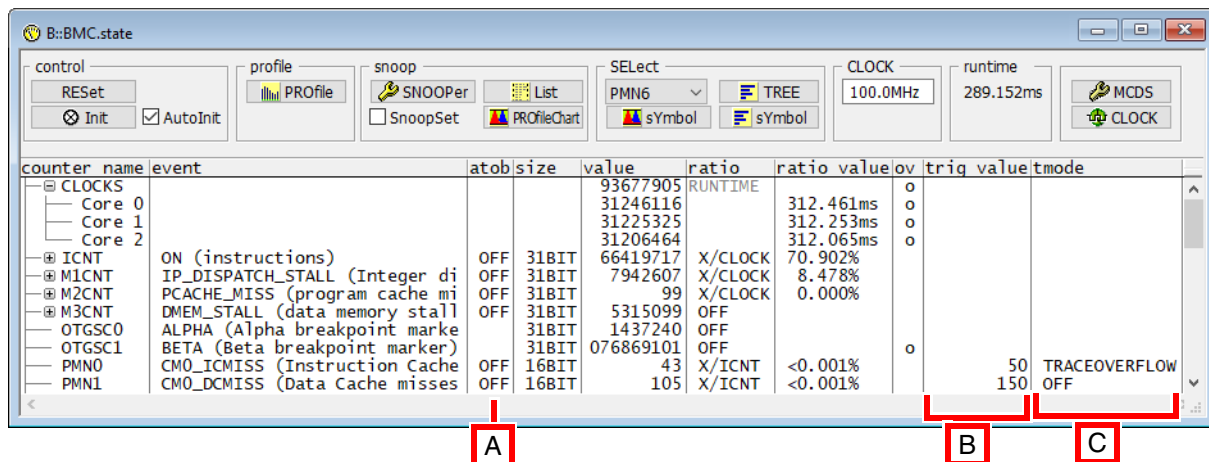
Benchmark counters are on-chip counters that count specific hardware events, e.g., the number of executed instructions. This allows to calculate typical performance metrics like clocks per instruction (CPI). The benchmark counters can be read at run-time.

TriCore CPUs support three different types of benchmark counters:

- **Performance counters** in the core debug controller of each core. These counters are available in AUDO MAX and AURIX CPUs.
 - Clock counter CCNT.
 - Instruction counter ICNT.
 - Multi counters M1CNT, M2CNT, M3CNT that can be configured to count specific events, e.g., cache hits and misses. The available events depend on the used device.
- **Event Counters** in the OTGS. These counters are only available for AURIX and later devices. These counters can capture events generated by breakpoints with action alpha, beta, charly, delta or echo.
- **Event counters** in the MCDS. These counters are only available in emulation devices. For AUDO CPUs these counters are named CNT0 to CNT15. For AURIX CPUs these counters are named PMN0 to PMN31. Refer to **“Benchmark Counters”** in MCDS User’s Guide, page 40 (mcds_user.pdf) for more information.

In TRACE32, these benchmark counters can be programmed through the **BMC.state** window.

Performance counters and event counters of TriCore CPUs can be started or stopped using on-chip breakpoints. This A-to-B mode allows to determine performance metrics for a single code block. This mode is available for AUDO MAX CPUs with a TC1.6 core and all AURIX CPUs.

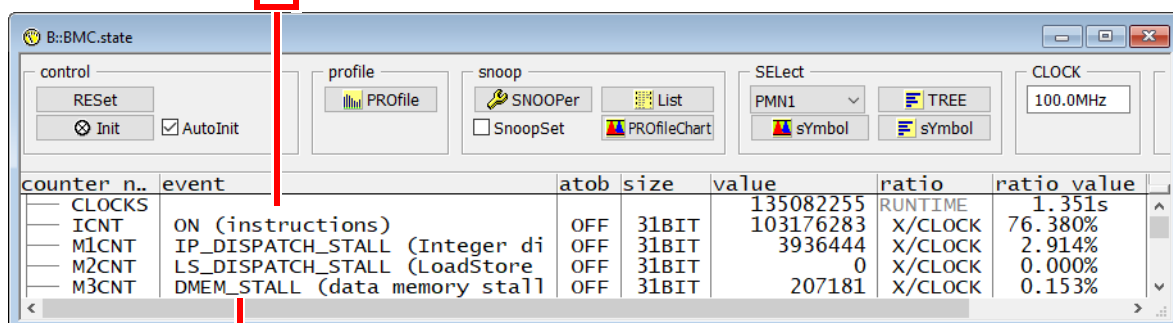


- A** Enable or Disable A-to-B mode. In A-to-B mode only events between the Alpha and Beta marker are counted. See **BMC.<counter>.ATOB**.
- B** Defines the BenchMarkCounter trigger value. See **BMC.<counter>.TRIGVAL**.
- C** Enables/disables the BenchMarkCounter trigger. See **BMC.<counter>.TRIGMODE**.

Example: Measuring Instructions and Stalls per Clock Cycle

This example explains how to measure the number of instructions [A] and the stalls per clock cycle [B]. The analysis can be set up in the **BMC.state** window:

A



counter n..	event	atob	size	value	ratio	ratio value
CLOCKS				135082255	RUNTIME	1.351s
ICNT	ON (instructions)	OFF	31BIT	103176283	X/CLOCK	76.380%
M1CNT	IP_DISPATCH_STALL (Integer di	OFF	31BIT	3936444	X/CLOCK	2.914%
M2CNT	LS_DISPATCH_STALL (LoadStore	OFF	31BIT	0	X/CLOCK	0.000%
M3CNT	DMEM_STALL (data memory stall	OFF	31BIT	207181	X/CLOCK	0.153%

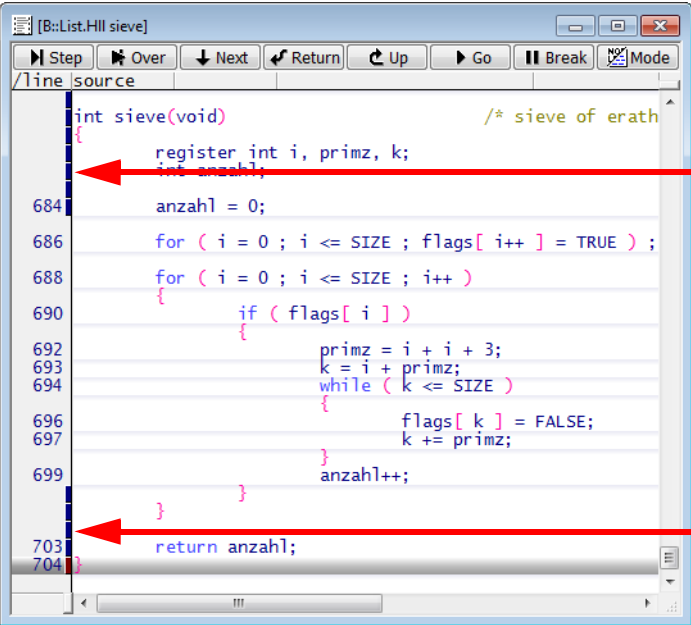
B

or using the following commands in PRACTICE:

```
BMC.CLOCK 100.Mhz ; Set the BMC Clock
BMC.ICNT.EVENT ON ; Set the Events
BMC.M1CNT.EVENT IP_DISPATCH_STALL
BMC.M2CNT.EVENT LS_DISPATCH_STALL
BMC.M3CNT.EVENT DMEM_STALL
BMC.ICNT.RATIO X/CLOCK ; Set the counters RATIO
BMC.M1CNT.RATIO X/CLOCK
BMC.M2CNT.RATIO X/CLOCK
BMC.M3CNT.RATIO X/CLOCK
```

Example: A-to-B Mode (single shot)

In the following example, the runtime of the function `sieve()` is measured, and the instructions executed during runtime are counted. The result is displayed in the **BMC.state** window.

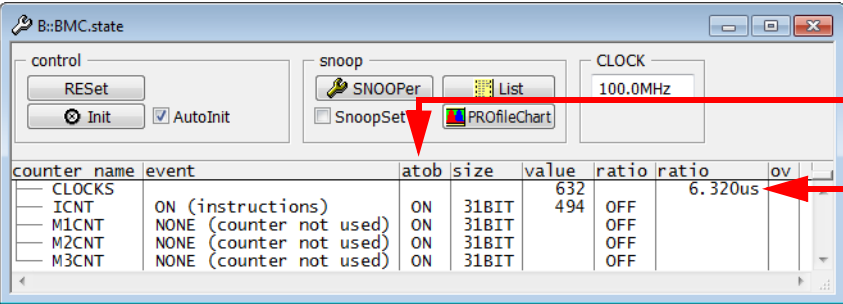


```
int sieve(void) /* sieve of erath
{
    register int i, primz, k;
    int anzahl;
684    anzahl = 0;
686    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
688    for ( i = 0 ; i <= SIZE ; i++ )
690    {
        if ( flags[ i ] )
        {
692            primz = i + i + 3;
693            k = i + primz;
694            while ( k <= SIZE )
696            {
                flags[ k ] = FALSE;
                k += primz;
697            }
699            anzahl++;
    }
703    return anzahl;
704 }
```

Alpha breakpoint
Break.Set sieve /Program /Alpha

Instructions are only counted in this range.

Beta breakpoint
Break.Set sYmbol.EXIT(sieve)-2 /Program /Beta



A-to-b mode active

counter	name	event	atob	size	value	ratio	ratio	ov
CLOCKS					632			
ICNT		ON (instructions)	ON	318BIT	494	OFF	6.320us	
M1CNT		NONE (counter not used)	ON	318BIT		OFF		
M2CNT		NONE (counter not used)	ON	318BIT		OFF		
M3CNT		NONE (counter not used)	ON	318BIT		OFF		

Runtime

This example can be set up using the following PRACTICE script.

```
CLOCK.ON
Break.Set sieve /Program /Alpha
Break.Set sYmbol.EXIT(sieve)-2 /Program /Beta
Break.Set sYmbol.EXIT(sieve)
BMC.ICNT.EVENT ON
BMC.ICNT.ATOB ON
```

It is not possible to have different A-to-B mode settings for the counters of the core debug controller. Executing the command for one of the counters will automatically set the state for the other counters.

For the performance counters of the core debug controller, the alpha and beta marker must not overlap with another breakpoint.

Example: A-to-B Mode (average)

Starting from AURIX devices and release R.2020.02, average runtimes can be measured using a combination of the performance counters in the core debug controller and the event counters in the OTGS.

In the following example, the average runtime of the function `sieve()` is measured. The result is displayed in the **BMC.state** window.

```
CLOCK.ON
Break.SetFunc sieve
BMC.ICNT.EVENT ON
BMC.ICNT.ATOB ON
BMC.OTGSC0.EVENT Alpha
BMC.OTGSC0.RATIO TIME/X

; measure for one second
Go
WAIT 1.s
Break

BMC.state
```

Result:

The screenshot shows the BMC.state window with the following data table:

counter_name	event	atob	size	value	ratio	ratio value	ov
CLOCKS				18321732	RUNTIME	183.217ms	
ICNT	ON (instructions)	ON	31BIT	15855345	OFF		
M1CNT	NONE (counter not used)	ON	31BIT		OFF		
M2CNT	NONE (counter not used)	ON	31BIT		OFF		
M3CNT	NONE (counter not used)	ON	31BIT		OFF		
OTGSC0	ALPHA (Alpha breakpoint marker)		31BIT	32031	TIME/X	5.720us	
OTGSC1	NONE (counter not used)		31BIT		OFF		

A complex example recording the results in **BTrace** can be found in the demo folder under `~/demo/tricore/etc/bmc`.

Example: Record Counters Periodically

The TRACE32 SNOOPer trace can be used to record the event counters periodically, if the target system allows to read the event counters while the program execution is running.

```
BMC.state                                ; display the BMC configuration
                                         ; window

BMC.M1CNT.EVENT DATA_X_HIT              ; count data cache / data buffer
                                         ; hits

BMC.M2CNT.EVENT DATA_X_CLEAN            ; count data cache / data buffer
                                         ; misses

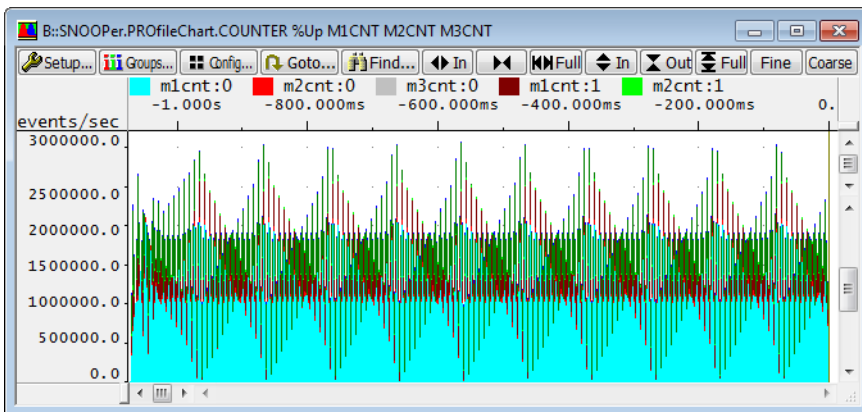
BMC.SnoopSet ON                          ; configure the TRACE32 SNOOPer
                                         ; trace for event counter recording

Go                                       ; start the program execution to
                                         ; fill the SNOOPer trace

Wait 1.s

Break                                   ; stop the program execution

SNOOPer.PROfileChart.COUNTER %Up         ; display a profile statistic
M1CNT M2CNT M3CNT
```



The **SNOOPer.List** and **SNOOPer.PROfileChart.COUNTER** commands allow to specify the counters you want to display.

Watchpins can be used to pulse a physical output pin on the target or on the debugger when a special breakpoint is hit (without halting the cpu). The breakpoints use the action **/WATCH**. Watchpins are especially useful for analyzing the timing of an application.

Watchpins use the on-chip logic of the chip. Thus, they differ between AUDO and AURIX devices.

AUDO Devices

To set up watchpins on AUDO devices, first set the watch-breakpoint, then route the watch event to an output pin using the TrOnchip.BreakBus feature, e.g.:

```
Break.Set sieve /WATCH                ; set WATCH breakpoint
TrOnchip.BreakBus0.BreakOUT Enable    ; route to pin
```

See the chapter “**CPU specific TrOnchip Commands**”, page 121 for details.

NOTE: The GPIO pins are not driven automatically, they have to be configured manually.

AUDO devices do not allow to set Alpha/Beta/Charly/Delta/Echo breakpoints as watchpoints (as opposed to AURIX devices).

AURIX Devices

AURIX devices can pulse up to eight different output pins, for details see chapter “**Features and Restrictions**”, page 40.

Setup

Setting up watchpins on AURIX devices requires three basic steps:

1. Setting the breakpoint
2. Routing to the pin
3. Changing the speedgrade

All these steps are also shown in a demo script:

```
PSTEP ~/demo/tricore/etc/trace_trigger/watchpins/aurix_watchpin.cmm
```

1. Setting the breakpoint:

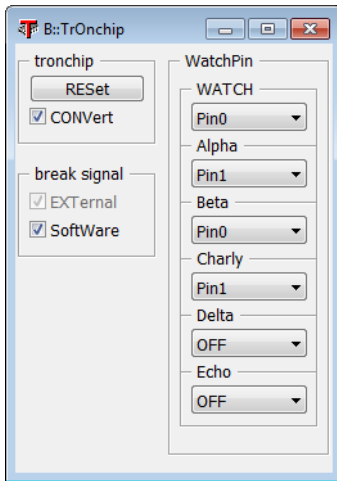
Set a breakpoint with action watch, e.g.:

```
Break.Set sieve /WATCH ; set WATCH breakpoint
```

2. Routing to the pin:

The watch event can be routed to an output pin using the TrOnchip.WatchPin feature, e.g.:

```
TrOnchip.WatchPin.WATCH.Pin0 ; route to pin
```



NOTE:

Enabling WatchPins automatically enables the respective pin as Trigger Output Pin (overriding user configuration), so make sure not to use the GPIO pin for something else at the same time.

AURIX devices allow to set additional watchpoints using the Alpha/Beta/Charly/Delta/Echo breakpoints (as opposed to AUDO devices), e.g.:

```
Break.Set sieve /ALPHA ; set WATCH breakpoint
TrOnchip.WatchPin.ALPHA.Pin1 ; route to pin
```

For a sample pin mapping see chapter [“Features and Restrictions”](#), page 40.

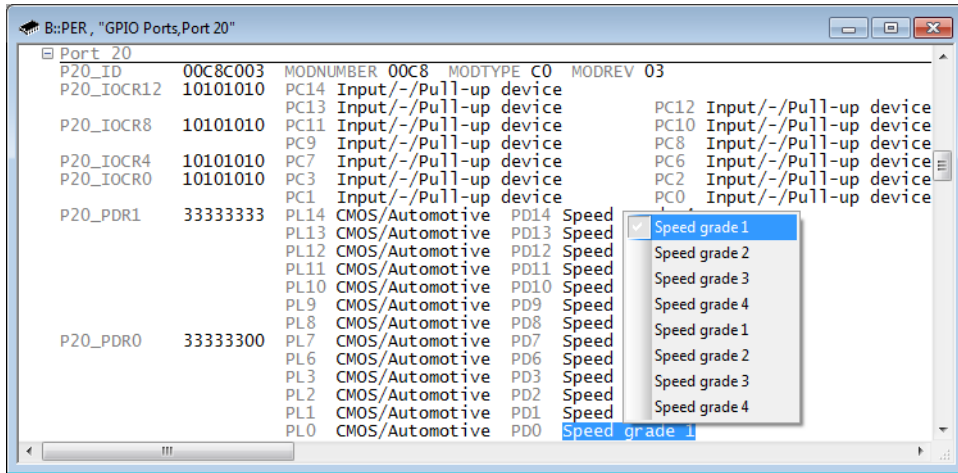
3. Changing the speedgrade:

In order to see short pulses (i.e. watch breakpoints on single addresses or on an address range with only a few hundred instructions) the speed grade of the respective GPIO pin has to be changed manually.

This can be done either by command, e.g.:

```
; set P20_PDR0.PD0 to "Speed Grade 1" (Pad Driver Mode for Port 20 Pin 0)
PER.Set D:0xf003c040 (Data.Long(D:0xf003c040)&(~0x7))
```

Or in the peripheral window, e.g.:



Features and Restrictions

In general there are 8 output pins. Please be aware that they might be used for a different purpose already. For example, it might not be a good idea to use Pin3, which is used for the TDO/DAP2.

Sample pin mapping (See Infineon documentation for details):

Pin	Package Pin	Note
Pin0	P20.0	/BRKOUT
Pin1	P20.1	/BRKIN
Pin2	P21.6	(TDI)
Pin3	P21.7	(TDO, DAP2)
Pin4	P32.6	
Pin5	P32.7	
Pin6	P33.14	
Pin7	P33.15	

There is a maximum of 4 watch breakpoints per core, i.e. 12 watch breakpoints for a device with three cores.

The most restrictive part is the availability of the internal lines for routing the watch breakpoints. There are 7 lines, but some of them are already preoccupied:

- Two of them are already occupied internally
- One line is occupied per core

So there are 2-4 lines still available to be routed dynamically (e.g. 4 lines available for a single-core device, 2 lines when multi-core debugging a device with three cores). TRACE32 tries to find an optimal routing.

To sum up, at most 4 different pins can be used at the same time. Still, multiple watch breakpoints can be routed to the same output pin. This might be especially useful when no more resources are available.

Accessing Cached Memory Areas and Cache Inspection

Most AURIX devices and some AUDO devices have data caches as well as instruction caches. There is one cache per CPU and these caches are located between the CPU and the system bus. The CPU will use this cache for certain address ranges only. These address ranges are referred to as *cached memory areas* in the following.

If a CPU with active cache modifies data in a cached memory area (e.g., the LMU), this modification will be first stored to its local cache only. It will be written back to memory if explicitly requested by the CPU or implicitly when the cache line shall be reused.

Normally, TRACE32 will access memory using the system bus. This will bypass the caches. When you, e.g., view data that is currently cached by a CPU, this will lead to a difference in the data displayed by the debugger and the data seen by the CPU. TRACE32 provides the following methods to view the correct data:

- Use **access class** “DC”. For example, the data at address 0x90040000 (LMU of TC39x) can be displayed using the command
`Data.dump DC:0x90040000`
- Set **SYStem.Option.DCREAD ON** to display the correct cached data when using access class “D”. This access class is used by default for all data accesses. This is especially useful when displaying variables in the **Var.Watch** / **Var.View** window, because they will then be displayed from the CPU's point of view.

The exact behavior differs between AUDO (TC1xx) and AURIX (TC2xx and later) CPUs and is described below.

A similar issue occurs when TRACE32 tries to write data in a memory area that is currently cached by a CPU. An access through the system bus will only update the memory but not the data cached by the CPU. TRACE32 provides different methods to handle this. This is also described below.

Please note, that it is not possible to access cached memory areas while the CPU is running. In this case, physical memory is accessed.

In some use cases (e.g., performance optimization) the exact cache state is of interest. In the following this is referred to as *cache inspection*. The corresponding commands can be found in the **CACHE** command group. For TC2xx and later devices this is subject to certain constraints that are described below.

AUDO Devices

- Reads using access class “DC” will always evaluate the cache tags to present the cached data if applicable.
- Writes using access class “DC” will first flush the respective cache lines and then write the new values to memory.
- Reads using access class “D” with **SYStem.Option.DCREAD ON** will be redirected to access class “DC”.
- Writes using access class “D” with **SYStem.Option.DCFREEZE OFF** will be redirected to access class DC.

Cache inspection is always possible.

This section does not apply to early AURIX devices (TC27x-Astep, TC2Dx). These behave like AUDO devices.

TC1.6E (efficiency) cores implement a small data read buffer instead of a data cache. The data read buffer is not accessible by TRACE32.

For TC1.6P and TC1.6.2 cores TRACE32 provides two different methods how cached memory areas can be read or written:

- *cache evaluation method*: this method will evaluate the cache tags and content to reconstruct the cached memory area on read as well as to update the cache contents on writes.
- *target code method*: this method will execute code on the CPU to read or write cached memory areas.

The cache evaluation method can only be used when certain constraints are fulfilled. This is explained in the following.

CPU Constraints for Cache Evaluation and Cache Inspection

The cache evaluation method as well as cache inspection require that TRACE32 is able to read out the cache tags and contents. AURIX CPUs allow reading this by mapping the cache into the CPU address space using the MTU. As soon as the cache is mapped by TRACE32, the CPU will clean the cache content due to security reasons. In addition, this might be reported to other units as security/safety alarm. This has the following implications:

- The cache must only be mapped when the cache is empty. Otherwise data might be lost.
- The cache must stay mapped during the complete debug session.
- In some use-cases mapping the cache might not be possible at all.

This is considered by TRACE32 as follows:

- **SYStem.Option.MAPCACHE** controls if TRACE32 will map the cache at all. By default it is activated (ON).
- TRACE32 will map the cache only if **SYStem.Option.MAPCACHE ON** and the cache is known to be empty. This is the case after the following commands: **SYStem.Up**, **CACHE.INVALIDATE IC**, **CACHE.INVALIDATE DC**, **CACHE.FLUSH DC**.
- TRACE32 will unmap the cache only if executing **SYStem.Option.MAPCACHE OFF**.

This implies that the cache is not mapped after connecting to the device using **SYStem.Mode.Go**, **SYStem.Mode.Attach** and after power losses or external resets while **SYStem.Option.RESetBehavior RunRestore** is set. In this case, the cache can be manually mapped by halting the CPU and executing **CACHE.INVALIDATE DC**, **CACHE.FLUSH DC** and/or **CACHE.INVALIDATE IC**.

Due to the internal architecture of the CPU (e.g. usage of the store buffers), stores might not become visible in the caches of a CPU until an `isync` or `dsync` instructions is executed. The latter can be done automatically by TRACE32. Please see **SYStem.Option.DSYNC** for further details.

This method will evaluate the cache tags to display and modify the cached data where applicable. This has the advantage that the cache structure (i.e., which memory content is cached) does not change.

On writes, the corresponding physical memory will be modified in parallel. This is done to avoid marking the cache line as dirty and thus triggering writebacks that will not occur during normal operation. This will not change the cache tags.

This method requires a mapped cache. Please see the previous section for more details.

TRACE32 will potentially use this method if **SYStem.Option.MAPCACHE ON**.

In particular:

- Reads and writes using access class “DC” will always use this method. In case the cache is not mapped an error will be returned.
- Reads using access class “D” with **SYStem.Option.DCREAD ON** will first evaluate the core configuration with respect to cache bypass and physical memory attributes. In case the cache is relevant for the current access, it will use the cache evaluation method. Otherwise physical memory is accessed directly. In case the cache is not mapped an error will be shown and physical memory will be accessed.
- Writes using access class “D” will first evaluate the core configuration with respect to cache bypass and physical memory attributes. In case the cache is relevant for the current access it will use the cache evaluation method. Otherwise physical memory is accessed directly. In case the cache is not mapped an error will be shown and physical memory will be accessed.
SYStem.Option.DCFREEZE will be ignored since the cache tags are not changed.

Reading and Writing using Target Code Method

This method implements reads and writes by executing appropriate code on the CPU. This has the advantage that the result always matches the CPU view. This method will modify the cache structure (i.e., which memory content is cached) and trigger additional writebacks. The effort for this method is considerably higher since the CPU state and some target memory needs to be saved and restored.

TRACE32 will potentially use this method if **SYStem.Option.MAPCACHE OFF**:

In particular:

- Reads and writes using access class “DC” will always use this method.
- Reads using access class “D” with **SYStem.Option.DCREAD ON** and **SYStem.Option.DCFREEZE OFF** will first evaluate the core configuration with respect to cache bypass and physical memory attributes. In case the cache is relevant for the current access it will use the target code method. Otherwise physical memory is accessed directly.
- Writes using access class “D” with **SYStem.Option.DCFREEZE OFF** will first evaluate the core configuration with respect to cache bypass and physical memory attributes. In case the cache is relevant for the current access it will use the target code method. Otherwise physical memory is accessed directly.

For further details about target code execution, see “[Target Code Execution](#)”, page 57.

Use Case: No cache used

Set **SYStem.Option.MAPCACHE OFF** in case of unwanted security/safety alerts or concerns about the cache being mapped in the CPU access space.

Use Case: Cache used, Cache Mapping Possible

Leave **SYStem.Option.MAPCACHE ON** to use cache evaluation method.

Consider setting **SYStem.Option.DCREAD ON** for correct display of variables. Consider setting **SYStem.Option.DSYNC ReadWrite** to ensure no data remains in the CPU.

In case you experience problems with unwanted security/safety alerts, see “[Use Case: Cache used, Cache Mapping Not Possible](#)”, page 45

Use Case: Cache Inspection

In case you want to analyze the cache behavior of your application, the cache must be mapped. This means **SYStem.Option.MAPCACHE ON** is required.

Use Case: Cache used, Cache Mapping Not Possible

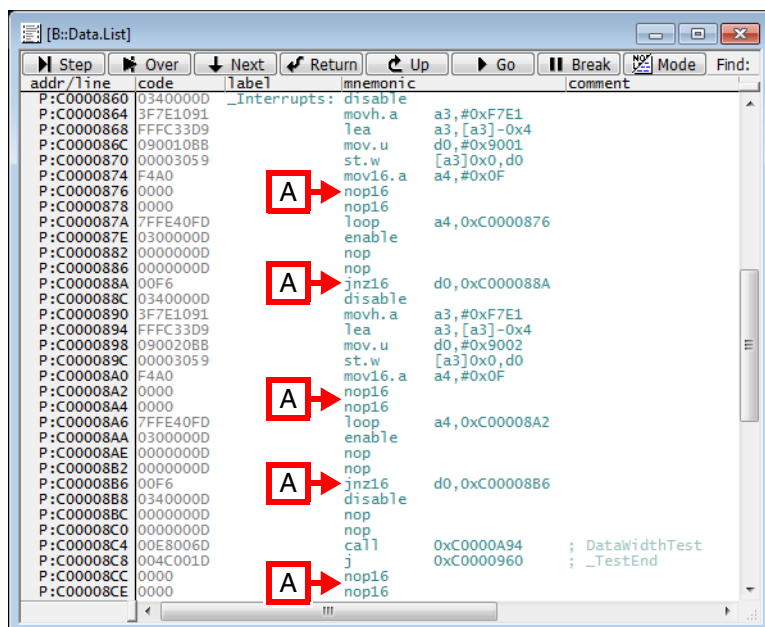
In case you experience problems with unwanted security/safety alerts or you are concerned about the cache being mapped in the CPU access space, use target code method. This means set **SYStem.Option.MAPCACHE OFF** and **SYStem.Option.DCFREEZE OFF**.

Consider setting **SYStem.Option.DCREAD ON** for correct display of variables.

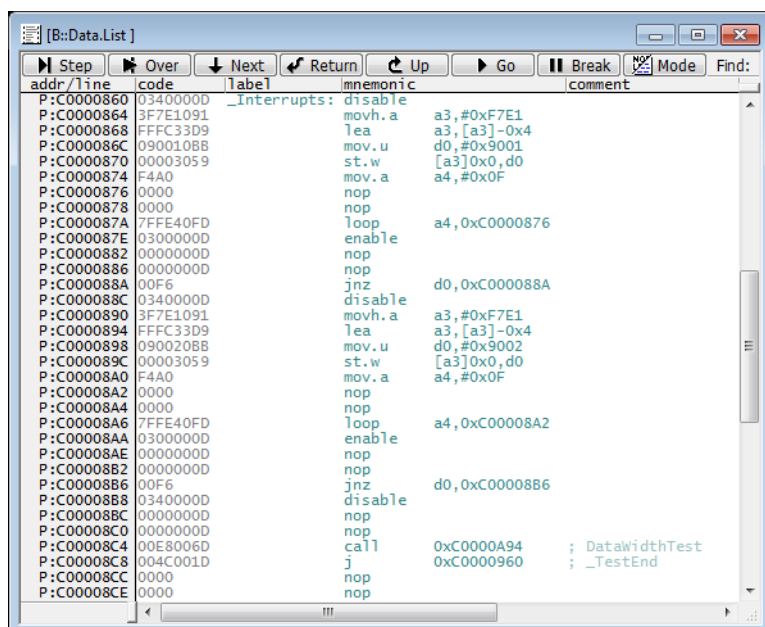
Assembler and Disassembler Options

By default, the disassembler displays all opcodes as defined by Infineon (Simple). Alternatively the 16-bit opcodes can be marked with a “16” tag to identify them more easily (Generic); see [A] in figure below. The **SETUP.DIS** command is used to configure the behavior.

Generic disassembler mode:



Simple disassembler mode:



In case an assembler mnemonic is available in 16-bit as well as in a 32-bit flavor, the 32-bit variant is preferred. To force the usage of the 16-bit equivalent the “16” tag can be used.

Parallel Usage of a 3rd-Party Tool

Some 3rd-party tools require access to the debug port and/or debug resources as well. Examples of 3rd-party tools are tools for measurement, calibration and data-stimulation (MC tools). This section describes how to use such a 3rd-party tool in parallel with TRACE32.

There are different options of how to share the debug port. The appropriate method depends on the 3rd-party tool. In general, TRACE32 supports the following options:

- Software debugging over XCP: In this mode, all debug commands are sent to the 3rd-party tool using the standardized XCP protocol. For details, see [“XCP Debug Back-End”](#) (backend_xcp.pdf).
- Physical sharing: In this mode, the debug signals are shared using a special debug adapter for the 3rd-party tool. This mode is described in the next section.

The 3rd-party tool might also require certain debug resources of the target CPU. Typical examples are:

- The Core Debug Controller covers all debug resources that are part of the CPU like hardware breakpoints.
- The Cerberus block (CBS) covers all chip-wide debug resources. See also [“Cerberus Access Protection”](#), page 56.
- Emulation Memory can be divided up between a calibration tool and TRACE32 to allow calibration and tracing in parallel. For details and configuration, please refer to chapter [“Emulation Memory”](#) in MCDS User’s Guide, page 74 (mcds_user.pdf).

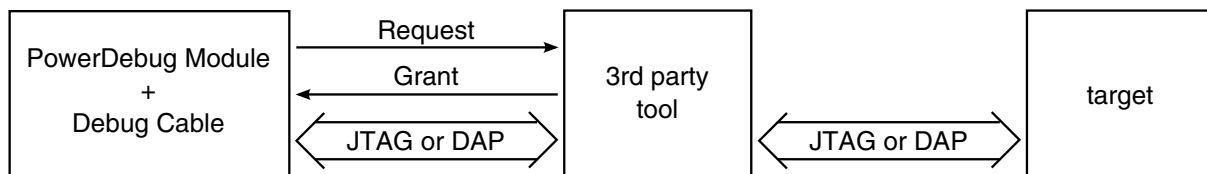
You must make sure TRACE32 is not trying to use these resources in parallel, e.g. by special configuration. Examples for known 3rd-party tools can be found under `~/demo/tricore/etc/`. In doubt, please contact Lauterbach and/or the 3rd-party tool vendor.

NOTE:

Please be aware that some debug operations are intrusive, e.g., [Go](#), [Break](#), [SYStem.Mode Up](#), or FLASH programming, and therefore might interfere with the operation of the 3rd-party tool.

Physical Sharing of the Debug Port

The general configuration is depicted below.



The 3rd-party tool is connected between the debugger and the target. It acts as an arbiter for the JTAG or DAP debug port. In order to communicate with the target, the debugger requests the debug port from the tool and waits until access is granted.

Sharing of the debug port must be configured using **SYStem.CONFIG PortSHaRing** before bringing the system up. Typical start-up sequences:

- JTAG (e.g., ETAS ETK for AUDO CPUs):

```
SYStem.CPU TC1796ED           ; select your CPU
SYStem.CONFIG DEBUGPORTTYPE JTAG ; switch debug port to JTAG
SYStem.CONFIG PortSHaRing ON   ; enable port sharing
SYStem.CONFIG TCKLEVEL 0       ; pull-down on TCK (default)
SYStem.CONFIG TAPState 12.     ; Run-Test/Idle (default)
SYStem.Mode Attach             ; connect to CPU
```

The JTAG clock must be between 10 MHz (default) and 40 MHz.

- DAP (e.g., ETAS ETK-S20 for AURIX CPUs):

```
SYStem.CPU TC277TE           ; select your CPU
SYStem.CONFIG DEBUGPORTTYPE DAP2 ; switch debug port to DAP
SYStem.CONFIG PortSHaRing ON   ; enable port sharing
SYStem.Mode Attach             ; connect to CPU
```

For the most common CPUs and 3rd-party tools, PRACTICE start-up scripts (*.cmm) can be found under `~/demo/tricore/etc/`.

```
PSTEP                               ; step through a demo
~/demo/tricore/etc/debug_connections/portsh ; script
aring_etk/*.cmm
```

Avoid displaying invalid memory locations (Bus Error).

Details about electrical requirements and connector pin-outs can be found in **“Application Note Debug Cable TriCore”** (app_tricore_ocds.pdf).

Debugging an Application with the Memory Protection Unit Enabled

Debugging a TriCore device with the Memory Protection Unit enabled requires different device-dependent strategies.

TriCore v1.6 and Later

For TriCore devices with core architecture v1.6 and later, the Memory Protection Unit and the on-chip breakpoint implementation have been separated. Debugging such a device with the MPU enabled does not require any specific setup or configuration.

TriCore v1.3.1 and Earlier

TriCore devices with a core architecture up to v1.3.1 have the on-chip breakpoint feature implemented in the Memory Protection Unit. This means that either on-chip breakpoints or the MPU can be used, but not both at the same time. Do *not* activate the MPU for single stepping on assembler level.

For simplification debugging an application in RAM is explained first, then the additional configuration for an application located in FLASH.

Debugging with MPU Enabled in RAM

The first step is to prevent the debugger from configuring any on-chip breakpoint:

- 1. Single-stepping

Disable the usage of on-chip breakpoints for single-stepping on assembler level

```
SYSystem.Option.STEPSOFT ON
```

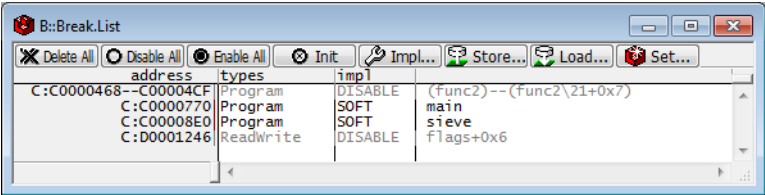
- 2. Program breakpoints

While single addresses are automatically configured as software breakpoints, make sure not to program any address range.

- 3. Data breakpoints

Do not configure any breakpoint on data accesses.

Watch the **Break.List** window and check the configured breakpoints.



Debugging with MPU Enabled in FLASH

For debugging an application running in FLASH using the MPU first apply the same configuration as for debugging in RAM. Additionally perform the following steps:

1. Enable software breakpoints in FLASH by configuring your flash, e.g. as you would do for FLASH programming:

```
; Set up FLASH programming for affected FLASH
FLASH.Create 1. 0xA0000000--0xA000FFFF 0x4000
FLASH.CREATE ...
...
FLASH.TARGET 0xD4000000 0xD0000000 0x1000 tc1797.bin
```

2. Enable FLASH programming in AUTO mode:

```
FLASH.AUTO ALL
```

3. Set software breakpoints as preferred breakpoint implementation:

```
Break.SELect Program SOFT
```

For more information, see:

- FLASH programming and commands: **“FLASH”** in General Commands Reference Guide F, page 25 (general_ref_f.pdf)
- **“Software Breakpoints in FLASH”** in Onchip/NOR FLASH Programming User’s Guide, page 32 (norflash.pdf)

NOTE: Software breakpoints in FLASH drastically reduce the life-time of your FLASH.
--

Debugging through Resets and Power Cycles

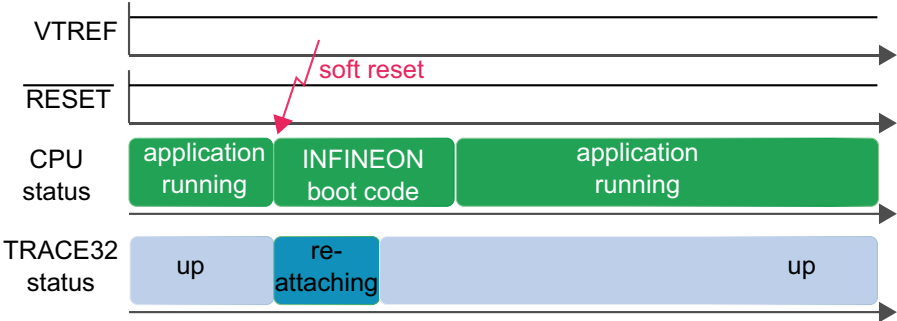
While debugging a TriCore CPU, resets and power cycles can be caused by the application software, on-chip logic or external logic, e.g., a watchdog. These events can lead to a reset of the debug logic and loss of the connection to the debugger. TRACE32 supports debugging despite such events for the following scenarios.

Soft Resets

Soft resets cover *application* and *system resets* of the TriCore. These are typically caused by the application software or on-chip logic. They do not reset the debug logic. Programmed on-chip breakpoints will be preserved. Such a reset will be automatically detected by the debugger by reading the corresponding status register.

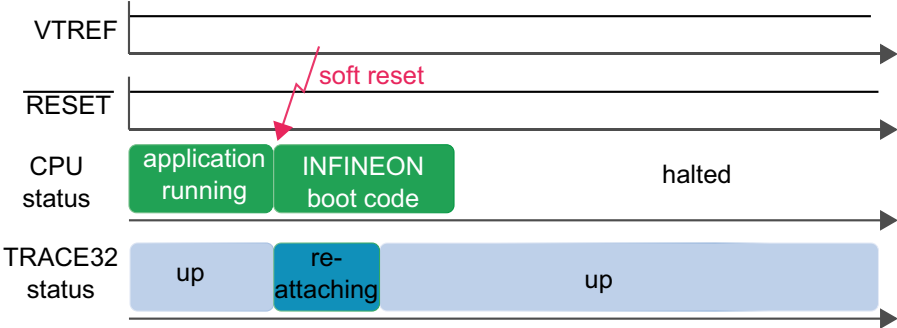
TRACE32 provides the following options to handle soft resets.

- CPU running after soft reset**



This behavior is selected by using [SYSystem.Option.RESetBehavior RestoreGo](#) or [SYSystem.Option.RESetBehavior RunRestore](#).

- CPU halted after soft reset**



This behavior is selected by using [SYSystem.Option.RESetBehavior Halt](#).

Hard Resets

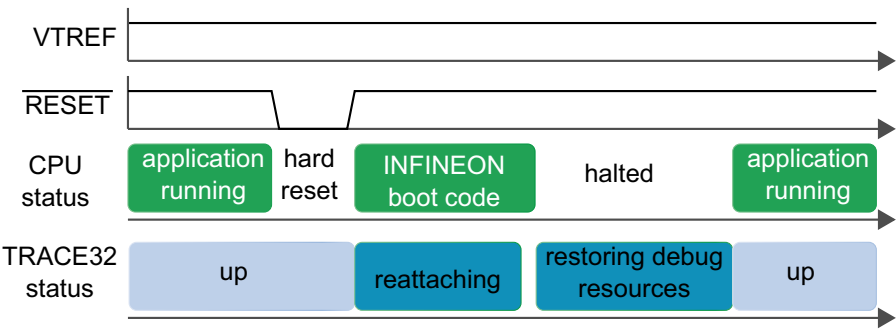
Hard resets cover *warm power-on resets*. These are typically caused by external logic pulling the `RESET` pin low. They reset the debug logic and the connection to the debugger. Programmed on-chip breakpoints will not be preserved.

Such a reset will be automatically detected by the debugger in the following ways:

- Reading status of `RESET` line. This does only work if all hard resets are visible at the `RESET` pin of the debug connector. Details can be found in [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf).
- Polling of the debug port using DAP. The debugger will detect the begin of a hard reset when the connection to the target is lost. The debugger will detect the end of a hard reset by polling the debug port until the target responds again. This requires that DAP2 is selected as type of the debug port, see [SYSTEM.CONFIG.DEBUGPORTTYPE DAP2](#).

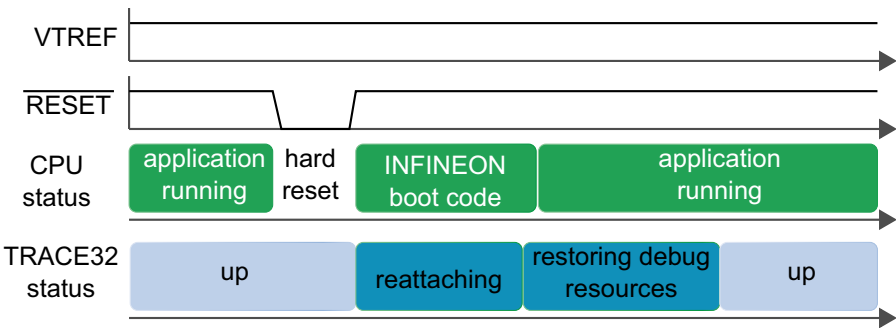
TRACE32 provides the following options to handle hard resets.

- **Restore debug resources after hard reset, start CPU**



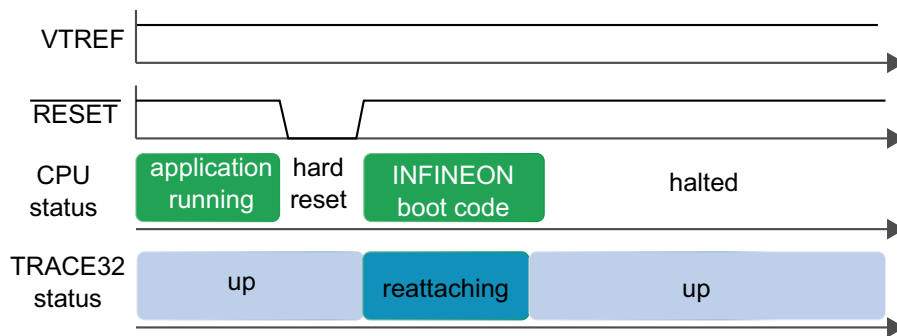
After a hard reset, TRACE32 will halt the CPU for a short time to restore the debug resources and continue execution afterwards. Breakpoints will trigger from the reset vector. This behavior is selected by using [SYSTEM.Option.RESetBehavior RestoreGo](#).

- **Restore debug resources while CPU is running after hard reset**



After a hard reset, TRACE32 will not halt the CPU. The debug resources will be restored while the CPU is running. Breakpoints triggered during this time will be missed. This behavior is selected by using [SYSTEM.Option.RESetBehavior RunRestore](#).

- **CPU halted after hard reset**



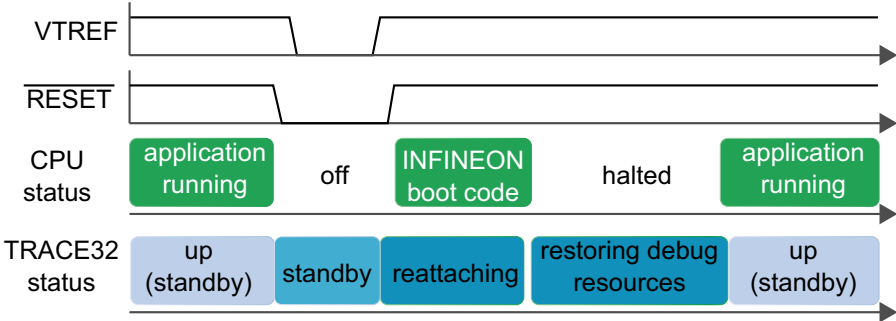
After a hard reset, TRACE32 will halt the CPU at the reset vector. This behavior is selected by using **SYstem.Option.RESetBehavior Halt**.

Power Cycles

Power cycles cover *cold power-on resets* caused by switching off CPU power completely, and then on again. They reset the debug logic and the connection to the debugger. Programmed on-chip breakpoints will not be preserved.

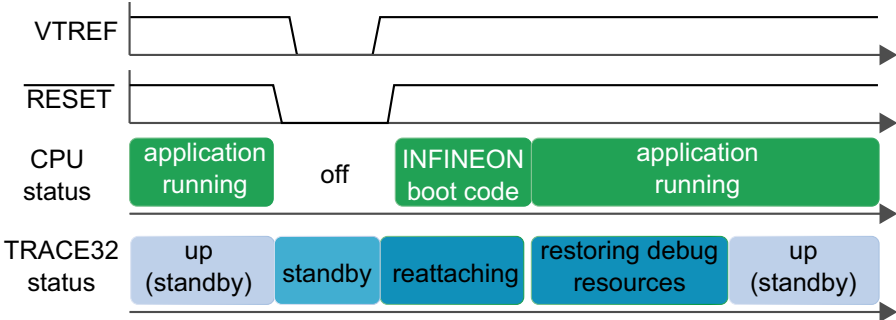
The debugger will automatically detect a power cycle by observing the VTREF pin of the debug connector. Details can be found in “[Application Note Debug Cable TriCore](#)” (app_tricore_ocds.pdf). Debugging through power cycles is supported by TRACE32 if **SYSystem.Mode StandBy** is selected beforehand. The following options are available:

- Restore debug resources after power cycle, start CPU**



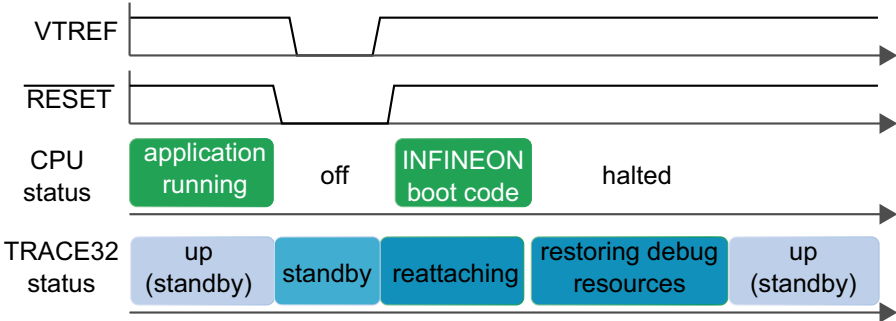
After detecting the end of a power cycle, TRACE32 will halt the CPU for a short time to restore the debug resources. Breakpoints will trigger from the reset vector. This behavior is selected by using **SYSystem.Option.RESetBehavior RestoreGo**.

- Restore debug resources while CPU is running after power cycle**



After detecting the end of a power cycle, TRACE32 will not halt the CPU. The debug resources will be restored while the CPU is running. Breakpoints triggered during this time will be missed. This behavior is selected by using **SYSystem.Option.RESetBehavior RunRestore**.

- CPU halted after power cycle**



After detecting the end of a power cycle, TRACE32 will halt the CPU at the reset vector. This behavior is selected by using **SYSystem.Option.RESetBehavior Halt**.

Suspending Peripherals

TriCore CPUs allow to suspend peripheral modules automatically when the core(s) are halted. Depending on the CPU family, dedicated mechanisms are used for watchdogs and system timers. These are described after the general mechanism.

The general mechanism consists of two components:

- A chip-wide level signal (*suspend signal*) that is generated by the debug logic of the chip. If **SYStem.Option.PERSTOP** is set, TRACE32 manages this signal according to the overall debug configuration.
- Each peripheral module has configuration bits that establish how it is suspended. For AURIX devices, this is typically managed via the OCDS control and status register (<module>_OCS), while for AUDO devices, it is done through the clock control register (<module>_CLC).

These registers can be configured by the target application or through commands in the debugger's startup script, such as **PER.Set.simple**, **PER.Set.Field**, or **PER.Set.ByName**.

Example: Suspend GTP12 on a TC3xx chip

```
SYStem.Option.PERSTOP ON
; GTP12_OCS.SUS[_P]=0x11
; Per.Set.Field D:0xF00018E8 %Long 0x1F000000 0x11
PER.Set.ByName .GPT120_OCS.SUS "Set hard suspend"
```

In case such a setting shall be automatically restored after a reset or power less, this can be achieved using **Data.STANDBY** and **Data.STARTUP**.

During an assembler step **Step.Asm**, **Go.direct /SingleCORE** and target code executed by TRACE32 (see “**Target Code Execution**”, page 57), the suspend signal will be kept active.

Components directly suspended by the MCBS (AUDO devices) or OTGS (AURIX devices) can be controlled by **TrOnchip.SusTarget**.

Suspending the System Timers of TC3xx

In TC3xx CPUs the system timers (STMs) are directly connected to the suspend out signal of the CPU. In case **SYStem.Option.PERSTOP ON** is set, TRACE32 will manage the suspend out signal for all assigned cores. The suspend of the system timer must still be configured by setting **STMx_OCS.SUS**. Example:

```
SYStem.Option.PERSTOP ON
; STM0_OCS.SUS[_P]=0x12
; Per.Set.Field D:0xF00010E8 %Long 0x1F000000 0x12
PER.Set.ByName .STM0_OCS.SUS "Set Suspended"
```

Suspending the Watchdogs

TriCore CPUs disable their internal watchdogs automatically if the debugging system is enabled. Optionally, the watchdogs can be connected to suspend signal as well. See [SYStem.Option.WDTSUS](#) for more details.

Code Overlays

Some memories, e.g., the program scratch pad RAM (PSPR), can be accessed from the CPU in a very performant way. An optimization technique is to dynamically copy different code parts into such a memory and execute them from there.

For the debugger this means, that addresses map to different symbols at different times. For a correct mapping, the TRACE32 Overlay mechanism needs to be enabled. For details, refer to See [SYStem.Option.OVERLAY](#). An example can be found in `~/demo/tricore/etc/overlay`.

Data Overlays

The Data Access Overlay (OVC) mechanism of TC2xx and newer chips allows to redirect data accesses made by a CPU to a certain memory (e.g., flash) to another memory (e.g., RAM). Effectively this means, that the address map of a CPU differs from the address map of the system bus.

By default, the debugger will not recognize this, since it accesses all memories by the system bus. This means the debugger will, e.g., display the original values of the memory instead of the values in the overlay memory.

For a correct resolution, [SYStem.Option.OVC ON](#) needs to be set. Then the debugger will check the status of the OVC module, read its configuration if necessary and resolve accesses accordingly. An example can be found in `~/demo/tricore/etc/ovc`.

Cerberus Access Protection

The Cerberus of a TriCore CPU is one of the main debug modules. Some implementations have access control registers. By default, TRACE32 will configure this register to allow access by the debugger only. This behavior can be changed using [SYStem.Option.CBSACCEN<x>](#).

When configuring these registers manually, make sure to grant access to the debugger.

Please consider also [“Parallel Usage of a 3rd-Party Tool”](#), page 47.

Target Code Execution

In certain situations TRACE32 will download and/or execute code to your target to achieve specific actions. This includes but is not limited to:

- target controlled flash programming (see [FLASH](#) command group)
- target code referenced in [Data.EPILOG.TARGET](#), [Data.PROLOG.TARGET](#), etc.
- [Var.Set](#), [Var.Assign](#) if used with a function.
- [CACHE.INVALIDATE](#), [CACHE.FLUSH](#), [CACHE.CLEAN](#).
- Memory access using access class “D” or “DC” if core access is enabled. For further details, see [“Reading and Writing using Target Code Method”](#), page 44.

Internal Break Bus (JTAG)

The JTAG connector has two break lines which can be connected to an internal Break Bus of TriCore devices of the AUDO family:

Pin	Break Bus
nBRKIN	0
nBRKOUT	1

An AUDO-TriCore chip has several modules (Break Target) which can react on break signals, such as TriCore PCP, MCDS and the Suspend Switch and several modules (Break Source) which can generate break signals, such as TriCore, PCP, MCDS, DMA, Peripheral Buses and MLI bridges.

The Break Buses can be used to distribute break signals from a Break Source to one or more Break Targets. For example TriCore can be stopped concurrently when PCP breaks and vice versa.

NOTE:	Break signals are edge signals and only active for a short period of time.
--------------	--

In this section:

- [SYStem.Up Errors](#)
- [Debugging Optimized Code](#)
- Please also see the chapter [“FAQ”](#), page 59.

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 or the debug connector is mounted in wrong orientation.
- External controlled /RESET line:
The debugger controls the processor reset and uses the /RESET line to reset the CPU on most of the **SYStem.Mode** commands. Therefore only non-active, open-drain driver may be connected to this signal.
- There is logic added to the JTAG signals:
The debugger supports multi-processor debugging only in certain cases. When inserting other processors in the JTAG chain, TriCore must be the first processor. No other TriCore and no processor with a Cerberus based JTAG interface (e.g. XC16x, XC2000) are allowed in the chain.
- There are additional loads or capacities on the JTAG line.

For more information on pins, signals and daisy-chaining please refer to [“Debug Interface Configuration”](#) in Application Note Debug Cable TriCore, page 37 (app_tricore_ocds.pdf).

Debugging Optimized Code

It is recommended to debug applications with compiler-optimizations turned off. However, this is not always possible.

Optimized code often splits a single HLL line into multiple blocks of non-consecutive assembler instructions. Depending on the compiler a single HLL line can generate multiple blocks of code, making single stepping fiddly and setting breakpoints complex.

Such lines can be summarized when loading an application with the **Data.LOAD** command:

- The **/SingleLineAdjacent** option summarizes adjacent blocks of assembler code generated for an HLL line.
- The **/SingleLine** option summarizes blocks of assembler code generated for an HLL line.

For example:

```
Data.LOAD.Elf my_application.elf /SingleLineAdjacent
```

These options are explained in detail in chapter “[Details on Generic Load Options](#)” in General Commands Reference Guide D, page 86 (general_ref_d.pdf).

FAQ

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

Tracing allows an in-depth analysis of the behavior and the timing characteristics of the embedded system.

A trace records information about a program's execution during runtime, usually including program flow and read/written data. The cores as well as chip internal buses can act as trace sources.

This chapter covers the following topics:

- [“On-chip Trace \(OCDS-L3\)”](#), page 60
- [“Serial Off-Chip Trace”](#), page 69
- [“Parallel Off-Chip Trace”](#), page 69

Only basic information will be given here, details and advanced tasks are described in separate documents, see [“Further Reading”](#), page 68.

The trace messages for the On-chip Trace and the Serial Off-chip Trace are generated by the MCDS or Mini-MCDS. The MCDS is only implemented in Emulation Devices, the Mini-MCDS in some Product Devices. Please refer to [“MCDS User's Guide”](#) (mcds_user.pdf) for more information.

On-chip Trace (OCDS-L3)

On-chip tracing is only possible with an Infineon Emulation Device (ED), offering the MCDS (MultiCore Debug Solution) for implementing trace triggers, filters and generation of trace messages (MCDS messages).

Use [Trace.METHOD Onchip](#) for selecting the on-chip trace.

Quick Start for Tracing with On-chip Trace (OCDS-L3)

It is assumed that you are tracing a TC1766ED B-Step on an Infineon TriBoard-TC1766.300 or above.

1. Prepare the Debugger

Load your application and prepare for debug. For details, see [“Single-Core Debugging - Quick Start”](#), page 17.

2. Specify Trace Source and Recording Options

Select the core to trace (e.g. TriCore), and what should be recorded (e.g. program flow and timestamps). When enabling timestamps, the CPU clock has to be added also.

```
MCDS.SOURCE TriCore FlowTrace ON           ; enable TriCore program flow
                                           ; trace

MCDS.TimeStamp TICKS                        ; enable Ticks as timestamps

MCDS.CLOCK SYStem 20.0MHz                   ; configure CPU clock for correct
                                           ; timestamp evaluation
```

3. Start and Stop Tracing

```
Go                                           ; start tracing

Break                                       ; stop tracing
```

Note that tracing can also be stopped by a breakpoint.

4. View the Results

```
Onchip.List                               ; view recorded trace data
```

Supported Features

- Program Flow Trace for TriCore and PCP
- Data Trace for TriCore and PCP
- Ownership Trace for PCP
- FPI Bus Trace (independent of TriCore or PCP), both System Peripheral Bus (SPB) and Remote Peripheral Bus (RPB, if available)
- Timestamps
- **Simple Trace Control**

See the **Onchip.Mode** commands for a general setup of the on-chip trace, the **MCDS** commands and **“Basic Trace Usage”** in MCDS User’s Guide, page 24 (mcds_user.pdf) for a detailed setup of the on-chip MCDS resources.

NOTE:	A trace source can either be TriCore, PCP or both at the same time. It is not possible to enable a trace stream (e.g. Program Flow or Data Trace) for only one trace source when both are enabled.
-------	--

Trace Control

The On-chip settings can be done with the **Onchip** commands, e.g. from the **Onchip.state** window. The settings affect both TriCore and PCP trace.

NOTE:	Onchip.AutoArm has an effect only on TriCore, but not on PCP. Always make sure that PCP is running when arming the trace.
--------------	--

Trace Evaluation

In case one or more of the FBI buses have been traced, the **Onchip.List** window features two additional columns with information on the bus access:

BusMaster	Shows which Bus Master initiated the FPI bus access, e.g. DMA, PCP, ...
BusMODE	Shows in which mode the FPI Bus access was performed: User or Supervisor mode.

Bus accesses performed in User Mode are displayed in light gray.

See **<trace>.List** in “**General Commands Reference Guide T**” (general_ref_t.pdf) on how to customize the **Onchip.List** window to your needs.

Impact of the Debugger on FPI Bus Tracing

The debugger is either connected as Bus Master via the System Peripheral Bus (AUDO-NG and previous) or via the DMA controller (all other AUDO and AURIX).

All accesses performed by the debugger are visible on the buses and will be traced which consumes a large amount of on-chip trace memory. Enabling **SYStem.Option.DUALPORT** will increase the amount of traced bus data depending on the size and number of displayed data windows.

By default, accesses generated by the debugger are not displayed. To display those, use **Onchip.Mode.SLAVE OFF.Tri**

Trace Control Using Break.Set or Var.Break.Set

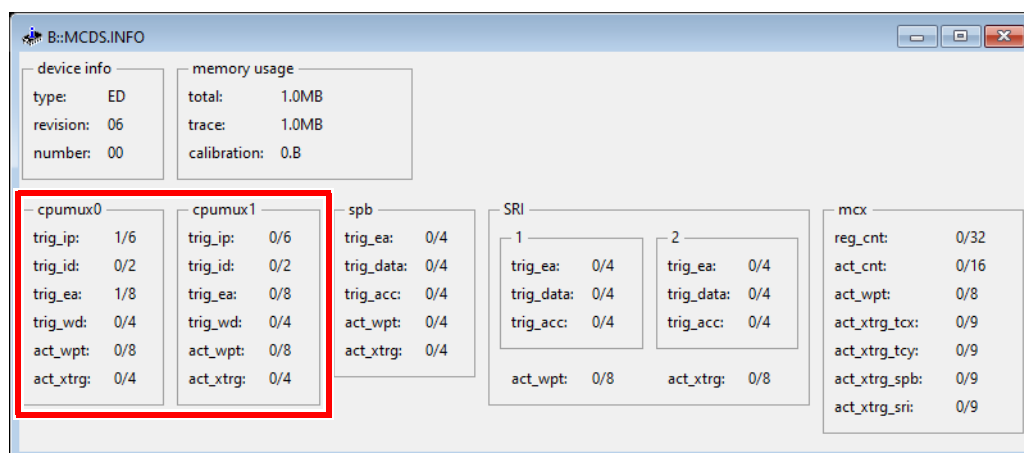
The **Break.Set** or **Var.Break.Set** commands provide basic trace control. For more complex trace scenarios please refer to “**Application Note for Complex Trigger Language**” (app_ctl.pdf).

A successfully loaded target application is needed for the following examples.

TriCore AURIX

With TRACE32 Release 09/2022, programming for trace control has been made easier for all TriCore AURIX MCUs. For the examples it is assumed that the user knows on which core a function is running or by which core a variable access is performed. This avoids the following problem: many AURIX processors have n cores, but **MCDS** only allows trace generation for r cores ($n \geq r$) only.

The **MCDS.INFO** window shows for how many cores **MCDS** can generate trace information. The number of **cpumux** is the characteristic value here. The screenshot below shows the MCDS.INFO window for a TriCore AURIX TC275TF, which has 3 core, but only 2 cpumux.



Example 1: Trace only the instructions of the sieve() function

The sieve() function executes on core 0. Subfunctions, interrupts etc. are not included.

```
; enable MCDS clock
CLOCK.ON
; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; instruct MCDS to generate program trace for the instruction within the
; sieve() function
Var.Break.Set sieve /Program /TraceEnable /Core 0.

; display the result
Trace.List
```

Example 2: Trace all instructions executed after the function sieve() was started until its exit

The function sieve executes on core 0. Subfunctions, interrupts etc. are included.

```
; enable MCDS clock
CLOCK.ON
; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; instruct MCDS to start the generation of program trace when the
; function sieve is entered
Break.Set sieve /TraceON /Core 0.

; instruct MCDS to stop the generation of program trace when the
; function sieve is exited
Break.Set sYmbol.EXIT(sieve) /Program /TraceOFF /Core 0.

; display the result
Trace.List
```

Example 3: Stop the trace recording at the exit of the function sieve()

The function sieve executes on core 0.

```
; enable MCDS clock
CLOCK.ON
; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; stop trace recording when the exit of function sieve() is reached
Break.Set sYmbol.EXIT(sieve) /Program /TraceTrigger /Core 0.

; display the result
Trace.List
```

Example 4: Trace write accesses to a variable

The write access to flags[12] is done by core 1 or core 2.

```
; enable MCDS clock
CLOCK.ON
; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; instruct MCDS to generate a data trace when a write access to
; the variable flags[12] occurs by core 1 or core 2.
Var.Break.Set flags[12] /Write /TraceEnable /Core 1.
Var.Break.Set flags[12] /Write /TraceEnable /Core 2.

; display the result
Trace.List
```


Example 5: Trace Specific write accesses to a variable

The data value 0. is written to flags[12] by core 0.

```
; enable MCDS clock
CLOCK.ON
; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; instruct MCDS to generate a data trace when a write access to the
variable flags[12] with the data value 0. occurs from core 0.
Var.Break.Set flags[12] /Write /Data.auto 0. /TraceEnable /Core 0.

; display the result
Trace.List
```

Aged MCUs like the AUDO Max family have only a single TriCore core, which is capable of a program and data trace (write accesses only) for the emulation devices. For this generation of MCUs the classic command syntax is still valid.

Example 1: Trace only the instructions of the sieve() function

Subfunctions, interrupts etc. are not included.

```
; display MCDS state window
MCDS.state

; enable MCDS clock
CLOCK.ON

; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; instruct MCDS to generate program trace for the instruction within the
; sieve() function
Var.Break.Set sieve /Program /TraceEnable
; TriCore program trace needs to be enabled
MCDS.SOURCE.Set TriCore.Program ON

; display the result
Trace.List
```

Example 2: Trace all instructions executed after the function sieve() was started until its exit

Subfunctions, interrupts etc. are included.

```
; display MCDS state window
MCDS.state

; enable MCDS clock
CLOCK.ON

; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; advise MCDS to start the generation of program trace when the
; function sieve is entered
Break.Set sieve /Program /TraceON

; advise MCDS to stop the generation of program trace when the
; function sieve is exited
Break.Set sYmbol.EXIT(sieve) /Program /TraceOFF

; TriCore program trace needs to be enabled
MCDS.SOURCE.Set TriCore.Program ON

; display the result
Trace.List
```

Example 3: Stop the trace recording at the exit of the function sieve()

```
; display MCDS state window
MCDS.state

; enable MCDS clock
CLOCK.ON

; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; stop trace recording when the exit of function sieve() is reached
Break.Set sYmbol.EXIT(sieve) /Program /TraceTrigger

; TriCore program trace needs to be enabled
MCDS.SOURCE.Set TriCore.Program ON

; display the result
Trace.List
```

Example 4: Trace write accesses to a variable

```
; display MCDS state window
MCDS.state

; enable MCDS clock
CLOCK.ON

; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

; advise MCDS to generate a data trace message when a write access to
; the variable flags[12] occurs
Var.Break.Set flags[12] /Write /Onchip /TraceEnable

; TriCore data trace for write accesses needs to be enabled
MCDS.SOURCE.Set TriCore.WriteAddr ON
MCDS.SOURCE.Set TriCore.WriteData ON

; TriCore program trace can be disabled
MCDS.SOURCE.Set TriCore.Program OFF

; show trace list window
Trace.List
```

Aged TriCore microcontrollers

```
; display MCDS state window
MCDS.state

; enable MCDS clock
CLOCK.ON

; instruct MCDS to generate trace timestamps
MCDS.TimeStamp.ON

Var.Break.Set flags[12] /Write /Data.auto 0x0 /TraceEnable /Core 0.

; TriCore data trace for write accesses needs to be enabled
MCDS.SOURCE.Set TriCore.WriteAddr ON
MCDS.SOURCE.Set TriCore.WriteData ON

; TriCore program trace can be disabled
MCDS.SOURCE.Set TriCore.Program OFF

; show trace list window
Trace.List
```

Further Reading

For advanced tasks regarding the MCDS, see [“MCDS User’s Guide”](#) (mcds_user.pdf).

Serial Off-Chip Trace

The Aurora Giga-Bit Trace (AGBT) is a high-speed serial trace for AURIX devices.

For advanced tasks, see the [“MCDS User’s Guide”](#) (mcds_user.pdf).

Parallel Off-Chip Trace

For more information, see chapter [“Parallel Off-chip Trace - OCDS-L2 Flow Trace \(Analyzer\)”](#), page 134.

CPU specific BMC Commands

The **BMC** (**BenchMark Counter**) commands provide control and usage of the on-chip benchmark and performance counters on the chip if available.

For information about *architecture-independent* **BMC** commands, refer to “**BMC**” in General Commands Reference Guide B, page 9 (general_ref_b.pdf).

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

BMC.SELect

Select counter for statistic analysis

Format:

BMC.SELect <counter>

<counter>:

PMN0 | PMN1 | ...

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

BMC.<counter>.ATOB

Control A-to-B mode

Format:

BMC.<counter>.ATOB [ON | OFF]

Enable or disable A-to-B mode of the counter. In A-to-B mode only events between the alpha and beta marker are counted.

The Alpha and Beta markers may be of type Program, Read, Write and ReadWrite. For details about setting the Alpha and Beta markers see [Break.Set](#).

BMC.<counter>.TRIGMODE

BMC trigger mode

Format:

BMC.<counter>.TRIGMODE [OFF | TRACEOVERFLOW]

Enables/disables the generation of BenchMark Counter trace messages.

An example can be found under “[Example: Record BMC Counters in the Trace](#)” in MCDS User’s Guide, page 43 (mcds_user.pdf).

BMC.<counter>.TRIGVAL

BMC trigger value

Format: BMC.<counter>.TRIGVAL [<value>]
--

Defines the BenchMark Counter trigger value. Entry in the trace when counter has reached trigger value and is then reset.

An example can be found under “[Example: Record BMC Counters in the Trace](#)” in MCDS User’s Guide, page 43 (mcds_user.pdf).

CPU specific SYStem.CONFIG Commands

The **SYStem.CONFIG** commands are used to configure the behavior of the complete target system for debugging, e.g., the debug interface or the chaining of several CPUs.

SYStem.CONFIG.state

Display target configuration

Format:

SYStem.CONFIG.state [/<tab>]

<tab>:

DebugPort | Jtag | XCP

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: DebugPort , JTAG , and XCP .
DebugPort	Lets you configure the electrical properties of the debug connection, such as the communication protocol or the used pinout.
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.
XCP	Lets you configure the XCP connection to your target. For descriptions of the commands on the XCP tab, see “ XCP Debug Back-End ” (backend_xcp.pdf).

Format:	SYSystem.CONFIG <parameter> <number_or_address> SYSystem.MultiCore <parameter> <number_or_address> (deprecated)
<parameter>:	CORE <core>
<parameter>: (JTAG):	DRPRE <bits> DRPOST <bits> IRPRE <bits> IRPOST <bits> TAPState <state> TCKLevel <level> TriState [ON OFF] Slave [ON OFF]

The four parameters IRPRE, IRPOST, DRPRE, DRPOST are required to inform the debugger about the TAP controller position in the JTAG chain, if there is more than one core in the JTAG chain (e.g. Arm + DSP). The information is required before the debugger can be activated e.g. by a **SYSystem.Up**. See **Daisy-chain Example**.

For some CPU selections (**SYSystem.CPU**) the above setting might be automatically included, since the required system configuration of these CPUs is known.

TriState has to be used if several debuggers (“via separate cables”) are connected to a common JTAG port at the same time in order to ensure that always only one debugger drives the signal lines. 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.

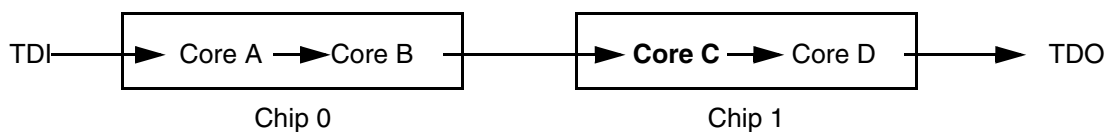


Multicore debugging is not supported for the DEBUG INTERFACE (LA-7701).

CORE	For multicore debugging one TRACE32 PowerView GUI has to be started per core. To bundle several cores in one processor as required by the system this command has to be used to define core and processor coordinates within the system topology. Further information can be found in SYSystem.CONFIG.CORE .
DRPRE	(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	(default: 0) <i><number></i> 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	(default: 0) <i><number></i> 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	(default: 0) <i><number></i> 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.
TAPState	(default: 7 = Select-DR-Scan) This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.
TCKLevel	(default: 0) Level of TCK signal when all debuggers are tristated.
TriState	(default: OFF) If several debuggers share the same debug port, this option is required. The debugger switches to tristate mode after each debug port access. Then other debuggers can access the port. JTAG: This option must be used, if the JTAG line of multiple debug boxes are connected by a JTAG joiner adapter to access a single JTAG chain.
Slave	(default: OFF) If more than one debugger share the same debug port, all except one must have this option active. JTAG: Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET).

Daisy-Chain Example



Below, configuration for core C.

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 0. 1. ; Target Core C is Core 0 in Chip 1
```

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

Format:	SYStem.CONFIG.CORE <core_index> <chip_index> SYStem.MultiCore.CORE <core_index> <chip_index> (deprecated)
<chip_index>:	1 ... i
<core_index>:	1 ... k

Default *core_index*: depends on the CPU, usually 1. for generic chips

Default *chip_index*: derived from CORE= parameter of the configuration file (config.t32). The CORE parameter is defined according to the start order of the GUI in T32Start with ascending values.

To provide proper interaction between different parts of the debugger, the systems topology must be mapped to the debugger's topology model. The debugger model abstracts chips and sub cores of these chips. Every GUI must be connect to one unused core entry in the debugger topology model. Once the **SYStem.CPU** is selected, a generic chip or non-generic chip is created at the default *chip_index*.

Non-generic Chips

Non-generic chips have a fixed number of sub cores, each with a fixed CPU type.

Initially, all GUIs are configured with different *chip_index* values. Therefore, you have to assign the *core_index* and the *chip_index* for every core. Usually, the debugger does not need further information to access cores in non-generic chips, once the setup is correct.

Generic Chips

Generic chips can accommodate an arbitrary amount of sub-cores. The debugger still needs information how to connect to the individual cores e.g. by setting the JTAG chain coordinates.

Start-up Process

The debug system must not have an invalid state where a GUI is connected to a wrong core type of a non-generic chip, two GUIs are connected to the same coordinate or a GUI is not connected to a core. The initial state of the system is valid since every new GUI uses a new *chip_index* according to its CORE= parameter of the configuration file (config.t32). If the system contains fewer chips than initially assumed, the chips must be merged by calling **SYStem.CONFIG.CORE**.

Format:

SYStem.CONFIG.BreakPIN [PortPort | TdiPort | PortTdo | TdiTdo]

Default: PortPort.

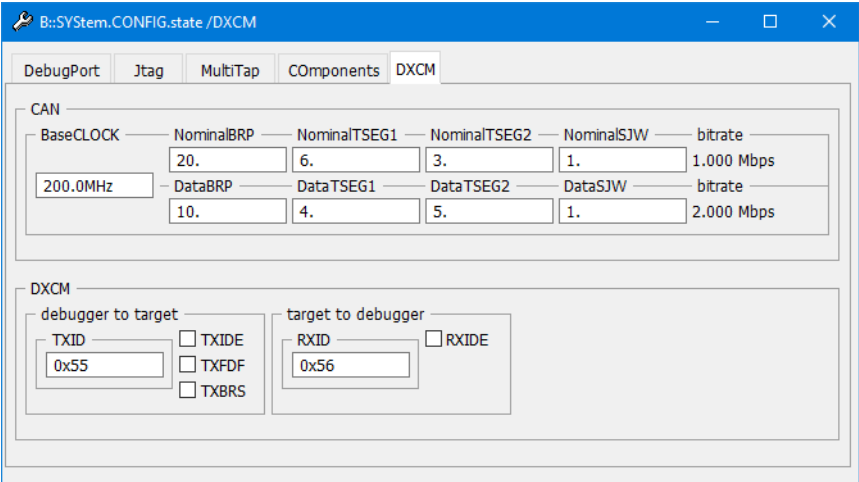
Maps a **Break Bus** to either a GPIO port pin or an unused JTAG pin. It is dependent on the **Interface Mode** which **Break Bus** can be mapped to which pin:

	Break Bus 0	Break Bus 1
PortPort	GPIO port pin	GPIO port pin
TdiPort	TDI pin	GPIO port pin
PortTDO	GPIO port pin	TDO pin
TdiTdo	TDI pin	TDO pin

The command group **SYSystem.CONFIG.CAN** configures the timing properties for debugging over a Controller Area Network (CAN) interface. Currently, the only supported protocol that uses the CAN interface is DxCM (DAP over CAN messages).

These timing parameters must be chosen to be compatible with the bus. Refer to ISO 11898 for details about these parameters.

These settings can also be viewed and changed using the command **SYSystem.CONFIG.state /DXCM**.



Use the command group **SYSystem.CONFIG.DXCM** to set parameters specific to the DxCM protocol.

SYSystem.CONFIG.CAN.BaseCLOCK

Base clock for CAN interface

Format:

SYSystem.CONFIG.CAN.BaseCLOCK <frequency>

Configures the base clock used for the CAN interface. The <frequency> can be in the range of 10 MHz to 200 MHz. For optimum performance, it is suggested to prefer higher frequencies.

SYStem.CONFIG.CAN.NominalBRP

Set CAN nominal baud rate prescaler

Format:	SYStem.CONFIG.CAN.NominalBRP <i><prescaler></i>
<i><prescaler></i> :	1. ... 4095.

Configures the baud rate prescaler used for nominal transmission.

The effective bit rate during nominal transmission can be computed as $\text{BaseCLOCK} / \text{NominalBRP} / (1 + \text{NominalTSEG1} + \text{NominalTSEG2})$.

SYStem.CONFIG.CAN.NominalTSEG1

Set CAN nominal Phase_seg1

Format:	SYStem.CONFIG.CAN.NominalTSEG1 <i><length></i>
<i><length></i> :	1. ... 4095.

Configures the length of Phase_Seg1 for nominal transmission.

SYStem.CONFIG.CAN.NominalTSEG2

Set CAN nominal Phase_seg2

Format:	SYStem.CONFIG.CAN.NominalTSEG2 <i><length></i>
<i><length></i> :	1. ... 4095.

Configures the length of Phase_Seg2 for nominal transmission.

SYStem.CONFIG.CAN.NominalSJW

Set CAN nominal SJW parameter

Format:	SYStem.CONFIG.CAN.NominalSJW <i><length></i>
<i><length></i> :	1. ... 4095.

Configures the Synchronization Jump Width for nominal transmission.

Format: **SYStem.CONFIG.CAN.DataBRP** *<length>*

<length>: **1. ... 4095.**

Configures the baud rate prescaler used for data transmission.

This setting is only relevant for CAN-FD (ISO 11898-1:2015) transmission with baud rate switching. Note that this needs to be set correctly if the bus used baud rate switching, even if the frames for debugging do not.

The effective bit rate during data transmission can be computed as $\text{BaseCLOCK} / \text{DataBRP} / (1 + \text{DataTSEG1} + \text{DataTSEG2})$.

SYStem.CONFIG.CAN.DataTSEG1

Set CAN data Phase_seg1

Format: **SYStem.CONFIG.CAN.DataTSEG1** *<length>*

<length>: **1. ... 4095.**

Configures the length of Phase_Seg1 for data transmission.

This setting is only relevant for CAN-FD (ISO 11898-1:2015) transmission with baud rate switching. Note that this needs to be set correctly if the bus used baud rate switching, even if the frames for debugging do not.

SYStem.CONFIG.CAN.DataTSEG2

Set CAN data Phase_seg2

Format: **SYStem.CONFIG.CAN.DataTSEG2** *<length>*

<length>: **1. ... 4095.**

Configures the length of Phase_Seg2 for data transmission.

This setting is only relevant for CAN-FD (ISO 11898-1:2015) transmission with baud rate switching. Note that this needs to be set correctly if the bus used baud rate switching, even if the frames for debugging do not.

Format: **SYStem.CONFIG.CAN.DataSJW** *<length>*

<length>: **1. ... 4095.**

Configures the Synchronization Jump Width for data transmission.

This setting is only relevant for CAN-FD (ISO 11898-1:2015) transmission with baud rate switching. Note that this needs to be set correctly if the bus used baud rate switching, even if the frames for debugging do not.

The **SYStem.CONFIG.DAP** commands are used to configure the debugger connection to the target CPU via the DAP interface mode. Before these commands can be used, a DAP interface mode has to be selected by using the **SYStem.CONFIG.DEBUGPORTTYPE** command.

For details, see “**Application Note Debug Cable TriCore**” (app_tricore_ocds.pdf).

SYStem.CONFIG.DAP.BreakPIN

Define mapping of break pins

Format:

SYStem.CONFIG.DAP.BreakPIN [PortPort | TdiPort | PortTdo | TdiTdo]

Default: PortPort.

This command is an alias to **SYStem.CONFIG BreakPIN** for backward compatibility reasons.

SYStem.CONFIG.DAP.CRC6

Enable CRC6 mode

Format:

SYStem.CONFIG.DAP.CRC6 [ON | OFF]

Default: OFF.

Enables the CRC6 mode of the DAP connection. This setting should only be changed if a 3rd-party tool using CRC6 mode is used in parallel at the same debug port (see also “**Parallel Usage of a 3rd-Party Tool**”, page 47). This requires an AURIX CPU. For further details about this mode, please refer to the Infineon User Manual.

SYStem.CONFIG.DAP.DAPENable

Enable DAP mode on PORST

Format:

SYStem.CONFIG.DAP.DAPEN [TARGET | ON | OFF]

Default: TARGET.

Defines if the DAP interface of the CPU is enabled during a Power On Reset (PORST). This command requires that the debugger DAP interface is enabled by **SYStem.CONFIG.DEBUGPORTTYPE** before.

For target boards where a pull-up resistor on \overline{TRST} line permanently enables the DAP interface the TARGET setting is required.

In case the CPU DAP interface should not be enabled although a debugger is attached to the target board, the OFF setting is recommended. When performing a **SYStem.Mode Go** or **SYStem.Mode Up**, the debugger enables the CPU DAP interface automatically when performing the PORST. **SYStem.Mode Attach** is not possible in this case.

If the CPU DAP interface should be enabled as long as the debugger is attached, the ON setting is required. All **SYStem.Mode** options are possible in this case, including hot attach.

See the “**Application Note Debug Cable TriCore**” (app_tricore_ocds.pdf) for details.

NOTE:	This command only has an effect in case the <code>TRST</code> line is connected to the <code>DAPEN</code> output pin. The <code>ALTEN</code> pin always behaves like the OFF setting and should only be used in case DAP or JTAG mode should be possible via the 16-pin connector.
--------------	---

SYStem.CONFIG.DAP.SISP

Configure SISP setting

[build 134017 - DVD 09/2021]

Format:	SYStem.CONFIG.DAP.SISP [AUTO 0 1 2 3]
---------	---

Default: AUTO.

Configures the SISP (Startbit insetivity period) setting used for the DAP connection. This setting should only be changed if a 3rd-party tool using a certain SISP value is used in parallel at the same debug port (see also “**Parallel Usage of a 3rd-Party Tool**”, page 47). This requires an AURIX CPU. For further details about this setting, please refer to the Infineon User Manual.

SYStem.CONFIG.DAP.USERn

Configure and set USER pins

Format:	SYStem.CONFIG.DAP.USER0 [In Out Set <i><level></i>] SYStem.CONFIG.DAP.USER1 [In Out Set <i><level></i>]
<i><level></i> :	Low High

- Default for **USER0**: In.
- Default for **USER1**: Out and Low.

Configures the `USER0` and `USER1` pins of the 10 pin DAP Debug Connector as input or output. The output level can be Low or High.

Use the functions **DAP.USER0()** and **DAP.USER1()** for reading the current status.

The availability of the USER pins depends on the Debug Cable, the selected interface mode and the DAP Enabling Mode. See the **“Application Note Debug Cable TriCore”** (app_tricore_ocds.pdf) for details.

Format:	SYStem.CONFIG.DEBUGPORT <port>
<port>:	DebugCable0 DebugCableA InfineonDAS0 XCP0 Unknown

Selects the interface to the target. The available options depend on whether TRACE32 uses a hardware debugger or runs in HostMCI mode (without TRACE32 hardware).

With TRACE32 hardware

DebugCable0	Uses the debug cable directly connected to a PowerDebug hardware module.
DebugCableA	Uses the whisker connected to a CombiProbe.

HostMCI mode

InfineonDAS0	Selects the Infineon DAS backend as interface. For a detailed description and examples, see “Debugging via Infineon DAS Server” (backend_das.pdf).
XCP0	Selects the XCP backend as interface. For a detailed description and examples, see “XCP Debug Back-End” (backend_xcp.pdf).
Unknown	No backend is selected. Debugging is not possible.

Format:	SYStem.CONFIG.DEBUGPORTTYPE <type> SYStem.CONFIG.Interface [JTAG DAP2] (deprecated)
<type>:	JTAG DAP2 DAP3 DAPWide DAP4 DXCPL DXCM

Default: DAP2 (JTAG).

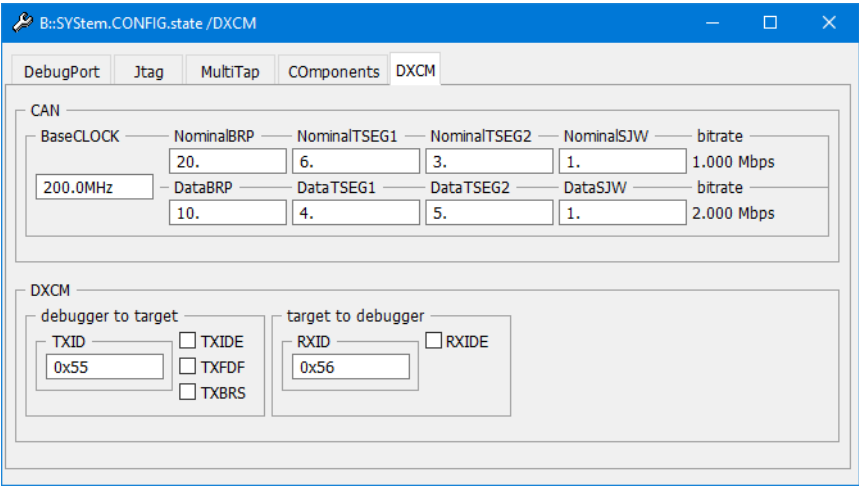
Configures the debug port type to be used by the debugger. Attempts to configure the target device to use the specified debug port type if possible. Both target device and Debug Cable must support this mode, see [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf) for details.

As of build 92378 - DVD 02/2018, the default is **DAP2**, unless the Debug Cable or CPU do not support this mode.

DAP3: Supported for AURIX only.

The command group **SYStem.CONFIG.DXCM** configures the protocol parameters for the DxCM (DAP over CAN messages).

These settings can also be viewed and changed using the command **SYStem.CONFIG.state /DXCM**.



Use the command group **SYStem.CONFIG.CAN** to set general CAN timing parameters.

SYStem.CONFIG.DXCM.TXID

Control frame message ID

Format:

SYStem.CONFIG.DXCM.TXID <id>

<id>:

0x000 ... 0x7FF (base frame format)
0x00000000 ... 0x1FFFFFFF (extended frame format)

Sets the message ID to be used for frames sent to the device.

SYStem.CONFIG.DXCM.TXIDE

Control frame format

Format:

SYStem.CONFIG.DXCM.TXIDE [ON | OFF]

Controls whether frames sent to the device will use the base frame format (setting OFF) or the extended frame format (setting ON). This corresponds to the value of the IDE (Identifier Extension) field of the frames.

Format:	SYStem.CONFIG.DXCM.TXFDF [ON OFF]
---------	--

Controls whether frames sent to the device will use the ISO 11898-1:2015 FD format (setting ON) or the classic format (setting OFF).

Setting will also cause the device to reply using FD frames.

SYStem.CONFIG.DXCM.TXBRS

Control the use of baud rate switching

Format:	SYStem.CONFIG.DXCM.TXBRS [ON OFF]
---------	--

Controls whether frames sent to the device will use baud rate switching. This only has an effect if **SYStem.CONFIG.DXCM.TXFDF** is also set.

Setting will also cause the device to reply with baud rate switching.

SYStem.CONFIG.DXCM.RXID

Set ID for frames from target

Format:	SYStem.CONFIG.DXCM.RXID <id>
<id>:	0x000 ... 0x7FF (base frame format) 0x00000000 ... 0x1FFFFFFF (extended frame format)

Sets the message ID of the debugger used to listen to reply telegrams from the device.

SYStem.CONFIG.DXCM.RXIDE

Expect extended frames from target

Format:	SYStem.CONFIG.DXCM.RXIDE [ON OFF]
---------	--

Configures whether the debugger expects frames in the base frame format (setting OFF) or the extended frame format (setting ON).

The abbreviation DXCPL stands for DAP over CAN Physical Layer by Infineon.

SYStem.CONFIG.DXCPL.Timing

Configure SPD timing for DXCPL

Format:	SYStem.CONFIG.DXCPL.Timing [AUTO <time>]
---------	--

Default: AUTO.

Configures the edge distance for a '0' bit.

- For AURIX device about 0.80µs recommended.
- For AUDO MAX device it depends on the crystal which is used for the clock. 0.50µs for a 20MHz crystal and 1.00µs for a 10MHz crystal.

AUTO	The option AUTO checks the working timing settings and prints it in the AREA.view window. AUTO should only used with SYStem.Mode Attach .
------	---

SYStem.CONFIG.EXTWDTDIS

Control external watchdog

Format:	SYStem.CONFIG.EXTWDTDIS <option>
<option>:	OFF High Low HighwhenStopped LowwhenStopped SLAVE

Default for Automotive/Automotive PRO Debug Cable: High.
Default for XCP: SLAVE.

Controls the WDTDIS pin of the debug port. This configuration is only available for tools with an Automotive Connector (e.g., Automotive Debug Cable, Automotive PRO Debug Cable) and XCP.

OFF	The WDTDIS pin is not driven. (XCP only)
High	The WDTDIS pin is permanently driven high.

Low	The WDTDIS pin is permanently driven low.
HighwhenStopped	The WDTDIS pin is driven high when program is stopped (not XCP).
LowwhenStopped	The WDTDIS pin is driven low when program is stopped (not XCP).
SLAVE	The WDTDIS state of the XCP slave is not changed. (XCP only)

See also [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf).

SYStem.CONFIG.PortSHaRing Control sharing of debug port with other tool

Format: **SYStem.CONFIG.PortSHaRing [ON | OFF | Auto]**
 SYStem.Option.ETK [ON | AUTO | OFF] (deprecated)

Configure if the debug port is shared with another tool, e.g., an ETAS ETK.

ON	Request for access to the debug port and wait until the access is granted before communicating with the target.
OFF	Communicate with the target without sending requests.
Auto	Automatically detect a connected tool on next SYStem.Mode Up , SYStem.Mode Attach or SYStem.Mode Go . If a tool is detected switch to mode ON else switch to mode OFF .

The current setting can be obtained by the **PORTSHARING()** function, immediate detection can be performed using **SYStem.DETECT.PortSHaRing**.

See also [“Parallel Usage of a 3rd-Party Tool”](#), page 47.

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

Selects the processor type. The default is TC1797.

<cpu>

For a list of supported CPUs, use the command `SYStem.CPU *` or refer to the [chip search](#) on the Lauterbach website.

NOTE:

In case your device is listed on the website but not listed in the `SYStem.CPU *` list, you may require a software update. Please contact the [Lauterbach support](#) in this case.

Format:	SYStem.JtagClock <rate> SYStem.BdmClock (deprecated)
<rate>:	10000. ... 500000000.

Default: 10 MHz.

Selects the frequency for the JTAG clock. This influences the speed of data transmission between target and debugger.

Not all values in between the frequency range can be generated by the debugger. The debugger will select and display the possible value if it cannot generate the exact value.
It is also possible to enter units, e.g. 10.0 MHz.

SYStem.JtagClock EXT is not supported by TriCore.

NOTE:	The JTAG clock must be lower or equal to the CPU clock. Otherwise JTAG communication will fail. The possible maximum of the JTAG clock is extremely dependent of the target board layout.
--------------	--

Format:	SYStem.LOCK [ON OFF]
---------	-------------------------------

Default: OFF.

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

It must be ensured that the state of the TriCore 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.



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

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

Default: CPU.

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

The run-time memory access has to be activated for each window by using the memory class **E**: (e.g. **Data.dump ED:0xA1000000**) or by using the format option **%E** (e.g. **Var.View %E var1**). It is also possible to enable non-intrusive memory access for all memory areas displayed by setting **SYStem.Option.DUALPORT ON**.

Enable CPU (deprecated)	The debugger performs non-intrusive memory accesses via the CPU internal buses (FPI Bus).
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.
Denied	Non-intrusive memory access is disabled while the CPU is executing code.

Format:	SYStem.Mode <mode>
	SYStem.Attach (alias for SYStem.Mode Attach) SYStem.Down (alias for SYStem.Mode Down) SYStem.Up (alias for SYStem.Mode Up)
<mode>:	Down NoDebug Prepare Go Attach Up StandBy

Initial mode: Down.

Down	Default state and state after fatal errors. The behavior can be configured with SYStem.Option.DOWNMODE .
NoDebug	The CPU is running. Debug mode is not active, debug port is tristate. In this mode the target behaves as if the debugger is not connected.
Prepare	Establishes connection to the target and resets the debug module. This debugging mode is used if no CPU shall be debugged. The mode allows to access memory and peripherals.
Attach	User program remains running (no reset) and the debug mode is activated. After this command the user program can be stopped with the break command or if any break condition occurs. The debugger should be in NoDebug mode when performing an Attach .
Go	The CPU is reset and starts executing user code. Debug mode is active. After this command the CPU can be stopped with the Break command or if any break condition occurs.
Up	The CPU is reset and halted at the reset vector. Debug mode is active. In this mode the CPU can be started and stopped. This is the most common way to activate debugging.
StandBy	Activates debugging through power cycles, see “Power Cycles” , page 53.

The **SYStem** modes are not only commands to bring the debugger in a certain debug state, they also reflect the current debug state of the target. **SYStem.Mode Attach** and **SYStem.Mode Go** are only transitional states which will result in an **Up** state on success. Any critical failure will transition the debug state to **SYStem.Mode Down** immediately.

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

NOTE:	<p>For the SYStem.Mode Up the JTAG clock must be high enough. Otherwise the debugger is not able to configure the Halt-After-Reset before the on-chip startup firmware has completed.</p> <p>SYStem.JtagClock 500.0KHz or higher is a saved value for all CPUs.</p>
--------------	---

CPU and Architecture specific SYStem.Option Commands

The **SYStem.Option** commands are architecture and CPU specific commands.

SYStem.Option.BREAKFIX Enable workaround for asynchronous breaking

Format:	SYStem.Option.BREAKFIX [ON OFF]
---------	--

Default: ON for all TriCore v1.3.1 cores, OFF otherwise.

This option is mainly used as a workaround for silicon bug OCDS_TC.028, where the functionality of the loop instruction may be corrupted by FPI accesses to the [0xF7E10000--0xF7E1FFFF] address region. The Debug Status Register (DBGSR), used for asynchronous breaking, resides in this region.

Switching the **SYStem.Option.BREAKFIX ON** will temporarily suspend the TriCore when breaking to allow safe accesses to the critical region. The break is then issued by configuring the Memory Protection System of the TriCore.

Format:	SYSystem.Option.CBSACCEN<x> TarGeT <value>
<x>:	0
<value>:	CerBeruS <mask> [CerBeruS <mask>] ...

Default: CerBeruS.

Handles the behavior of TRACE32 with respect to the CBS_ACCEN<x> registers of the Cerberus block.

TarGeT	TRACE32 will not touch the specified CBS_ACCEN<x> register.
<value>	The value is written to the register when the core is started.

The <value> is the OR-composition of the following:

CerBeruS	Access for the debugger.
<mask>	Arbitrary mask. For valid masks, refer to the TriCore User Manual.

Example 1: Do not touch the CBS_ACCEN0 register.

```
SYSystem.Option.CBSACCEN0 TarGeT
```

Example 2: Enable access from the debugger to the Cerberus block.

```
SYSystem.Option.CBSACCEN0 CerBeruS
```

Example 3: Enable access from the debugger and two more masters (described by <mask>) to the Cerberus block. The two syntax examples are equivalent:

```

;          |----- <value> -----|
;          <mask>                    <mask>
SYSystem.Option.CBSACCEN0 CerBeruS 0x00000001 0x00000002

;          |----- <value> -----|
;          <mask>                    <mask>
SYSystem.Option.CBSACCEN0 0x00000001 CerBeruS 0x00000002
```

Format:

SYStem.Option.DCFREEZE [ON | OFF]

Default: ON.

Protects the cache status of all memories, i.e., it ensures that the data cache tags are not modified on write accesses. If disabled, cache lines may be invalidated/flushed on write accesses. For details, see [“Accessing Cached Memory Areas and Cache Inspection”](#), page 42.

SYStem.Option.DCREAD

Control cache behavior of reads

Format:

SYStem.Option.DCREAD [ON | OFF]

Default: OFF.

If disabled, physical memory is read directly (as seen from the bus).

If enabled, memory reads with [access class D](#): (data) and variable windows display the memory values from the data cache, if valid. If data is not available in cache, physical memory will be read. Memory will be seen as from the CPU's point of view.

TriCore cores do not allow cache access during CPU runtime. All accesses will be automatically redirected to physical memory when the CPU is running, given that run-time memory access is enabled (e.g. with [SYStem.Option.DUALPORT](#)). The following table illustrates the effect.

DCREAD	CPU is	Read	Access class D : is like
ON	stopped	through data cache	DC:
ON	running	physical memory	NC:
OFF	stopped or running	physical memory	NC:

More information is available in chapter [“Accessing Cached Memory Areas and Cache Inspection”](#), page 42.

Format:	SYStem.Option.DSYNC OFF ReadWrite Halt
---------	---

Default: OFF

Automatically executes a `dsync` instruction to make sure the CPU writes all data back. If the data is part of an active cache it will be written to the data cache. Otherwise it is written memory.

OFF	Feature disabled.
ReadWrite	Execute <code>dsync</code> instruction on first read or write on that core.
Halt	Execute <code>dsync</code> instruction when the CPU is halted.

`SYStem.Option.DSYNC ReadWrite` is expected to have better performance in a multi-core system if only one or few cores are of interest.

This system option uses target code execution (see [“Target Code Execution”](#), page 57).

Format:	SYStem.Option.DOWNMODE TriState ReSeT
---------	--

Configures the behavior of **SYStem.Mode Down**:

TriState (default)	All drivers of the debug port are switched off.
ReSeT	The CPU is held in reset.

SYStem.Option.DUALPORT

Implicitly use run-time memory access

Format:	SYStem.Option.DUALPORT [ON OFF]
---------	--

All TRACE32 windows that display memory are updated while the processor is executing code (e.g. **Data.dump**, **List.auto**, **PER.view**, **Var.View**). This setting has no effect if **SYStem.MemAccess** is disabled.

If only selected memory windows should update their content during runtime, leave **SYStem.Option.DUALPORT OFF** and use the access class prefix **E** or the format option **%E** for the specific windows.

As of build 108805, **SYStem.Option.DUALPORT ON** no longer interferes with flash programming.

OCDS-L2 trace only

Format:	SYStem.Option.DataTrace [ON OFF]
---------	---

Default is OFF.

The CPU trace port does not support data tracing. This option uses code instrumentation to output some data.

This is not a real data trace since it doesn't trace the buses. But it may be enough to output some RTOS related information as thread IDs or task IDs.

For transferring more data, you may want to use the **FDX** command group, for a real bus trace, you may want to use the on-chip tracing features of the Emulation Devices (TC17xxED).

SYStem.Option.EndInitProtectionOverride

Override ENDINIT protection

Format:	SYStem.Option.EndInitProtectionOverride [ON OFF AUTO]
---------	--

Default: AUTO.

Handles the behavior of TRACE32 with respect to the ENIDIS bit of the CBS_OSTATE register.

AUTO	TRACE32 automatically sets the bit for write accesses to segment 0xF with access width long. TRACE32 automatically clears the bit after the write access.
OFF	TRACE32 clears the bit immediately and does not further access it.
ON	TRACE32 sets the bit immediately and does not further access it.

Format:SYSystem.Option.HeartBeat [ON | OFF]

Default: OFF.

This option should be used as a workaround for the FPI bus problem on RiderA and AudoLite only. Otherwise bus errors occur when the debugger checks if the processor is running by reading a core special function register via FPI bus. With this option active there are two additional conditions checked first before any read via FPI bus will be done by the debugger. If one of these conditions is true, it is assumed that the processor is still running.

NOTE:RiderA and AudoLite are not supported any more, so this feature is obsolete.

If a RiscTrace is used together with the debugger, the debugger first checks if the RiscTrace is still recording. The RiscTrace may not be used in stack mode.

If a RiscTrace is not used or stack mode is required, the debugger checks if pin 15 (reserved) of the JTAG connector will be toggled. If toggled it is assumed that the processor is running. Therefore it is required to connect a signal there, which will be toggled while the processor is running. A good signal would be the ORed signal of the pipeline status information EMUSTAT0-4 which are available on the OCDS Level 2 connector pin 4-8. Please note that on AudoLite these pins must be activated by `Data.Set 0xF0003748 %Long 0xFFFFFFFF`. The connection of one of these pipeline status information lines (e.g. pin 6) might be sufficient. The signal high level must be lower or equal than VCCS (Pin 2 of JTAG connector) and at least 2.0 V. The signal low level must be lower than 0.8 V. The switching frequency should be higher than 10 Hz.

To avoid other FPI bus conflicts, do not use **SYSystem.Option.DUALPORT** and do not use the access class attribute E.

SYSystem.Option.HoldReset

Reset duration

Format:SYSystem.Option.HoldReset <time>

Configure the minimum time the debugger holds the RESET line active on **SYSystem.Up** or **SYSystem.Mode Go**.

Format:	SYStem.Option.HSMRESTART [ON OFF]
---------	--

Default: OFF

Issues an additional application reset during **SYStem.Up** or if the chip is reset while **SYStem.Option.RESetBehavior Halt** or **SYStem.Option.RESetBehavior RestoreGo** is active. This will release the HSM from halt state. Use only if HSM is not accessible for the debugger.

SYStem.Option.ICFLUSH

Flush instruction cache at "Go" or "Step"

Format:	SYStem.Option.ICFLUSH [ON OFF]
---------	---

Default: ON.

If enabled, an instruction cache flush will be performed with each **Go** or **Step**. This is especially required if software breakpoints are used.

AURIX TC1.6P cores are unable to flush the instruction cache while the core is halted. Thus, flushing will be performed as soon as the core switches to running state. During flush sequence accesses to cacheable memory will stall the CPU, which can result in a slight asynchrony in a multicore system.

Newer AUDO-Future TriCore derivatives (TC11xx, TC1762, TC1764, TC1766, TC1766ED, TC1792, TC1796 and TC1796ED) have a silicon bug which prevents the line buffer cache to be invalidated during CPU halt. See the **demo scripts** for a workaround.

SYStem.Option.IMASKASM

Disable interrupts while single stepping

Format:	SYStem.Option.IMASKASM [ON OFF]
---------	--

Default: OFF.

If enabled, the interrupt enable (IE) bit of the CPU will be cleared during assembler single-step operations. This mean, the interrupt routine is not executed during single-step operations. After single step, the interrupt enable bit is restored to the value before the step. Starting from R.2024.02, read outs of the IE bit during the step are corrected on a best effort basis. The same applies for modifications of the interrupt enable status during the step.

However, direct or indirect reads of the IE bit during the step may show the value modified by the debugger.

On AUDO-Future devices, **IMASKASM** might not always work due to chip bug CPU_TC.115 (see the corresponding Infineon Errata sheets). Enable **SYSystem.Option.STEPONCHIP** concurrently as a workaround.

SYSystem.Option.IMASKHLL

Disable interrupts while HLL single stepping

Format:

SYSystem.Option.IMASKHLL [ON | OFF]

Default: OFF.

If enabled, the interrupt enable (IE) bit of the CPU will be cleared during HLL single-step operations. This means, the interrupt routine is not executed during single-step operations. After single step, the interrupt enable bit is restored to the value before the step.

Direct or indirect reads of the IE bit during the step will show the value modified by the debugger.

Modifications of the interrupt enable status during the step will be lost.

On AUDO-Future devices, **IMASKHLL** might not always work due to chip bug CPU_TC.115 (see the corresponding Infineon Errata sheets). Enable **SYSystem.Option.STEPONCHIP** concurrently as a workaround.

SYSystem.Option.JTAGENSEQ

Use JTAG initialization sequence

Format:

SYSystem.Option.JTAGENSEQ [NONE | JTAG | <signature>]

When JTAG is selected as type of the debug port (see **SYSystem.CONFIG.DEBUGPORTTYPE JTAG**), the debugger must switch the debug port of the CPU to JTAG. This can either be done using $\overline{\text{TRST}}$ or a dedicated JTAG enable sequence.

NONE (default)	Do not send a sequence on SYSystem.Mode Up and SYSystem.Mode Go but use $\overline{\text{TRST}}$. Use standard sequence on SYSystem.Mode Attach .
JTAG	Send standard JTAG enable sequence in all cases.
<signature>	Specify custom upper word for the DAP turn-off command on which the enable sequence is based. Not required for normal operation.

Format: **SYStem.Option.KEYCODE** [<pwd0> <pwd1> ... <pwd7>]

Set debug interface password for AURIX devices in case the debug interface was locked by programming UCB_DBG. The password will be sent automatically on **SYStem.Mode Up**, **SYStem.Mode Go** and **SYStem.DETECT CPU** as well as on reconnects during **SYStem.Mode STANDBY** and after resets.

Each parameter <pwd0> to <pwd7> is a 32-bit number. Omitting all parameters disables sending the password.

SYStem.Option.KEYCODEWarnNotAccepted

Set warning level

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

Default: ON.

In some cases, the chip will not respond to the password exchange sequence enabled by **SYStem.Option.KEYCODE**. This option configures if a warning is emitted in these cases.

SYStem.Option.LBIST

LBIST gap handling

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

Default: OFF.

If LBIST is activated for TC3xx, there is an interruption of the debug connection during device start-up (LBIST gap). This option automatically manages this gap after resets, on reconnects during **SYStem.Mode StandBy**, and when the following commands are executed:

- **SYStem.Mode Up**
- **SYStem.Mode Go**
- **SYStem.DETECT CPU**

OFF	No LBIST expected. Occurrence of an LBIST gap will lead to an error.
ON	LBIST expected. TRACE32 waits for the end of the LBIST gap and reconfigures the debug connection automatically. If no LBIST gap occurs, a warning is displayed.
AUTO	Automatic LBIST handling. TRACE32 waits for the end of the LBIST gap and reconfigures the debug connection automatically. If no LBIST gap occurs, then no warning is displayed.

SYStem.Option.MACHINESPACES

Address extension for guest OSes

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

Default: OFF

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

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

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

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

The host machine always uses machine ID 0. Guests have a machine ID larger than 0. TRACE32 currently supports machine IDs up to 30.
- The debugger address translation (**MMU** and **TRANSlation** command groups) can be individually configured for each virtual machine.
- Individual symbol sets can be loaded for each virtual machine.

Besides debugging virtualized systems, this option can also be used to set up an iAMP system (see **“iAMP Debugging”**, page 22).

Format:	SYStem.Option.MAPCACHE [ON OFF Ponly]
---------	--

Default: ON / Ponly for TC2xx.

For the cache analysis in the **CACHE** command group and **SYStem.Option.DCREAD**, TC2xx and TC3xx require to map the cache via MTU. This operations is potentially intrusive, i.e., it modifies status bits of the MTU.

OFF	Do not map cache.
ON	Map cache if appropriate, e.g., on SYStem.Up .
Ponly	Map cache of TC1.6P cores only if appropriate, e.g., on SYStem.Up . This is the default for TC2xx CPUs to avoid problems with flushing the TC1.6E cores.

For details, see “**CPU Constraints for Cache Evaluation and Cache Inspection**”, page 43.

Format:	SYSystem.Option.OCDSELOW [ON OFF]
---------	--

Default: OFF.

If enabled the `nOCDSE` line of the Debug Cable is driven low, otherwise it is:

- High in case of the Automotive Debug Cable.
- Tri-stated in case of Uni- and Bi-directional Debug Cables.

See [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf) for more information on Debug Cables.

This option is ignored on TriCore devices where the `nOCDSE` line is needed for enabling the OCDS, or when [SYSystem.Option.ETK](#) is not OFF.

SYSystem.Option.OVC

Enable OVERLAY memory access

Format:	SYSystem.Option.OVC [ON OFF] SYSystem.Option.OVERLAY (deprecated)
---------	--

Default: OFF.

Enables evaluation of Data Access Overlay (OVC) configuration during memory accesses.
Only supported for AURIX devices.

ON	Memory will be seen as from the CPU's point of view. The debugger reads the OVC configuration from the target on each access. For flash regions with configured overlay, the overlay memory content will be shown instead of the original flash content.
OFF	Debugger displays the original Flash content only.

See also: [“Data Overlays”](#), page 56.

Format:	SYSystem.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 <i><overlay_id>:<address></i> . 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 <i>execution area</i> (for active overlays) and the <i>storage area</i> . 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:

```
SYSystem.Option.OVERLAY ON
List.auto 0x2:0x11c4                ; List.auto <overlay_id>:<address>
```

See also: [“Code Overlays”](#), page 56

Format:	SYStem.Option.PERSTOP [ON OFF]
---------	---

Default: ON.

This controls the operation mode of the peripherals (e.g. timer), when a debug event is raised, e.g. when the target is halted or when a breakpoint is hit. A debug event causes the peripherals to suspend if this option is activated and the suspend enable bit in the peripheral module is set. Usually this is done in its Clock Register <module>_CLC on AUDO devices, or in the OCDS Control and Status Register <module>_OCS on AURIX devices.

If enabled, TRACE32 will only set the suspend signal. The user has to activate the enable suspend bit in the desired peripheral modules either manually or by his application.

See [Trigger Onchip Commands](#) for the Suspend Switch for advanced programming features.

Format:	SYStem.Option.PMILBFIX [ON OFF]
---------	--

Default: ON.

This option is implemented for AUDO-NG devices only.

This is a workaround for a silicon bug (CPU_TC.053), where the PMI Line Buffer is not invalidated during CPU halt. The bug comes into effect, when the debugger sets software breakpoints, replacing the original instruction at the breakpoint address with a debug instruction and thereby halting the CPU. When execution is resumed, the obsolete debug instruction is still present in the PMI Line Buffer, preventing the CPU to resume execution. Therefore, the PMI Line Buffer needs to be invalidated before the execution is resumed.

If this option is turned on, the PMI Line Buffer will be invalidated when program execution is resumed.

When using software breakpoints in FLASH memory this bugfix is not applicable. The following procedure in your start-up script can be used as a workaround.

```
Data.Assemble <code_address> debug16 nop nop nop nop nop nop nop nop
Data.PROLOG.TARGET <code_address> ++0x1F (<code_address> + 0x20) ++0x1F
Data.PROLOG.ON
Register.Set PC <code_address> + 0x02
Step
```

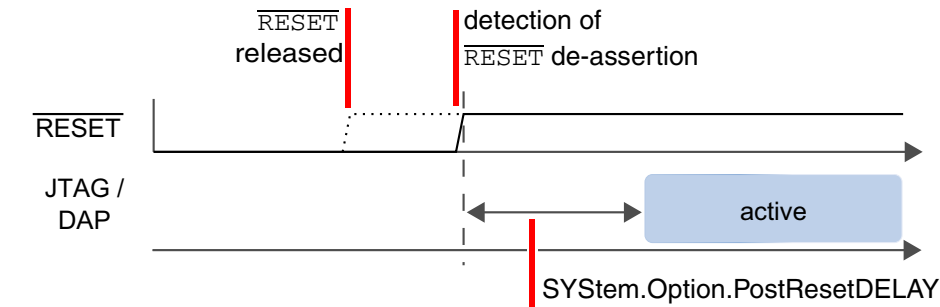
These commands define a simple target program, that is automatically started by the TRACE32 software directly before program execution, fetching the stated instructions at <code_address> and thereby invalidating the PMI Line Buffer. Replace <code_address> with an address in unused and valid memory.

Format:

SYSystem.Option.PostResetDELAY <time>

Default: 100.us

This option affects the behavior of the commands **SYSystem.Mode Up**, **SYSystem.Mode Attach** and **SYSystem.Mode Go**, as depicted below.



It configures the delay that is inserted after the **RESET** line was released by the debugger and the **RESET** line is detected as de-asserted, but before accessing the debug port via DAP or JTAG.

- NOTE:**

- Too short delays may result in debug port errors.
 - Too long delays may result in failure to stop at the reset vector.

SYSystem.Option.ReadOnly

Block all write accesses

Format:

SYSystem.Option.ReadOnly [ON | OFF]

Default: OFF

This option blocks all write accesses to the target. This includes accesses from the debugger itself. Commands like **Go.direct** and **Break.direct** will not work while this option is active.

Format: **SYStem.Option.RESetBehavior** *<mode>*

<mode>:
Halt
RestoreGo
RunRestore

Sets the behavior in case of a reset. For a detailed explanation, see [Debugging through Resets](#).

Halt Halt after reset.

RestoreGo Halt after power-on reset, restore the debugging resources (e.g., breakpoints), and then go. After a soft reset, the debugger does not stop the CPU. The debugging resources remain in place.

RunRestore Restore debugging resources after power-on reset while CPU is running. Breakpoints shortly after a reset will be missed. After a soft reset, the debugger does not stop the CPU. The debugging resources remain in place.

SYStem.Option.ResetDetection

Set how hard resets are detected

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

<mode>:
DEFault
TIMEOUT

Configure which of the following methods TRACE32 uses to detect hard resets.

DEFault A hard reset is detected on a low pulse of the $\overline{\text{RESET}}$ line or a time out of the debug connection.

TIMEOUT A hard reset is detected on a time out of the debug connection only.

Format:

SYSystem.Option.ResetMode <method>

<method>:

PORST | EPORST | SYS | APP

Configures the reset method used by **SYSystem.Up** and **SYSystem.Mode Go**.

PORST	Power on Reset by asserting the $\overline{\text{RESET}}$ line on the debug connector.
EPORST	Emulated Power on Reset using the Cerberus module.
SYS	System reset using the Cerberus module.
APP	Application reset using the Cerberus module.

SYSystem.Option.RESetTMS

State of TMS line at reset

Format:

SYSystem.Option.RESetTMS [ON | OFF]

Default: OFF.

Configures the state of the TMS line at the time the $\overline{\text{RESET}}$ line is released by commands **SYSystem.Mode Up** and **SYSystem.Mode Go**. This option is only required when chaining a TriCore with other devices, e.g., an XC800.

SYSystem.Option.RUNRESTOREDELAY

Delay of restore after reset

[build 148624 - DVD 09/2022]

Format:

SYSystem.Option.RUNRESTOREDELAY <time>

Default: 100.us

Allows to adjust the behavior of TRACE32 in case of a reset or power cycle triggered by the target and **SYSystem.Option.RESetBehavior** is set to **RunRestore**. It allows to adjust the time between the detection of the reset and the begin of the reattach/restore sequence.

Normally, it should not be necessary to adjust this value.

SYStem.Option.SLOWRESET

Long timeout for resets

Format:

SYStem.Option.SLOWRESET [ON | OFF]

Default: OFF.

If enabled, TRACE32 waits up to 25s for the end of a reset.

SYStem.Option.SOFTLONG

Set 32 bit software breakpoints

Format:

SYStem.Option.SOFTLONG [ON | OFF]

Default: OFF.

This option must be used if the program RAM allows only 32-bit accesses. When enabled the debugger uses always 32-bit accesses to write the software breakpoints.

SYStem.Option.SSWWAIT

Emulate SSWWAIT

[build 144272 - DVD 09/2022]

Format:

SYStem.Option.SSWWAIT [IGNore | OFF | AUTO | ON]

Emulates the startup behavior of DMU_SP_PROCONHSMCFG on TC2x and TC3x CPUs.

This command has no effect when the HSM debug instance is enabled. It is intended to ensure correct timing between the TriCore and HSM when the HSM is running in the background.

Default: IGNORE

IGNore	Do not check DMU_SP_PROCONHSMCFG.SSWWAIT at all.
OFF	Assumes that DMU_SP_PROCONHSMCFG.SSWWAIT is off, throws a warning if on.
AUTO	Emulates SSWWAIT if DMU_SP_PROCONHSMCFG.SSWWAIT is on.
ON	Emulates SSWWAIT, throws a warning if DMU_SP_PROCONHSMCFG.SSWWAIT is off.

SYStem.Option.STEPONCHIP

Step with onchip breakpoints

Format:	SYStem.Option.STEPONCHIP [ON OFF]
---------	--

Default: OFF.

If turned on, single stepping is implemented by using a single on-chip breakpoint with Break-After-Make. This is mainly used as workaround for the silicon bug CPU_TC.115 where an already accepted interrupt is not suspended. See chapter [Single Stepping](#) for details.

Note that in case interrupts are not disabled, the interrupt handler will run silently in the background without the user noticing. This leads to data modification which may affect the behavior of the stepped function.

This option has no effect if [SYStem.Option.STEPSOFT](#) is ON.

SYStem.Option.STEPSOFT

Step with software breakpoints

Format:	SYStem.Option.STEPSOFT [ON OFF]
---------	--

Default: OFF.

If turned off, single stepping is performed by using on-chip breakpoints. For an overview, see [“Single stepping”](#).

If turned on, single stepping is implemented by using software breakpoints only. This is necessary for AUDO-NG an earlier cores if the Memory Protection Registers are used by the target application for Memory Protection. For single stepping in flash memory, set up flash configuration and enable [FLASH.AUTO ALL](#).

Note that in case interrupts are not disabled, the interrupt handler will run silently in the background without the user noticing. This leads to data modification which may affect the behavior of the stepped function.

Format: **SYStem.Option.TB1766FIX** [ON | OFF]

Default: OFF.

Bug fix only required for some TriBoards TC1766. On those, two trace pins are swapped. Therefore the debugger switches the signals, so that the trace is working correctly.

SYStem.Option.UNLOCKTIME

Timeout for debug port unlock

[build 148624 - DVD 09/2022]

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

Default: 100.ms

Allows to adjust the behavior of **SYStem.Up**, **SYStem.Mode Go**, etc. It allows to adjust the maximum time TRACE32 waits for the debug port to unlock. Normally, it should not be necessary to adjust this value.

SYStem.Option.WDTFIX

Disables the watchdog on SYStem.Up

Format: **SYStem.Option.WDTFIX** [ON | OFF]
SYStem.Option.WATCHDOGFIX [ON | OFF] (deprecated)
SYStem.Option.TC1796FIX [ON | OFF] (deprecated)
SYStem.Option.TC19XXFIX [ON | OFF] (deprecated)

Default: OFF.

Some early TriCore AUDO derivatives, e.g. TC1130, TC1796, TC19xx, ..., have a silicon bug where the watchdog is not suspended when the CPU is halted. As a workaround, this options disables the watchdog on **SYStem.Mode Up** and **SYStem.Mode Go**.

To find out if your derivative is affected by this bug, have a look at the **“Example Scripts”** and at the Errata Sheets available from Infineon.

Do not use too low a JTAG clock frequency otherwise the debugger might be too slow to disable the watchdog in time. The default frequency should be fine.

Format:

SYStem.Option.WDTSUS [ON | OFF]

Default: OFF.

By default the TriCore watchdog timer is disabled when OCDS is enabled, i.e. when the debugger is attached.

Setting this option to ON will link the watchdog timer to the suspend bus, i.e. the watchdog timer will be running only if the suspend bus is inactive. In other words: The watchdog timer will run concurrently with the CPU and halt concurrently with the CPU. TRACE32 will concurrently set **SYStem.Option.PERSTOP ON** to automatically pull the suspend bus when breaking. This allows debugging the watchdog timer.

Consult the Infineon documentation for details.

SYStem.RESetOut

In-target reset

Format:

SYStem.RESetOut

When issuing this command, the debugger will drive the nRESET line low for 2 ms. This will reset the target including the CPU but not the debug port. The debugger might report one or more bus errors during the reset.

SYStem.state

Open SYStem.state window

Format:

SYStem.state

Opens the **SYStem.state** window with settings of CPU specific system commands. Settings can also be changed here.

CPU specific TrOnchip Commands

TriCore derivatives such as the TC1100 and devices of the AUDO family have a Multi-Core Break Switch (MCBS) providing two internal Break Buses and a Suspend Switch.

TrOnchip.BreakBusN.BreakIN

Configure break pin of "BreakBus N"

Format:	TrOnchip.BreakBus0.BreakIN [ENable DISable] TrOnchip.BreakBus1.BreakIN [ENable DISable]
---------	--

Default: DISable.

When enabled, the nBRKIN pin (nBRKOUT pin) of the JTAG connector acts as input for BreakBus 0 (BreakBus 1). Available for devices with an MCBS (up to AUDO-MAX family).

For TC1100, TC1115 and TC1130 only Break Bus 0 (nBRKIN pin) can be configured as input.

NOTE:	Configuring a BreakBus as BreakIN and BreakOUT at the same time will result in undefined behavior.
--------------	---

TrOnchip.BreakBusN.BreakOUT

Configure break pin of "BreakBus N"

Format:	TrOnchip.BreakBus0.BreakOUT [ENable DISable] TrOnchip.BreakBus1.BreakOUT [ENable DISable]
---------	--

Default: DISable.

When enabled, the nBRKOUT pin (nBRKIN pin) of the JTAG connector acts as output for BreakBus 1 (BreakBus 0). Available for devices with an MCBS (up to AUDO-MAX family).

For TC1100, TC1115 and TC1130 only Break Bus 1 (nBRKOUT pin) can be configured as output.

NOTE:	Configuring a BreakBus as BreakIN and BreakOUT at the same time will result in undefined behavior.
--------------	---

Format:	TrOnchip.BreakIN.<target> [BreakBus0 BreakBus1]
<target>:	TriCore MCDS SusSWitch

Default: BreakBus0.

Connects the break <target> with the specified **BreakBus**. A break target must always be connected to a **BreakBus**. Available for devices with an MCBS (up to AUDO-MAX family). Not all targets are available on each derivative.

TriCore	Sets the TriCore as target.
MCDS	MCDS is only available on an Emulation Device.
SusSWitch	SusSWitch is the break to Suspend Switch.

Format:	TrOnchip.BreakOUT.<source> [BreakBus0 BreakBus1] TrOnchip.BreakOUT.HaLTEN [ENable DISable]
<source>:	TriCore MCDS DMA SBCU RBCU MLI0 MLI1

Default: BreakBus0.

NOTE:	Configuring a BreakBus as BreakIN and BreakOUT at the same time will result in undefined behavior.
-------	---

<source>

Connect the break <source> with the specified **BreakBus**. A break source must always be connected to a **BreakBus**. Available for devices with an MCBS (up to AUDO-MAX family). Not all sources are available on each derivative.

Depending on the derivative, not all trace sources are available. The break source **MCDS** is only available on an Emulation Device. The following table shows which sources are available on which device. Sources separated by a / are shared and can be disabled in the corresponding chip modules separately. For information on how to program the module correctly, refer to the Infineon User's Manual of your device.

TC11xx	AUDO-NG	AUDO-Future
TriCore	TriCore	TriCore
DMA	DMA	DMA
BCU / MLI0	S-BCU / MLI0	S-BCU / (MCDS)
MLI1	R-BCU / MLI1	MLI0
	(MCDS)	MLI1

NOTE:	Note that the R-BCU is only available on TC1792, TC1796 and TC1796ED. Note that the TrOnchip.view window summarizes shared sources within the same box.
-------	---

HaLTEN

Default: DISable.

This special target is only available for the AUDO-Future family. When enabled, the CPU HALT state is broadcast as a level signal on the connected break bus.

NOTE:	Do not enable the break target PCP in case HaLTEN is enabled for TriCore and TriCore is the break source to the same break bus where PCP is break target. In this case the synchronous multicore start will fail.
-------	---

Format:TrOnchip.CONVert [ON | OFF]

Converts on-chip breakpoints, freeing on-chip breakpoint resources.

- ON

On cores up to core version 1.3.1, breakpoint ranges will be reduced to single address breakpoints.

As of core version 1.6, breakpoint ranges from and to the same address will be converted to a single address breakpoint.
- OFF

No conversion.

TrOnchip.CountX

Event X counter value

Format:TrOnchip.CountX <value>

This feature is only available for the 5 V trace preprocessor (LA-7879).

The counter <value> is incremented each time the event X has occurred. When the value is equal to 0, the assigned status (TR[1..0] CR or EXT) is sampled.

On 3.3 V preprocessors (LA-7928), this counter is not active, the status will be sampled every time the event X occurred.

TrOnchip.CountY

Event Y counter value

Format:TrOnchip.CountY <value>

See [TrOnchip.CountX](#).

Format:	TrOnchip.EXTernal [ON OFF]
---------	------------------------------

Default: OFF.

When enabled, TriCore will break on an break signal distributed via the BreakBus which TriCore listens to.

For AURIX devices, this option is always on.

NOTE:	Enabling this feature will disturb the OCDS-L2 break actions TraceON, TraceOFF, TraceEnable and TraceTrigger. Instead TriCore will stop when hitting such a break action. Unconditional tracing is still possible, OCDS-L3 (On-chip trace) is not affected.
-------	--

TrOnchip.PERSTOPOUT

Route suspend signal to pin

Format:	TrOnchip.PERSTOPOUT OFF <pin>
<pin>	Pin0 Pin1 ... Pin7

This command configures the OTGS of TC2xx and later devices to route the peripheral suspend signal (see [Suspending Peripherals](#)) to a pin. For the exact pins, please refer to the chip data sheet. Pin0 corresponds to $\overline{TGO0}$, Pin1 corresponds to $\overline{TGO1}$, etc.

TrOnchip.RESet

Reset settings for the on-chip trigger unit

Format:	TrOnchip.RESet
---------	----------------

Resets the settings for the trigger on-chip unit to default.

Format:	TrOnchip.SoftWare [ON OFF]
---------	-------------------------------------

Default: ON.

When enabled, TriCore will break when executing a debug instruction.

NOTE:	When this feature is disabled, the TriCore CPU will treat all debug instructions as nop instructions. This will silently disable all software breakpoints.
--------------	--

TrOnchip.SusSWitch

Enable or disable suspend switch

Format:	TrOnchip.SusSwitch [ENable DISable]
---------	--

Default: DISable.

Enables or disables forwarding of a break signal to the suspend bus.

Available for devices with an MCBS (up to AUDO-MAX family).

TrOnchip.SusSWitch.FORCE

Force generation of suspend signal

Format:	TrOnchip.SusSwitch.FORCE [ENable DISable]
---------	--

Default: DISable.

When enabled, the suspend switch generates a suspend signal independent of any state or mode.

Available for devices with an MCBS (up to AUDO-MAX family).

Format:	TrOnchip.SusSwitch.Mode [CAPTURE DIRECT CLEAR]
---------	--

Default: CAPTURE.

In **DIRECT** mode, the suspend switch forwards the signal coming from the break bus directly to the suspend bus.

In **CAPTURE** mode, a high-low transition on the break bus is stored in a flip-flop. Its state is used to drive the suspend bus. The flip-flop is reset with the **CLEAR** option. Available for devices with an MCBS (up to AUDO-MAX family).

TrOnchip.SusTarget

Connect special targets to the suspend bus

Format:	TrOnchip.SusTarget <target> ON OFF TrOnchip.SusTarget.FORCE ENable DISable
<target>:	TriCore PMU (AUDO devices with MCBS) DMA HSS (TC2x) DMA DMA1 HSS0 HSS1 (TC3x, TC4x)

Normally each peripheral has a clock register <mod>_CLC (AUDO) or a debug control register <mod>_OCS (AURIX TC2x and later) where it can be connected to the suspend bus. Some modules do not have such a register and are directly connected to the break switch. These modules can be programmed here.

If **FORCE** is selected, all special targets enabled for suspend are suspended immediately, independent of the current suspend bus state. This is also true for PCP if enabled in ICD PCP. Available for devices with an MCBS (up to AUDO-MAX family).

TrOnchip.SYNCHRONOUS

Switches mode for data breakpoints

Format:	TrOnchip.SYNCHRONOUS [ON OFF]
---------	---------------------------------

Default: OFF.

Switches the Break Before Make flag in TRxEVT registers of AURIX cores. This flag is only effective for data breakpoints.

If the flag is ON the processor stops on a breakpoint hit before the new value is written to the respective destination/address. If the flag is set OFF the processor stops after the new value is written.

TrOnchip.TDelay

Trace trigger delay

Format: **TrOnchip.TDelay** <value> (deprecated)
Use **Break.Set** instead.

The trace will not stop immediately after trigger if a trigger delay is programmed.

TrOnchip.TExtMode

Mode for external trigger input

Format: **TrOnchip.TExtMode** [COMP | EDGE]

Default: EDGE.

This defines if the external trigger pin on the preprocessor (only LA-7879) reacts on level (**COMP**) or edge (**EDGE**).

TrOnchip.TExtPol

Polarity of external trigger input

Format: **TrOnchip.TExtPol** [+ | -]

Default: -.

This defines if the external trigger pin on the preprocessor (only LA-7879) reacts on active high/positive edge (+) or active low/negative edge (-).

TrOnchip.TMode

Trace mode

Format: **TrOnchip.TMode** <mode> (deprecated)
Use **Break.Set** instead.

This command is obsolete and only left for compatibility reasons.

Format:	TrOnchip.TR0 <source> <event>
<source>:	Code <breakpoint> Address <breakpoint> CYcle OFF Read Write Access
<event>:	WATCH [ON OFF]

Specifies the source for the trigger event 0 and the action associated with the trigger event. Available for devices with an MCBS (up to AUDO-MAX family).

Code <breakpoint>	Code range for the debug event generation, the code range has to be marked by an Alpha Beta Charly Delta Echo breakpoint.
Address <breakpoint>	Data range for the debug event generation, the data range has to be marked by an Alpha Beta Charly Delta Echo breakpoint.
CYcle	Cycle type for the data range: OFF Read Write Access

Specify the event for trigger source 0

WATCH [ON OFF]	Default: OFF. When WATCH is ON and the event X or Y occurs, program execution will stop and the status will be output. If switched to OFF, only the status will be output.
----------------------------	---

NOTE:	This way of setting Trace Triggers is obsolete. The correct way is to use Breakpoints .
--------------	--

Example 1: Stop the program execution when the function `sieve` is entered.

Break.Set sieve /Alpha	; Set a breakpoint of the type Alpha to sieve
TrOnchip.TR0.Code Alpha	; Select the code range as source for the trigger event 0 generation and assign the breakpoint type Alpha
TrOnchip.TR0.WATCH OFF	; The program execution will be stopped if the trigger event 0 occurs

Example 2: Stop the program execution when a write access to flags[3] occurs.

```
; Set a breakpoint of the type Alpha to flags[3]

Var.Break.Set \\demo\taskc\flags[3]; /Alpha

; Select the data range as source for the trigger event 0 generation and
; assign the breakpoint type Alpha

TrOnchip.TR0.Address Alpha

; Select Write as cycle type for the data access

TrOnchip.TR0.CYcle Write

; The program execution will be stopped if the trigger event 0 occurs

TrOnchip.TR0.WATCH OFF
```

TrOnchip.TR1

Specify trigger event 1

See [TrOnchip.TR0](#) for further information.

TrOnchip.state

Show on-chip trigger window

Format:	TrOnchip.state
---------	----------------

Shows the trigger on-chip window.

Format:	TrOnchip.WatchPin OFF <pin>
<trigger>	WATCH Alpha Beta Charly Delta Echo
<pin>	Pin0 Pin1 ... Pin7

This command configures the OTGS of TC2xx and later devices to route a core trigger to a pin. This can be used to correlate program events with external signals. For the exact pins, please refer to the chip data sheet. Pin0 corresponds to $\overline{TGO0}$, Pin1 corresponds to $\overline{TGO1}$, etc.

Additional configuration for the pins, e.g., speed grade, might be required.

Example:

This example shows how to route a range breakpoint on "func1" to pin " $\overline{TGO0}$ ". The pin will be active (low) while the program counter is inside func1 but not inside a called function.

```
Break.Set sYmbol.RANGE(func1) /Alpha
TrOnchip.WatchPin.Alpha Pin0
```

An extended example can be found at
~~/demo/tricore/etc/trace_trigger/watchpins/aurix_watchpin.cmm.

Format:	TrOnchip.X <source>
<source>:	TR0 TR1 CR EXT

The on-chip comparator unit can react on different sources. Available for devices with an MCBS (up to AUDO-MAX family).

TR0	On-chip address comparator TR0
TR1	On-chip address comparator TR1

CR	Core Event: MTCR/ MFCR instruction is executed or a core SFR is modified.
EXT	External Break In pin is asserted.

TrOnchip.Y

Select trigger source Y

See [TrOnchip.X](#) for further information.

Trace Connector

For more information on connectors, adapters, converters, and cables, see [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf).

Technical Data for Debugger

Mechanical Dimensions

	OCDS Uni-Dir Debug Cable V0	Bidirectional Cables and Automotive Debug Cables
Length:	4.2 cm	5.5 cm
Width:	5.2 cm	6.25 cm
Height:	1.6 cm	1.7 cm
Length of cable:	approx. 37.5 cm	approx. 37.8 cm

The Debug Cables with new housings are shipped within Europe since March 2006. They have a removable cable and are RoHS compliant. The electrical components and the schematics are identical to ones in the old housing.

Further details are available in [“Application Note Debug Cable TriCore”](#) (app_tricore_ocds.pdf).

This chapter covers the following topics:

- [“Parallel Off-chip Trace - OCDS-L2 Flow Trace \(Analyzer\)”](#), page 134

Parallel Off-chip Trace - OCDS-L2 Flow Trace (Analyzer)

Use [Trace.METHOD Analyzer](#) for selecting the Analyzer.

Overview

This chapter describes the OCDS-L2 Flow Trace, which is available on some devices of the AUDO-NG family, e.g. the TC1796.

For all other traces, please refer to chapter [“Tracing”](#), page 60.

Quick Start for Tracing with OCDS-L2 Trace (Analyzer)

It is assumed that you are tracing a TC1766 B-Step on an Infineon TriBoard-TC1766.300 or above.

1. Prepare the Debugger

Load your application and prepare for debug. See [“Quick Start for OCDS-L1 Debugger”](#) for more details.

2. Connect the PreProcessor to the Trace Connector on the Target

Plug the preprocessor into the trace connector on the target board. In case of an AMP40 connector you need to care for the correct orientation of the connector. Check for Pin 1.

3. Configure the Trace Port on Your Target

```
Data.Set 0xF0001110 %Long 0xA0A0A0A0 ; enable Port 5 for OCDS-L2 output
Data.Set 0xF0001114 %Long 0xA0A0A0A0
Data.Set 0xF0001118 %Long 0xA0A0A0A0
Data.Set 0xF000111C %Long 0xA0A0A0A0
```

On TriBoard-TC1766.300 the trace connector is connected to GPIO port 5 which needs to be set up for tracing.

4. Fine Tuning

```
Analyzer.Clock 20.0MHz ; specify CPU clock
```

The preprocessor uses a compression algorithm which affects the accuracy of the timestamp information. For improving the accuracy by the factor 4, specify the CPU clock frequency.

The trace is now configured.

5. Start and Stop Tracing

```
Go ; start tracing
Break ; stop tracing
```

Recording is stopped when the TriCore halts, e.g. when a breakpoint was hit.

6. View the Results

```
Analyzer.List ; view recorded trace data
```

Supported Features

- Program Flow Trace for TriCore
- Program Flow Trace for PCP
- Timestamp
- **Simple Trace Control**

NOTE:	Note that OCDS-L2 does not allow to trace TriCore and PCP at the same time.
-------	---

Version History

The trace hardware is under constant development. So it is possible that you have different trace hardware versions. Only the latest trace hardware versions are housed (see [“Mechanical Dimensions”](#) in Application Note Debug Cable TriCore, page 66 (app_tricore_ocds.pdf)).

Order number	LA-7879	LA-7879	LA-7928
Housing	no	no	yes
PCB version	MERF1-4	MERF5-6	MERF7-
Connector	AMP40	SAMTEC60	SAMTEC60
Counter	yes	yes	no
Frequency	80 MHz	133 MHz	180 MHz
Termination	passive (clock)	passive (clock)	active 50Ohm Thevenin
Threshold	no	no	yes
Trace control	yes	yes	yes
JTAG on board	no	no	yes
Trace depth	x2	x2	x4
Number of cable	1	1	2

Timestamp Accuracy

The TriCore does not generate any timestamps for OCDS-L2 trace.

The trace preprocessor demultiplexes multiple OCDS-L2 trace samples (2 in case of LA-7879, 4 in case of LA-7928) into a single trace package which is stored in one trace memory frame and marked with a timestamp generated by the trace hardware. The accuracy is trace hardware dependent, see [“Analyzer”](#) (general_ref_a.pdf) for details.

This results in inaccurate execution times. Especially when [Trace.PortFilter](#) is used, you can get extremely high time values or even durations below the resolution of the trace hardware - shown as e.g. "< 20 ns". PowerView is able to compensate for the inaccuracy by setting [Analyzer.CLOCK](#) *<cpufreq>* according to the CPU frequency.

Note that in case of high CPU frequencies and [Analyzer.CLOCK](#) *<cpufreq>* is set, negative cycles may be displayed. In case of negative timestamps, it is recommended to adjust the *<cpufreq>*. Slightly higher values may already eliminate all or most of the negative timestamps.

OCDS-L2 trace is a Program Flow Trace and so timestamps will never be instruction accurate.

Concurrent Usage of OCDS-L2 Off-chip Trace and OCDS-L3 On-chip Trace

The parallel OCDS-L2 off-chip trace and the OCDS-L3 on-chip trace can be used in parallel with restrictions. For details, see [“Concurrent Usage of Different Trace Methods”](#) in MCDS User’s Guide, page 98 (mcds_user.pdf).

This use case is not recommended.

Simple Trace Control

TriCore OCDS-L2 trace supports the same features:

- Trace all
- Trace to
- Trace from
- Trace from to
- Trigger
- Enable



OCDS level 2 is a flow trace. That means only relative program flow information is given out by the CPU.
TRACE32 needs synchronization points for starting trace disassembling. The trace cannot be shown correctly until such a synchronization point is reached.

Trace Break Signals (OCDS-L2)

The trace connector has three break lines (see [Trace Connector](#)) which will output a defined status when certain events occurred a definable number of times:

event	TR0	TR1
[EMU] BREAK0	1	1
[EMU] BREAK1	0	0
[EMU] BREAK2	0	1

BREAK[2..0] corresponds to EMUSTAT[7..5] in the trace.

TR[1..0], CR and EXT can be assigned to events X and Y.

A successfully loaded target application is needed for the following examples.



Trace is just a pointer to the currently selected trace method (see [Trace.METHOD](#)).

Example 1: Trace from

Start recording at address 0xA0001000.

```
Analyzer.view                ; show trace setup window
Analyzer.Mode STACK          ; set trace to stack mode
Analyzer.List                ; show trace list window
Break.Set 0xA0001000 /TraceON ; set start address
```

Example 2: Trace to

Stop recording at address 0xA0001100, but do not halt the CPU.

```
Analyzer.view                ; show trace setup window
Analyzer.Mode STACK          ; set trace to stack mode
Analyzer.List                ; show trace list window
Break.Set 0xA0001100 /TraceOFF ; set end address
```

Example 3: Trace from to

Start recording at address 0xA0001000 and stop recording at address 0xA0001100, but do not halt the CPU. If recording was stopped once it will restart when the start address is reached once more.

```
Analyzer.view                ; show trace setup window
Analyzer.Mode STACK          ; set trace to stack mode
Analyzer.List                ; show trace list window
Break.Set 0xA0001000 /TraceON ; set start address
Break.Set 0xA0001100 /TraceOFF ; set end address
```

Example 4: Trace Trigger

Trigger recording at address 0xA0001200. The trace will stop recording after a programmable period, see [Trace.Trigger](#) for more information.

```
Analyzer.view           ; show trace setup window
Analyzer.Mode STACK     ; set trace to stack mode
Analyzer.List           ; show trace list window
Break.Set 0xA0001200 /TraceTrigger ; set trigger address
```

Example 5: Trace Enable

When address 0xA0001400 is reached, write a sample into the trace memory.

```
Analyzer.view           ; show trace setup window
Analyzer.Mode STACK     ; set trace to stack mode
Analyzer.List           ; show trace list window
Break.Set 0xA0001400 /TraceEnable ; set enable address
```

No Data in Trace.List Visible

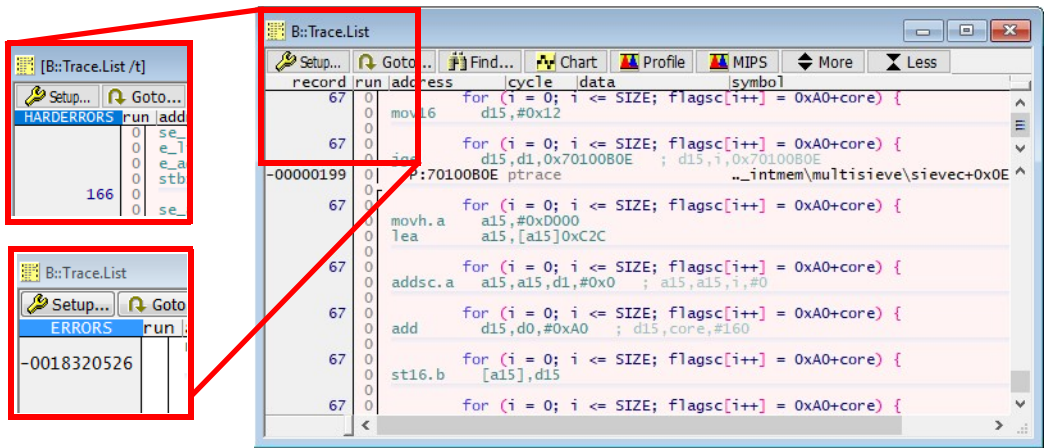
Different reasons are possible:

- CPU trace port not mapped.
Mostly the trace port shares its functionality with another peripheral. In this case, one or more dedicated registers have to be set correctly. See also the demo scripts within the TRACE32 installation directory: `~/demo/tricore/hardware/*`.
- Reference voltage not set up correctly (LA-7928 only).
Reference voltage pin on trace connector connected? If not, the threshold value must be set to the middle of the trace signal manually (half of target I/O voltage level).
- Poor trace/clock signal.
Change termination setting (LA-7928 only) and threshold value. If a change of the setup does not help, further target hardware modifications are required. Please ask the technical support for details (support@lauterbach.com).

Error Diagnosis

NOTE:	Advanced trace analysis commands like Trace.STATistic.Func , Trace.STATistic.TASK display accurate results only if the trace recording works error-free.
--------------	--

Error messages are displayed in the upper left corner of the **Trace.List** window:

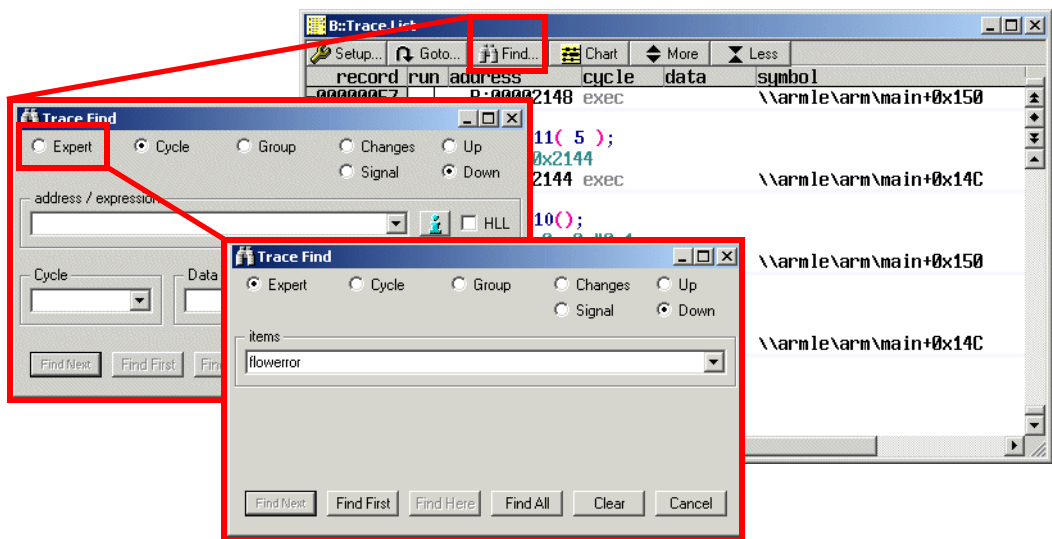


TRACE32 uploads only the requested trace information to the host to provide a quick display of the trace information. Consequently, potential errors outside the uploaded range are not immediately visible.

There are several ways to search for errors within the trace, all of them will force TRACE32 to upload the complete trace information to the host:

1. The **Trace Find** window

Pushing the **Find ...** button of the **Trace.List** window opens a special search window:



Select **Expert** and enter “flowererror” in the item field. The item entry is not case sensitive. Use the **Find First/Find Next** button to jump to the next flowererror within the trace. **Find All** will open a separate window with a list of all flowererrors.

2. Using command **Trace.FindAll , FLOWERERROR**

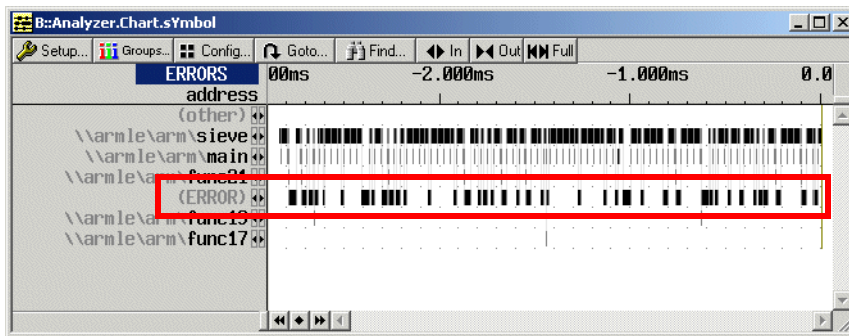
This command will search for **errors** within the entire trace buffer. The records will be listed within a separate window. This command corresponds to the **FindAll** option described above.

The image shows a window titled 'B::a.fa, flowererror' which displays a list of error records. The records are organized in a table with columns: '43', 'run', 'address', 'cycle', 'data', 'symbol', and 'ti.back'. The 'run' column contains addresses starting with 'R:'. The 'symbol' column contains function names like 'func21+0x10' and 'sieve+0x44'. The 'ti.back' column contains time values in microseconds (us).

43	run	address	cycle	data	symbol	ti.back
-00203067	R:00001E34	exec			\\armle\\arm\\func21+0x10	
-00198311	R:00001E34	exec			\\armle\\arm\\func21+0x10	65.500us
-00195443	R:0000226C	exec			\\armle\\arm\\sieve+0x44	39.530us
-00193559	R:00001DE4	exec			\\armle\\arm\\func19+0x10	25.970us
-00189917	R:0000227C	exec			\\armle\\arm\\sieve+0x54	49.930us
-00183125	R:00002254	exec			\\armle\\arm\\sieve+0x2C	93.570us
-00175149	R:0000227C	exec			\\armle\\arm\\sieve+0x54	109.800us
-00173609	R:00002254	exec			\\armle\\arm\\sieve+0x2C	21.200us
-00171469	R:0000226C	exec			\\armle\\arm\\sieve+0x44	29.500us
-00167741	R:0000227C	exec			\\armle\\arm\\sieve+0x54	51.360us
-00165011	R:00001E38	exec			\\armle\\arm\\func21+0x14	37.640us
-00162243	R:00002254	exec			\\armle\\arm\\sieve+0x2C	38.230us
-00159333	R:00002254	exec			\\armle\\arm\\sieve+0x2C	39.770us
-00149817	R:00002254	exec			\\armle\\arm\\sieve+0x2C	131.000us
-00140301	R:00002254	exec			\\armle\\arm\\sieve+0x2C	131.000us

3. Using command **Trace.Chart.sYmbol**

This command will start a statistical analysis. An additional symbol (ERROR) is shown if errors where found.



The search could take a long time depending on the used memory size of the trace module and the type of host interface. Check the status to estimate the time.

Error Messages

One of the following 2 errors may occur:

1. **HARDERROR**

There are no valid trace data. Possible reasons are:

- Traceport multiplexed with other IO functions (no valid trace data)
- Trace signal capturing failed (setup/hold time violations)
- Wrong version of PowerTrace module
- Target frequency too high

2. **FLOWERROR**

The traced data are not consistent with the code in the target memory. Possible reasons are:

- Memory contents has changed.
Self modifying code is not supported.
- Wrong trace data (as result of **HARDERRORS**)
- CPU trace port not mapped.

Mostly the trace port shares its functionality with another peripheral. In this case, one or more dedicated registers have to be set correctly. See also the demo scripts within the TRACE32 installation directory: `~/demo/tricore/hardware/*`.

- Reference voltage not set up correctly (LA-7928 only).

Reference voltage pin on trace connector connected? If not, the threshold value must be set to the middle of the trace signal manually (half of target I/O voltage level).

- Poor trace/clock signal.

Change termination setting (LA-7928 only) and threshold value. If a change of the setup does not help, further target hardware modifications are required. Please ask the technical support for details (**support@lauterbach.com**).

- Setup/hold time violation.
 - 3 ns setup / 1 ns hold time needed.
 - sample point: falling edge.
-