Supported Features 59
Trace Evaluation 60
Impact of the Debugger on FPI Bus Tracing 60
Simple Trace Control 61
Examples 61

**CPU specific Commands** 65

Analyzer.Mode PCP ............................ Select PCP trace 65
CPU specific BMC Commands 65
BMC.SELECT ................................. Select counter for statistic analysis 66
BMC.<counter>.ATOB ........................ Control A-to-B mode 66
BMC.<counter>.TRIGMODE ................. BMC trigger mode 66
BMC.<counter>.TRIGVAL .................... BMC trigger value 66
CPU specific SYStem.CONFIG Commands 68
SYStem.CONFIG.state ...................... Display target configuration 68
SYStem.CONFIG .................. Configure debugger according to target topology 69
Daisy-Chain Example 71
TapStates 72
SYStem.CONFIG.CORE ........................ Assign core to TRACE32 instance 73
SYStem.CONFIG BreakPIN .................. Define mapping of break pins 74
SYStem.CONFIG DAP ........................ Configure DAP interface 75
SYStem.CONFIG DAP.BreakPIN .......... Define mapping of break pins 75
SYStem.CONFIG DAP.DAPENable ........ Enable DAP mode on PORST 75
SYStem.CONFIG DAP.USERn ............... Configure and set USER pins 76
SYStem.CONFIG.DEBUGPORT ............ Select target interface 77
SYStem.CONFIG.DEBUGPORTTYPE .... Set debug cable interface mode 77
SYStem.CONFIG DXCPL ........................ Configure DXCPL 78
SYStem.CONFIG DXCPL.Timing .......... Configure SPD timing for DXCPL 78
SYStem.CONFIG.EXTWDTDIS ............. Disable external watchdog 78
SYStem.CONFIG PortSHaRing ............ Control sharing of debug port with other tool 79
SYStem.CPU ................................. Select CPU 80
SYStem.JtagClock ......................... Set the JTAG frequency 81
SYStem.LOCK ............................... Tristate the JTAG port 82
SYStem.MemAccess ....................... Run-time memory access (non-intrusive) 83
SYStem.Mode .............................. Establish the communication with the CPU 84
CPU and Architecture specific SYStem.Option Commands 85
SYStem.Option BREAKFIX ................. Enable workaround for asynchronous breaking 85
SYStem.Option CBSACCEN<x> ............ Cerberus access protection 86
SYStem.Option DCFREEZE ................. Do not modify cache structure 87
SYStem.Option DCREAD ................... Control cache behavior of reads 87
SYStem.Option DSYNC .................... Force data synchronization 88
SYStem.Option DOWNMODE Behavior of SYStem.Mode Down 89
SYStem.Option DUALPORT Implicitly use run-time memory access 89
SYStem.Option DataTrace Enable data tracing 90
SYStem.Option EndInitProtectionOverride Override ENDINIT protection 90
SYStem.Option ETK Debugging together with ETK from ETAS 90
SYStem.Option HeartBeat Bug fix to avoid FPI bus conflict 91
SYStem.Option HoldReset Reset duration 91
SYStem.Option HSMRESTART Restart HSM on connect 92
SYStem.Option ICFLUSH Flush instruction cache at 'Go' or 'Step' 92
SYStem.Option IMASKASM Disable interrupts while single stepping 92
SYStem.Option IMASKHLL Disable interrupts while HLL single stepping 93
SYStem.Option INTSTART Start address of interrupt routines 93
SYStem.Option INTUSE Number of implemented interrupts 93
SYStem.Option JTAGENSEQ Use JTAG initialization sequence 94
SYStem.Option KEYCODE Set debug interface password 94
SYStem.Option LBIST LBIST gap handling 94
SYStem.Option MAPCACHE Map cache automatically 95
SYStem.Option OCDSELOW Set OCDS line to low 96
SYStem.Option OVC Enable OVERLAY memory access 96
SYStem.Option OVERLAY Enable overlay support 97
SYStem.Option PERSTOP Enable global peripheral suspend 98
SYStem.Option PMILBFIX Enable PMI line buffer invalidation workaround 99
SYStem.Option PostResetDELAY Delay after RESET is released 100
SYStem.Option ReadOnly Block all write accesses 100
SYStem.Option RESetBehavior Set behavior when a reset occurs 101
SYStem.Option ResetDetection Set how hard resets are detected 101
SYStem.Option ResetMode Select reset method 102
SYStem.Option RESetTMS State of TMS line at reset 102
SYStem.Option SLOWRESET Long timeout for resets 102
SYStem.Option SOFTLONG Set 32 bit software breakpoints 103
SYStem.Option STEPONCHIP Step with onchip breakpoints 103
SYStem.Option STEPSOFT Step with software breakpoints 103
SYStem.Option TB1766FIX Bug fix for some TC1766 TriBoards 104
SYStem.Option TRAPSTART Start address of trap vectors 104
SYStem.Option WDTFIX Disables the watchdog on SYStem.Up 104
SYStem.Option WDTSUS Link the watchdog timer to the suspend bus 105
SYStem.RESETOut In-target reset 105
SYStem.state Open SYStem.state window 105

CPU specific TrOnchip Commands ................................................................................ 106
TrOnchip.BreakBusN.BreakIN Configure break pin of 'BreakBus N' 106
TrOnchip.BreakBusN.BreakOUT Configure break pin of 'BreakBus N' 106
TrOnchip.BreakIN.<target> Connect break <target> to BreakBus 107
TrOnchip.BreakOUT.<source> Connect break <source> to BreakBus 107
<source>

HaLTEN

TrOnchip.CONVert
Adjust range breakpoint in on-chip resource

TrOnchip.CountX
Event X counter value

TrOnchip.CountY
Event Y counter value

TrOnchip.EXTernal
Configure TriCore break on BreakBus event

TrOnchip.PERSTOPOUT
Route suspend signal to pin

TrOnchip.RESet
Reset settings for the on-chip trigger unit

TrOnchip.SoftWare
Configure 'TriCore' break on debug instruction

TrOnchip.SusSWitch
Enable or disable suspend switch

TrOnchip.SusSWitch.FORCE
Force generation of suspend signal

TrOnchip.SusSWitch.Mode
Set suspend switch mode

TrOnchip.SusTarget
Connect special targets to the suspend bus

TrOnchip.SYNCHRONOUS
Switches mode for data breakpoints

TrOnchip.TCompress
Trace data compression

TrOnchip.TDelay
Trace trigger delay

TrOnchip.TExtMode
Mode for external trigger input

TrOnchip.TExtPol
Polarity of external trigger input

TrOnchip.TMode
Trace mode

TrOnchip.TR0
Specify trigger event 0

TrOnchip.TR1
Specify trigger event 1

TrOnchip.state
Show on-chip trigger window

TrOnchip.WatchPin
Route core trigger to pin

TrOnchip.X
Select trigger source X

TrOnchip.Y
Select trigger source Y

Technical Data

JTAG Connector

Trace Connector

AMP 40 Connector

ERF8 22-pin Power.org Connector

Samtec 60 Connector

Technical Data for Debugger

Operation Voltage

Mechanical Dimensions

Technical Data for Trace

Operation Frequency

Operation Voltage

Mechanical Dimensions

Appendix

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

Overview

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

Supported Features
Version History 128
Timestamp Accuracy 128
Concurrent Usage of OCDS-L2 Off-chip Trace and OCDS-L3 On-chip Trace 129
Simple Trace Control 129
Trace Break Signals (OCDS-L2) 129
Trace Examples 130
Troubleshooting for OCDS-L2 Trace 132
No Data in Trace.List Visible 132
Error Diagnosis 132
Searching for Errors 133
Error Messages 134
## History

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27-Jul-20</td>
<td>Update description of command: SYstem.Option MAPCACHE.</td>
</tr>
<tr>
<td>11-May-20</td>
<td>Updated chapter “Accessing Cached Memory Areas and Cache Inspection”.</td>
</tr>
<tr>
<td>17-Jul-19</td>
<td>New commands: SYstem.Option MAPCACHE, SYstem.Option SLOWRESET.</td>
</tr>
<tr>
<td>28-Jan-19</td>
<td>New command: SYstem.Option EndInitProtectionOverride.</td>
</tr>
<tr>
<td>16-Jan-18</td>
<td>Updated description of SYstem.Mode Down and SYstem.CONFIG DEBUGPORTTYPE to describe changed behavior.</td>
</tr>
<tr>
<td>16-Jan-18</td>
<td>Added descriptions of the new commands SYstem.Option DOWNMODE and SYstem.Option CBSACCEN&lt;x&gt;. Revised section “Parallel Usage of a 3rd-Party Tool”.</td>
</tr>
</tbody>
</table>
### Safety Precautions

<table>
<thead>
<tr>
<th>Caution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.</td>
</tr>
</tbody>
</table>

Recommendation for the software start:
- 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.

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

Brief Overview of Documents for New Users

Architecture-independent information:

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

Architecture-specific information:

- “Processor Architecture Manuals”: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:
- “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.
- “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” (app_tricore_ocds.pdf). Lauterbach also offers various “Adapter 16-pin 100 mil to 50 mil” (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

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 enables with the Trace License for TriCore ED (TriCore-MCDS). Note that only TriCore Emulation Devices (ED) provide an onchip trace buffer.
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 126.

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.

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”, page 22.

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**” (icd_quick_installation.pdf) on how to install the TRACE32 software:

- An installer is available for a complete TRACE32 installation under Windows. See “**MS Windows**” in ICD Quick Installation, page 24 (icd_quick_installation.pdf).
- For a complete installation of TRACE32 under Linux, see “**PC_LINUX**” in ICD Quick Installation, page 26 (icd_quick_installation.pdf).

## Configuration

### System Overview

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

**Diagram:**

- **PC**
  - Windows / Linux / Mac OS
- **USB Cable**
- **Target**
  - Debug Connector
  - **Wall Mount Power Supply**
  - **POWER DEBUG INTERFACE / USB 3**
  - **Debug Cable**
Related Documents

- “Application Note Debug Cable TriCore” (app_tricore_ocds.pdf): All about the TriCore debug cables and available accessories.
- “Debugger Basics - SMP Training” (training_debugger_smp.pdf): SMP debugging.
- “AURIX Trace Training” (training_aurix_trace.pdf): Training for AURIX 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).

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for all TriCore chips. Choose File menu > Search for Script.

In your TRACE32 installation directory there is a subdirectory ~/demo/tricore/ where you will find example scripts and demo software. The subdirectory ~/demo/tricore/ includes:

<table>
<thead>
<tr>
<th>Subdirectory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardware/</td>
<td>Ready-to-run debugging and flash programming demos for evaluation boards (e.g. Infineon TriBoard). <strong>Recommended for getting started!</strong></td>
</tr>
<tr>
<td>flash/</td>
<td>Binaries for target based programming and example declarations for internal flash.</td>
</tr>
<tr>
<td>etc/</td>
<td>Various examples, e.g. data trace, terminal application, PCP debugging, MCDS, …</td>
</tr>
<tr>
<td>kernel/</td>
<td>Various OS Awareness examples.</td>
</tr>
<tr>
<td>compiler/</td>
<td>Compiler examples.</td>
</tr>
<tr>
<td>fdx/</td>
<td>Example applications for the FDX feature.</td>
</tr>
</tbody>
</table>
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 16
- "Multicore Debugging (AURIX)", page 18
- "Access Classes", page 25
- "Flash", page 29
- "Onchip Triggers (TrOnchip Window)", page 31
- "Internal Break Bus (JTAG)", page 55
- "Suspending Peripherals", page 54
- "Assembler and Disassembler Options", page 45
- "Parallel Usage of a 3rd-Party Tool", page 46
- "Internal Break Bus (JTAG)", page 55
- "Troubleshooting", page 56
- "FAQ", page 53
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.

   - 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:

```plaintext
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.
Multicore Debugging (AURIX)

This chapter covers the following topics:

- “SMP Debugging - Quick Start”, page 18
- “AMP Debugging - Quick Start”, page 20
- “AMP vs. SMP”, page 22
- “Selecting the right AURIX CPU”, page 23
- “Understanding Multicore Startup by Application Code”, page 23
- “About Ambiguous Symbols”, page 24

SMP Debugging - Quick Start

1. Inform TRACE32 which chip you want to debug.

   \texttt{SYStem.CPU TC275TE}

2. Inform TRACE32 which cores form the SMP system.

   Select the cores you want to debug. Do this with the \texttt{CORE.ASSIGN} command before running the \texttt{SYStem.Up} command, e.g.:

   \texttt{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.

   \texttt{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 \texttt{CORE.select} allows to switch between cores (also visible and changeable via the “State Line” (ide_user.pdf)). Many commands also offer the option \texttt{/CORE <core>}, e.g.:

   \texttt{CORE.select 0. ; select core 0}
   \texttt{Register.view ; view registers of}
   \texttt{; currently selected core}
   \texttt{i.e. core 0}
   \texttt{Register.view /CORE 1. ; view registers of core}
   \texttt{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
ON CMD CORE2 GOSUB ; define command "CORE2"
    (LOCAL &params
     ENTRY %Line &params
     InterCom.execute &addressGUI2 &params ; execute command on
     RETURN ; remote GUI
    )
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 “File config.t32” (installation.pdf) contains:

```
CORE=<integer> ; e.g. CORE=1
IC=NETASSIST
PORT=<port_number> ; enable InterCom ; 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. For details, please refer to “Start Stop Synchronisation” in ICD Debugger User's Guide, page 41 (debugger_user.pdf).

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.
AMP vs. SMP

For multicore debugging and tracing, TRACE32 supports two 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)

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

<table>
<thead>
<tr>
<th>SMP</th>
<th>AMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>One GUI</td>
<td>Multiple GUIs</td>
</tr>
<tr>
<td>Controls all cores at the same time.</td>
<td>Control cores individually.</td>
</tr>
<tr>
<td>Cores are automatically synchronized, i.e. all cores will run at the same time and halt at the same time.</td>
<td>GUIs can be synchronized (Go/Step/Break/SystemMode), see SYnch / InterCom</td>
</tr>
<tr>
<td>Same onchip breakpoints on each core</td>
<td>Individual onchip breakpoints per core</td>
</tr>
<tr>
<td>Recommended for RTOS (esp. with dynamic core/thread assignment)</td>
<td>Required for heterogenous systems, e.g. debugging TriCore concurrently with GTM core</td>
</tr>
</tbody>
</table>

The debug concept is explained in www.lauterbach.com/publications/debugging_amp_smp_systems.pdf.

To get started with a new AURIX target, we recommend to start with our “Demo and Start-up Scripts”, page 13. (~~/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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYStem.CPU TC275TE-Astep</td>
<td>e.g. TC277TE-64F200S AA EES</td>
</tr>
<tr>
<td>SYStem.CPU TC275TF</td>
<td>e.g. TC275TF-64F200W BC EES</td>
</tr>
<tr>
<td>SYStem.CPU TC275TP</td>
<td>e.g. TC275TP-64F200W CA EES</td>
</tr>
<tr>
<td>SYStem.DETECT.CPU</td>
<td>Detect the device automatically</td>
</tr>
<tr>
<td>PRINT CHIP.STEPPing()</td>
<td>Stepping is available with a</td>
</tr>
<tr>
<td></td>
<td>function (for scripting)</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>e.g. PSPR on TC275TE</th>
<th>Global addressing</th>
<th>Core-local addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 0</td>
<td>P:0x70100000</td>
<td>P:0xC0000000</td>
</tr>
<tr>
<td>Core 1</td>
<td>P:0x60100000</td>
<td>P:0xC0000000</td>
</tr>
<tr>
<td>Core 2</td>
<td>P:0x50100000</td>
<td>P:0xC0000000</td>
</tr>
<tr>
<td>Summary</td>
<td>Unambiguous addresses</td>
<td>Ambiguous addresses (!)</td>
</tr>
</tbody>
</table>

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.
For background information about the term access class, see “TRACE32 Glossary” (glossary.pdf).

For TriCore, the following memory access classes are available:

<table>
<thead>
<tr>
<th>Memory Access Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Program (memory as seen by CPU’s instruction fetch)</td>
</tr>
<tr>
<td>D</td>
<td>Data</td>
</tr>
</tbody>
</table>
| 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 and SYStem.CpuAccess 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 41.
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.

<table>
<thead>
<tr>
<th>On-chip Breakpoints</th>
<th>Instruction Breakpoints</th>
<th>Read / Write Breakpoints</th>
<th>Data Breakpoints</th>
</tr>
</thead>
<tbody>
<tr>
<td>(up to AUDO-FG)</td>
<td>up to 4 single or</td>
<td>up to 4 single or</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>up to 2 ranges</td>
<td>up to 2 ranges</td>
<td></td>
</tr>
<tr>
<td>(AUDO-MAX, AURIX)</td>
<td>up to 8 address or</td>
<td>up to 8 single</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>up to 4 ranges</td>
<td>address or</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>up to 4 ranges</td>
<td></td>
</tr>
</tbody>
</table>

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.

| 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. Please refer to the “MCDS Breakpoints” in MCDS User’s Guide, page 26 (mcds_user.pdf) for details.
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” (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.

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 31.

- **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.:

```plaintext
FLASH.ReProgram 0xA0080000--
                    0xA009FFFF /Erase ; Activate flash region
                             ; 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:

```plaintext
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 already flashed application
```

### Flashing the Hardware Security Module (HSM)

When flashing the HSM, use the same precautions as described in “Flash and Debugging AURIX Devices”, page 30.

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 26, 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 50 (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:

![BMC.state window](image)

or using the following commands in PRACTICE:

```plaintext
BMC.CLOCK 100.Mhz ; Set the BMC Clock
BMC.ICNT.EVENT ON ; Set the Events
BMC.M1CNT.EVENT IP_DISPATCHSTALL
BMC.M2CNT.EVENT LS_DISPATCHSTALL
BMC.M3CNT.EVENT DMEMSTALL
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.

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.ATOBy 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.

```plaintext
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:

![BMC.state window](image)

A complex example recording the results in **BTrace** can be found in the demo folder under `~/demo/tricore/etc/bmc`. 
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 DATA_X_HIT ; count data cache / data buffer
          ; hits
BMC.M2CNT 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 M1CNT M2CNT M3CNT ; display a profile statistic
```

The SNOOPer.List and SNOOPer.PROfileChart.COUNTER commands allow to specify the counters you want to display. The benchmark counters of the TriCore are named as follows.

- ICNT, M1CNT, M2CNT, ... Performance counters of the core.
- BMC0, BMC1, ... Event counters of the MCDS.
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**

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
TrOnchip.BreakBus0.BreakOUT Enable
```

; set WATCH breakpoint
; route to pin

See the chapter “CPU specific TrOnchip Commands”, page 106 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**

AURIX devices can pulse up to eight different output pins, for details see chapter “Features and Restrictions”, page 39.

**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/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
```

![Image of TrOnchip interface]

**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 39.

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.:

```plaintext
; 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.:

![TriCore Debugger and Trace](image.png)

**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):

<table>
<thead>
<tr>
<th>Pin</th>
<th>Package Pin</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin0</td>
<td>P20.0</td>
<td>/BRKOUT</td>
</tr>
<tr>
<td>Pin1</td>
<td>P20.1</td>
<td>/BRKIN</td>
</tr>
<tr>
<td>Pin2</td>
<td>P21.6</td>
<td>(TDI)</td>
</tr>
<tr>
<td>Pin3</td>
<td>P21.7</td>
<td>(TDO, DAP2)</td>
</tr>
<tr>
<td>Pin4</td>
<td>P32.6</td>
<td></td>
</tr>
<tr>
<td>Pin5</td>
<td>P32.7</td>
<td></td>
</tr>
<tr>
<td>Pin6</td>
<td>P33.14</td>
<td></td>
</tr>
<tr>
<td>Pin7</td>
<td>P33.15</td>
<td></td>
</tr>
</tbody>
</table>

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 CPUs point of view.

The exact behavior differs between AUDO (TC1xxx) and AURIX (TC2xx and later) CPUs and is described below.

A similar issues 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”.
- Write 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 accessed:

- **cache evaluation method**: this method will evaluate the cache tags and content to reconstruct the cached memory area.
- **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, loads or stores might not become visible outside the 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.
Reading and Writing using Cache Evaluation Method

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 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.

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 55.
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 44

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.
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 send 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 55.
- 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” (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.

```
PowerDebug Module + Debug Cable 3rd party tool target
   Request          Grant          JTAG or DAP
              JTAG or DAP
```

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 ~/demo/tricore/etc/etk/*.cmm ; step through a demo ; script
  ```

  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

   ```
   SYStem.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.

![Break.List window](image)
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. 0x00000000--0x000FFFF 0x4000
   FLASH.CREATE ...
   ...
   FLASH.TARGET 0x40000000 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” (general_ref_f.pdf)
- “Software Breakpoints in FLASH” (norflash.pdf)

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

  ![Diagram showing CPU running after soft reset]

  This behavior is selected by using `SYStem.Option RESetBehavior RestoreGo` or `SYStem.Option RESetBehavior RunRestore`.

- **CPU halted after soft reset**

  ![Diagram showing CPU halted after soft reset]

  This behavior is selected by using `SYStem.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`. 
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 SYStem.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 SYStem.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 SYStem.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 SYStem.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.

- Configuration bits for every peripheral module that allow to configure if and how the module is suspended. For AURIX devices, this is typically done using the OCDS Control and Status Register <module>_OCS. For AUDO devices, this is typically done using the Clock Control Register <module>_CLC. These registers must be programmed by the target application, or using Per.Set commands in the debugger startup script. Example: Suspend GTP12 on a TC3xx chip

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 55), the suspend signal will be kept active.

Suspending the System Timers of TC3xx

In TC3xx CPUs the system timers 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:

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.
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 46.

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 43.

Internal Break Bus (JTAG)

The JTAG connector has two break lines (see `target connector`) 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.
Troubleshooting

In this section:

- SYStem.Up Errors
- Debugging Optimized Code
- Please also see the chapter “FAQ”, page 53.

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” (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 /SingleLineLineAdjacent
These options are explained in detail in chapter “Details on Generic Load Options” in General Commands Reference Guide D, page 88 (general_ref_d.pdf).

FAQ

Please refer to our Frequently Asked Questions page on the Lauterbach website.
Tracing

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 58
- “Serial Off-Chip Trace”, page 64
- “Parallel Off-Chip Trace”, page 64

Only basic information will be given here, details and advanced tasks are described in separate documents, see “Further Reading”, page 63.

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 16.
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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCDS.SOURCE TriCore FlowTrace ON</td>
<td>enable TriCore program flow trace</td>
</tr>
<tr>
<td>MCDS.TimeStamp TICKS</td>
<td>enable Ticks as timestamps</td>
</tr>
<tr>
<td>MCDS.CLOCK SYSTEM 20.0MHz</td>
<td>configure CPU clock for correct timestamp evaluation</td>
</tr>
</tbody>
</table>

3. Start and Stop Tracing

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>start tracing</td>
</tr>
<tr>
<td>Break</td>
<td>stop tracing</td>
</tr>
</tbody>
</table>

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" (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**.
Simple Trace Control

The TriCore OCDS-L3 trace supports the same features as TriCore OCDS-L2 trace. Additionally triggers and filters on data address and data value can be configured.

- Trace all
- Trace to
- Trace from
- Trace from to
- Trigger
- Enable

The simple triggers work on all TriCore fetches and write accesses. Bus accesses are not considered by the simple trigger.

| NOTE: | MCDS uses compression to efficiently use the limited amount of on-chip trace memory. TRACE32 requires a synchronization point to decode all consecutive trace messages. |

Examples

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

Example 1: Trace only the Function sieve()

Enable the trace as long as the function `sieve()` is executed. Execution in sub-functions is not recorded.

```
MCDS.view ; show MCDS setup window
MCDS.SOURCE TriCore FlowTrace ON ; enable TriCore program flow trace
Onchip.List ; show trace list window
Break.Set Var.RANGE(sieve) /Program /Onchip /TraceEnable ; enable the trace as long as function sieve() is executed
```
Example 2: Trace the Function `sieve()` and Sub-Functions

Trace the complete function `sieve()`, including execution in any sub-function.

```plaintext
MCDS.view ; show MCDS setup window
MCDS.SOURCE TriCore FlowTrace ON ; enable TriCore program flow trace
Onchip.List ; show trace list window
Break.Set sieve /Program /Onchip /TraceON ; enable trace on entering
Break.Set sYmbol.EXIT(sieve) ; function sieve()
Break.Set /Program /Onchip /TraceOFF ; disable trace on leaving
Break.Set sYmbol.EXIT(sieve) ; function sieve()
```

Example 3: Trace Until

Stop tracing when the end of function `sieve()` is reached, TriCore keeps running. `Onchip.TDelay` can be used to stop recording after a programmable period (percentage of the trace memory). See the `Trace.Trigger` command for more information.

```plaintext
MCDS.view ; show MCDS setup window
MCDS.SOURCE TriCore FlowTrace ON ; enable TriCore program flow trace
Onchip.List ; show trace list window
Break.Set sYmbol.EXIT(sieve) ; disable trace on leaving
Break.Set /Program /Onchip /TraceTrigger ; function sieve()
```

Example 4: Trace Write Accesses to a Variable

Trace all write accesses to variable `flags` with offset 0xC, program flow trace is disabled to save on-chip trace memory.

```plaintext
MCDS.view ; show MCDS setup window
MCDS.SOURCE TriCore FlowTrace OFF ; disable TriCore program flow
MCDS.SOURCE TriCore WriteAddr ON ; enable TriCore write address
MCDS.SOURCE TriCore WriteData ON ; and write data trace
Onchip.List ; show trace list window
Break.Set flags+0x0C /Write /Onchip /TraceEnable ; enable recording when TriCore writes to address flags+0x0C
```
Example 5: Trace Specific Write Accesses to a Variable while Executing a Function

Enable recording when TriCore writes 0x01 with an access width of 8 bits to address flags+0x0C. The code that triggered the write access is also recorded. Due to pipeline effects and internal delays the recorded code may not exactly match the write instruction.

```
MCDS.view ; show MCDS setup window
MCDS.SOURCE TriCore FlowTrace ON ; enable TriCore program flow trace
MCDS.SOURCE TriCore WriteAddr ON ; enable TriCore write address
MCDS.SOURCE TriCore WriteData ON ; and write data trace
Onchip.List ; show trace list window
Break.Set flags+0x0C /Write /Data.Byte 0x01 /Onchip /TraceEnable ; enable recording when TriCore writes 0x01 to address flags+0x0C
```

Example 6: Trace Specific Write Accesses to a Variable

Enable recording when TriCore writes 0x01 with an access width of 8 bits to address flags+0x0C while executing function sieve() (excluding sub-function). The code that triggered the write access is also recorded. Due to pipeline effects and internal delays the recorded code may not exactly match the write instruction.

```
MCDS.view ; show MCDS setup window
MCDS.SOURCE TriCore FlowTrace ON ; enable TriCore program flow trace
MCDS.SOURCE TriCore WriteAddr ON ; enable TriCore write address
MCDS.SOURCE TriCore WriteData ON ; and write data trace
Onchip.List ; show trace list window
Break.Set Var.RANGE(sieve) /MemoryWrite flags+0x0C /Data.Byte 0x01 /Onchip /TraceEnable ; enable recording when TriCore writes 0x01 to address flags+0x0C while executing function sieve()
```

Further Reading

For AURIX and similar devices, see “AURIX Trace Training” (training_aurix_trace.pdf).

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.

This trace is explained in the "AURIX Trace Training" (training_aurix_trace.pdf).

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 126.
This chapter explains TriCore specific commands only. All other commands are documented in the “General Commands” (general_ref_<x>.pdf) manuals.

**Analyzer.Mode PCP**

Select PCP trace

<table>
<thead>
<tr>
<th>Format:</th>
<th>Analyzer.Mode PCP</th>
</tr>
</thead>
</table>

Enables the OCDS-L2 PCP trace. The CPU trace port has to be switched manually to Peripheral Control Processor output. See the demo scripts for processor specific settings.

PCP debugging is described in “PCP Debugger Reference” (debugger_pcp.pdf).

**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” (general_ref_b.pdf).

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

Select counter for statistic analysis

Format: \texttt{BMC.SELect <counter>}

\texttt{<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 \texttt{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: \texttt{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 \texttt{Break.Set}.

BMC.<counter>.TRIGMODE

BMC trigger mode

Format: \texttt{BMC.<counter>.TRIGMODE \{OFF | TRACEOVERFLOW\}}

Enables/disables the generation of BenchMark Counter trace messages.

An example can be found under “\textit{Example: Record BMC Counters in the Trace}” in MCDS User’s Guide, page 53 (mcds_user.pdf).

BMC.<counter>.TRIGVAL

BMC trigger value

Format: \texttt{BMC.<counter>.TRIGVAL \{value\}}

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

<table>
<thead>
<tr>
<th>Format:</th>
<th><strong>SYStem.CONFIG.state</strong> [/&lt;tab&gt;]</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;tab&gt;:</td>
<td><strong>DebugPort</strong></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>&lt;tab&gt;</th>
<th>Opens the <strong>SYStem.CONFIG.state</strong> window on the specified tab: <strong>DebugPort</strong>, <strong>JTAG</strong>, and <strong>XCP</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DebugPort</strong></td>
<td>Lets you configure the electrical properties of the debug connection, such as the communication protocol or the used pinout.</td>
</tr>
<tr>
<td><strong>Jtag</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>XCP</strong></td>
<td>Lets you configure the XCP connection to your target. For descriptions of the commands on the <strong>XCP</strong> tab, see “<strong>XCP Debug Back-End</strong>” (backend_xcp.pdf).</td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH
### SYStem.CONFIG

Configure debugger according to target topology

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.CONFIG</th>
<th>&lt;parameter&gt; &lt;number_or_address&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SYStem.MultiCore</td>
<td>&lt;parameter&gt; &lt;number_or_address&gt; (deprecated)</td>
</tr>
</tbody>
</table>

- **<parameter>:** CORE <core>
- **<parameter>:** DRPRE <bits>
- **<parameter>:** DRPOST <bits>
- **<parameter>:** IRPRE <bits>
- **<parameter>:** IRPOST <bits>
- **<parameter>:** TAPState <state>
- **<parameter>:** TCKLevel <level>
- **<parameter>:** TriState [ON | OFF]
- **<parameter>:** 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 SYStem.Up. See Daisy-chain Example.

For some CPU selections (SYStem.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.

---

**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 SYStem.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.

---

Multicore debugging is not supported for the DEBUG INTERFACE (LA-7701).
DRPOST  (default: 0) <number> of TAPs in the JTAG chain between the TDI signal of the debugger and the core of interest. If each core in the system contributes only one TAP to the JTAG chain, DRPOST is the number of cores between the TDI signal of the debugger and the core of interest.

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

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

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
```
<table>
<thead>
<tr>
<th>TapStates</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Exit2-DR</td>
</tr>
<tr>
<td>1</td>
<td>Exit1-DR</td>
</tr>
<tr>
<td>2</td>
<td>Shift-DR</td>
</tr>
<tr>
<td>3</td>
<td>Pause-DR</td>
</tr>
<tr>
<td>4</td>
<td>Select-IR-Scan</td>
</tr>
<tr>
<td>5</td>
<td>Update-DR</td>
</tr>
<tr>
<td>6</td>
<td>Capture-DR</td>
</tr>
<tr>
<td>7</td>
<td>Select-DR-Scan</td>
</tr>
<tr>
<td>8</td>
<td>Exit2-IR</td>
</tr>
<tr>
<td>9</td>
<td>Exit1-IR</td>
</tr>
<tr>
<td>10</td>
<td>Shift-IR</td>
</tr>
<tr>
<td>11</td>
<td>Pause-IR</td>
</tr>
<tr>
<td>12</td>
<td>Run-Test/Idle</td>
</tr>
<tr>
<td>13</td>
<td>Update-IR</td>
</tr>
<tr>
<td>14</td>
<td>Capture-IR</td>
</tr>
<tr>
<td>15</td>
<td>Test-Logic-Reset</td>
</tr>
</tbody>
</table>
**SYStem.CONFIG.CORE**  
Assign core to TRACE32 instance

<table>
<thead>
<tr>
<th>Format: SYStem.CONFIG.CORE</th>
<th>SYStem.MultiCore.CORE (deprecated)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;core_index&gt;</td>
<td>&lt;chip_index&gt;</td>
</tr>
<tr>
<td>SYStem.MultiCore.CORE</td>
<td>&lt;core_index&gt; &lt;chip_index&gt;</td>
</tr>
<tr>
<td>&lt;chip_index&gt;:</td>
<td>1 ... i</td>
</tr>
<tr>
<td>&lt;core_index&gt;:</td>
<td>1 ... k</td>
</tr>
</tbody>
</table>

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.
**SYStem.CONFIG BreakPIN**

Define mapping of break pins

| 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:

<table>
<thead>
<tr>
<th>Break Bus 0</th>
<th>Break Bus 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PortPort</td>
<td>GPIO port pin</td>
</tr>
<tr>
<td>TdiPort</td>
<td>TDI pin</td>
</tr>
<tr>
<td>PortTDO</td>
<td>GPIO port pin</td>
</tr>
<tr>
<td>TdiTdo</td>
<td>TDI pin</td>
</tr>
</tbody>
</table>

©1989-2020 Lauterbach GmbH

TriCore Debugger and Trace
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 Interface** 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.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.Interface** before.

For target boards where a pull-up resistor on 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.
**NOTE:**
This command only has an effect in case the TRST line is connected to the DAPEN output pin. The ALTEN 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.

| Format: SYStem.CONFIG.DAP.USER0 [In | Out | Set <level>] |
| Format: SYStem.CONFIG.DAP.USER1 [In | Out | Set <level>] |
| **<level>:** 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.
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**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DebugCable0</td>
<td>Uses the debug cable directly connected to a PowerDebug hardware module.</td>
</tr>
<tr>
<td>DebugCableA</td>
<td>Uses the whisker connected to a CombiProbe.</td>
</tr>
</tbody>
</table>

**HostMCI mode**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfineonDAS0</td>
<td>Selects the Infineon DAS backend as interface. For a detailed description and examples, see “Debugging via Infineon DAS Server” (backend_das.pdf).</td>
</tr>
<tr>
<td>XCP0</td>
<td>Selects the XCP backend as interface. For a detailed description and examples, see “XCP Debug Back-End” (backend_xcp.pdf).</td>
</tr>
<tr>
<td>Unknown</td>
<td>No backend is selected. Debugging is not possible.</td>
</tr>
</tbody>
</table>

**SYStem.CONFIG.DEBUGPORTTYPE** Set debug cable interface mode

Format: SYStem.CONFIG.DEBUGPORTTYPE [JTAG | DAP2 | DAP3 | DXCPL]

SYStem.CONFIG.Interface [JTAG | DAP2] (deprecated)

Default: DAP2 (JTAG).

Configures the debug port type to be used by the debugger. Both CPU 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 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**
Disable external watchdog

Format: `SYStem.CONFIG EXTWDTDIS <option>`

<option>: High, Low, HIGHWHENSTOPPED, LOWWHENSTOPPED

Default: High.

Controls the WDTDIS pin of the Automotive Debug Cable. This option is only available if an Automotive Debug Cable is connected to the PowerDebug module.

- **High**
  - The WDTDIS pin is permanently driven high.

- **Low**
  - The WDTDIS pin is permanently driven low.
**SYStem.CONFIG PortSHaRing**  
Control sharing of debug port with other tool

| Format: | SYStem.CONFIG PortSHaRing [ON | OFF | Auto] |

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 46.
Format: **SYStem.CPU** <cpu>

Default: TC1797.

Selects the processor type.

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 your responsible Lauterbach representative.
**SYStem.JtagClock**

Set the JTAG frequency

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.JtagClock &lt;rate&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SYStem.BdmClock (deprecated)</td>
</tr>
</tbody>
</table>

| <rate>: | 10000. … 50000000. |

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.
SYStem.LOCK

Tristate the JTAG port

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

Default: OFF.

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

It must be ensured that the state of the 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.
SYStem.MemAccess

Run-time memory access (non-intrusive)

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. Instead intrusive accesses can be configured with SYStem.CpuAccess.
SYStem.Mode
Establish the communication with the CPU

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Mode &lt;mode&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mode&gt;:</td>
<td>Down</td>
</tr>
<tr>
<td></td>
<td>NoDebug</td>
</tr>
<tr>
<td></td>
<td>Prepare</td>
</tr>
<tr>
<td></td>
<td>Go</td>
</tr>
<tr>
<td></td>
<td>Attach</td>
</tr>
<tr>
<td></td>
<td>Up</td>
</tr>
<tr>
<td></td>
<td>StandBy</td>
</tr>
</tbody>
</table>

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 52.

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:**
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. **SYStem.JtagClock 500.0KHz** or higher is a save value for all CPUs.
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.
### SYStem.Option CBSACCEN\(<x>\>

**Cerberus access protection**

<table>
<thead>
<tr>
<th>Format</th>
<th>SYStem.Option CBSACCEN(&lt;x&gt;&gt; \text{TarGeT} \mid &lt;\text{value}&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x&gt;:)</td>
<td>0</td>
</tr>
<tr>
<td>(&lt;\text{value}&gt;:)</td>
<td>CerBeruS \mid &lt;\text{mask}&gt; \mid \text{CerBeruS} \mid &lt;\text{mask}&gt; \ldots)</td>
</tr>
</tbody>
</table>

Default: CerBeruS.

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

<table>
<thead>
<tr>
<th>TarGeT</th>
<th>TRACE32 will not touch the specified CBS_ACCEN(&lt;x&gt;&gt; register.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{value}&gt;)</td>
<td>The value is written to the register when the core is started.</td>
</tr>
</tbody>
</table>

The \(<\text{value}>\) is the OR-composition of the following:

<table>
<thead>
<tr>
<th>CerBeruS</th>
<th>Access for the debugger.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;\text{mask}&gt;)</td>
<td>Arbitrary mask. For valid masks, refer to the TriCore User Manual.</td>
</tr>
</tbody>
</table>

**Example 1:** Do not touch the CBS_ACCEN0 register.

```
SYStem.Option CBSACCEN0 TarGeT
```

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

```
SYStem.Option CBSACCEN0 CerBeruS
```

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

```
; \mid \text{------------- <value> -------------}\mid
; \mid \text{<mask>} \mid <mask> \mid
SYStem.Option CBSACCEn0 CerBeruS 0x000000001 0x00000002
; \mid \text{------------- <value> -------------}\mid
; \mid \text{<mask>} \mid <mask> \mid
SYStem.Option CBSACCEn0 0x000000001 CerBeruS 0x000000002
```
SYStem.Option DCFREEZE

Do not modify cache structure

Default: ON.

If enabled the data cache contents will not be touched 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 41.

SYStem.Option DCREAD

Control cache behavior of reads

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.

<table>
<thead>
<tr>
<th>DCREAD</th>
<th>CPU is</th>
<th>Read</th>
<th>Access class D: is like</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>stopped</td>
<td>through data cache</td>
<td>DC:</td>
</tr>
<tr>
<td>ON</td>
<td>running</td>
<td>physical memory</td>
<td>NC:</td>
</tr>
<tr>
<td>OFF</td>
<td>stopped or running</td>
<td>physical memory</td>
<td>NC:</td>
</tr>
</tbody>
</table>

More information is available in chapter “Accessing Cached Memory Areas and Cache Inspection”, page 41.
SYStem.Option DSYNC

Force data synchronization

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option DSYNC OFF</th>
<th>ReadWrite</th>
<th>Halt</th>
</tr>
</thead>
</table>

Default: OFF

Automatically executes a \texttt{dsync} instruction to make sure the CPU writes all data back to caches and memory.

- **OFF**: Feature disabled.
- **ReadWrite**: Execute \texttt{dsync} instruction on first read or write on that core.
- **Halt**: Execute \texttt{dsync} instruction when the CPU is halted.

\texttt{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 55).
**SYStem.Option DOWNMODE**

Behavior of SYStem.Mode Down

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option DOWNMODE TriState</th>
<th>ReSeT</th>
</tr>
</thead>
</table>

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, Data.List, 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.
**SYStem.Option DataTrace**  
Enable data tracing

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. See the example in ~/demo/tricore/etc/tracedata/ in your demo directory for more information on how to modify your application.

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/unsets ENIDIS for accesses to protected registers. |
| OFF | The ENIDIS bit unset for accesses to protected registers. |
| ON | The ENIDIS bit set for accesses to protected registers. |

**SYStem.Option ETK**  
Debugging together with ETK from ETAS

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

©1989-2020 Lauterbach GmbH
SYStem.Option HeartBeat

Bug fix to avoid FPI bus conflict

| Format: | SYStem.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 SYStem.Option DUALPORT and do not use the access class attribute E.

SYStem.Option HoldReset

Reset duration

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option HoldReset &lt;time&gt;</th>
</tr>
</thead>
</table>

Configure the minimum time the debugger holds the RESET line active on SYStem.Up or SYStem.Mode Go.
### SYStem.Option HSMRESTART

**Restart HSM on connect**

| 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 mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step.

On AUDO-Future devices, IMASKASM might not always work due to chip bug CPU_TC.115 (see the corresponding Infineon Errata sheets). Enable SYStem.Option STEPONCHIP concurrently as a workaround.
**SYStem.Option IMASKHLL**  
Disable interrupts while HLL single stepping

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

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step.

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

**SYStem.Option INTSTART**  
Start address of interrupt routines

| Format: | SYStem.Option INTSTART <address> (deprecated)  
Use the MCDS.Option eXception command group instead. |
|---------|-----------------------------------------------|

Sets the start address of the interrupt routines. The on-chip trace uses this together with SYStem.Option INTUSE to determine if an interrupt occurred.

**SYStem.Option INTUSE**  
Number of implemented interrupts

| Format: | SYStem.Option INTUSE <value> (deprecated)  
Use the MCDS.Option eXception command group instead. |
|---------|------------------------------------------------|

Sets the number of implemented interrupts. The on-chip trace uses this together with SYStem.Option INTSTART to determine if an interrupt occurred.
SYStem.Option JTAGENSEQ

Use JTAG initialization sequence

| Format: | SYStem.Option JTAGENSEQ [NONE | JTAG | <signature>] |
|----------------|-----------------------------------------------|

When JTAG is selected as type of the debug port (see SYStem.CONFIG.DEBUGPORTTYPE JTAG), the debugger must switch the debug port of the CPU to JTAG. This can either be done using TRST or a dedicated JTAG enable sequence.

- **NONE** (default): Do not send a sequence on SYStem.Mode Up and SYStem.Mode Go but use TRST. Use standard sequence on SYStem.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.

SYStem.Option KEYCODE

Set debug interface password

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option KEYCODE [&lt;pwd0&gt; &lt;pwd1&gt; … &lt;pwd7&gt;]</th>
</tr>
</thead>
</table>

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 will clear the password.

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 MAPCACHE**
Map cache automatically

| 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 42.
SYStem.Option OCDSELOW

Set OCDS line to low

| Format: | SYStem.Option OCDSELOW [ON | OFF] |
|---------|----------------------------------|

Default: OFF.

If enabled the nOCDS 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 “Debug Cables” (app_tricore_ocds.pdf) for more information on Debug Cables.

This option is ignored on TriCore devices where the nOCDS line is needed for enabling the OCDS, or when SYStem.Option ETK is not OFF.

SYStem.Option OVC

Enable OVERLAY memory access

| Format: | SYStem.Option OVC [ON | OFF] |
|---------|--------------------------------|
|         | SYStem.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.
SYStem.Option OVERLAY

**Format:**

```
SYStem.Option OVERLAY [ON | OFF | WithOVS]
```

Default: OFF.

**ON**

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

**OFF**

Disables support for code overlays.

**WithOVS**

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

**Example:**

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

Enable global peripheral suspend

| 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.
SYStem.Option PMILBFIX

Enable PMI line buffer invalidation workaround

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
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.
SYStem.Option PostResetDELAY  

Delay after RESET is released

Format:  

SYStem.Option PostResetDELAY <time>

Default: 100.us

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

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

\textbf{NOTE:}
\begin{itemize}
  \item Too short delays may result in debug port errors.
  \item Too long delays may result in failure to stop at the reset vector.
\end{itemize}

SYStem.Option ReadOnly  

Block all write accesses

Format:  

SYStem.Option ReadOnly [ON | OFF]

Default: OFF

This option blocks all write accesses to the target. This includes accesses from the debugger itself. Commands like \texttt{Go.direct} and \texttt{Break.direct} will not work while this option is active.
SYStem.Option RESetBehavior  

Set behavior when a reset occurs

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option RESetBehavior &lt;mode&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mode&gt;:</td>
<td>Halt \n</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option ResetDetection &lt;mode&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;mode&gt;:</td>
<td>DEFault \n</td>
</tr>
</tbody>
</table>

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

DEFault

A hard reset is detected on a low pulse of the `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.
SYStem.Option ResetMode

Select reset method

<table>
<thead>
<tr>
<th>Format:</th>
<th>SYStem.Option ResetMode &lt;method&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;method&gt;:</td>
<td>PORST</td>
</tr>
</tbody>
</table>

Configures the reset method used by SYStem.Up and SYStem.Mode Go.

- **PORST**: Power on Reset by asserting the \texttt{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.

**SYStem.Option RESetTMS**

State of TMS line at reset

| Format: | SYStem.Option RESetTMS [ON | OFF] |

Default: OFF.

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

**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 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.
SYstem.Option TB1766FIX

Bug fix for some TC1766 TriBoards

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 TRAPSTART

Start address of trap vectors

Format: SYstem.Option TRAPSTART <address> (deprecated)
Use the MCDS.Option eXception command group instead.

Sets the start address of the trap vectors. The on-chip trace uses this if a trap occurred.

SYstem.Option WDTFIX

Disables the watchdog on SYstem.Up

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.
SYStem.Option WDTSUS  
Link the watchdog timer to the suspend bus

| 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.
**TrOnchip.BreakIN.<target>**

Connect break <target> to BreakBus

| 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.

**TrOnchip.BreakOUT.<source>**

Connect break <source> to BreakBus

| Format: | TrOnchip.BreakOUT.<source> [BreakBus0 | BreakBus1]  
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TrOnchip.BreakOUT.HaLTEN [ENable</td>
<td>DISelective]</td>
</tr>
</tbody>
</table>
| <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.
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.

<table>
<thead>
<tr>
<th>TC11xx</th>
<th>AUDO-NG</th>
<th>AUDO-Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>TriCore</td>
<td>TriCore</td>
<td>TriCore</td>
</tr>
<tr>
<td>DMA</td>
<td>DMA</td>
<td>DMA</td>
</tr>
<tr>
<td>BCU / MLI0</td>
<td>S-BCU / MLI0</td>
<td>S-BCU / (MCDS)</td>
</tr>
<tr>
<td>MLI1</td>
<td>R-BCU / MLI1</td>
<td>MLI0</td>
</tr>
<tr>
<td></td>
<td>(MCDS)</td>
<td>MLI1</td>
</tr>
</tbody>
</table>

**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.
**TrOnchip.CONVert**

Adjust range breakpoint in on-chip resource

| 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

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrOnchip.CountX &lt;value&gt;</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrOnchip.CountY &lt;value&gt;</th>
</tr>
</thead>
</table>

See **TrOnchip.CountX**.
**TrOnchip.EXTernal**  
Configure TriCore break on BreakBus event

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

Default: OFF.

When enabled, TriCore will break on a 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

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrOnchip.PERSTOPOUT OFF</th>
<th>&lt;pin&gt;</th>
</tr>
</thead>
</table>

<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 /TGO0, Pin1 corresponds to /TGO1, etc.

---

**TrOnchip.RESet**  
Reset settings for the on-chip trigger unit

<table>
<thead>
<tr>
<th>Format:</th>
<th>TrOnchip.RESet</th>
</tr>
</thead>
</table>

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

Configure "TriCore" break on debug instruction

| 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 | DIISable] |
|------------------|---------------------------|

Default: DIISable.

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 | DIISable] |
|------------------|---------------------------|

Default: DIISable.

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).
TrOnchip.SusSwitch.Mode

Set suspend switch mode

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>

<target>: TriCore [ON | OFF]  
PMU [ON | OFF]  
FORCE [ENable | DISable]

Default: DISable / OFF.

Normally each peripheral has a Clock Register mod_CLC where it can be connected to the suspend bus. Some modules do not have such a register and 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.TCompress**  
Trace data compression

Format:

```
TrOnchip.TCompress [ON | OFF]
```

Default: ON.

The trace hardware supports trace data compression. Redundant trace cycles will be not recorded and displayed. For better trace memory usage it should be enabled.

The accuracy of timestamps will be reduced by this option. Set **Analyzer.Clock** to compensate these effects.

**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.

TrOnchip.TR0  

Specify trigger event 0

| 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.

```plaintext
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.

```plaintext
; 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:

\texttt{TrOnchip.state}

Shows the trigger on-chip window.

TrOnchip.WatchPin

Route core trigger to pin

Format:

\texttt{TrOnchip.WatchPin OFF} \ | \ \texttt{<pin>}

\texttt{<trigger>} \quad \texttt{WATCH} \ | \ \texttt{Alpha} \ | \ \texttt{Beta} \ | \ \texttt{Charly} \ | \ \texttt{Delta} \ | \ \texttt{Echo}

\texttt{<pin>} \quad \texttt{Pin0} \ | \ \texttt{Pin1} \ | \ \texttt{...} \ | \ \texttt{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 /TGO0, Pin1 corresponds to /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 "/TGO0". The pin will be active (low) while the program counter is inside func1 but not inside a called function.

\begin{verbatim}
Break.Set sYmbol.RANGE(func1) /Alpha
TrOnchip.WatchPin.Alpha Pin0
\end{verbatim}

An extended example can be found at \texttt{~/demo/tricore/etc/watchpins/aurix_watchpin.cmm}.
The on-chip comparator unit can react on different sources. Available for devices with an MCBS (up to AUDO-MAX family).

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR0</td>
<td>On-chip address comparator TR0</td>
</tr>
<tr>
<td>TR1</td>
<td>On-chip address comparator TR1</td>
</tr>
<tr>
<td>CR</td>
<td>Core Event: MTCR/ MFCR instruction is executed or a core SFR is modified.</td>
</tr>
<tr>
<td>EXT</td>
<td>External Break In pin is asserted.</td>
</tr>
</tbody>
</table>

See **TrOnchip.X** for further information.
JTAG Connector

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMS/DAP1</td>
<td>1</td>
<td>2</td>
<td>VTREF</td>
</tr>
<tr>
<td>TDO/DAP2/USER0</td>
<td>3</td>
<td>4</td>
<td>GND</td>
</tr>
<tr>
<td>CPUCLOCK</td>
<td>5</td>
<td>6</td>
<td>GND</td>
</tr>
<tr>
<td>TDI/DAPEN</td>
<td>7</td>
<td>8</td>
<td>RESET-</td>
</tr>
<tr>
<td>TRST/-DAPDIR</td>
<td>9</td>
<td>10</td>
<td>BRKOUT-</td>
</tr>
<tr>
<td>TCK/DAP0</td>
<td>11</td>
<td>12</td>
<td>GND</td>
</tr>
<tr>
<td>BRKIN-</td>
<td>13</td>
<td>14</td>
<td>OCDSE-/BREQ-</td>
</tr>
<tr>
<td>RESERVED/BGRNT-</td>
<td>15</td>
<td>16</td>
<td>GND</td>
</tr>
</tbody>
</table>

A standard 2 x 8 pin header (pin-to-pin spacing: 0.1 inch = 2.54 mm.) is required on the target.

- Do not connect the “reserved” pin.
- Do connect all GND pins and the N/C pin for shielding purpose, though they are connected on the debugger.
- VCCS is the processor power supply voltage. It is used to detect if target power is on and it is used to supply the output buffers of the debugger. That means the output voltage of the debugger signals (TMS, TDI, TCLK, /TRST, /BRKIN) depends directly on VCCS. VCCS can be 2.25 … 5.5 V. The output buffer takes about 2 mA.
- /RESET and /OCDS_E are controlled by an open drain driver. An external watchdog must be switched off if the In-Circuit Debugger is used.
- /BRKIN and /BRKOUT must be configured in MCBS (Multi Core Break Switch) for before they can be used.
- CPUCLOCK is optional. The CPU clock output has to be enabled by setting the appropriate register.

For an example design see the Infineon TriBoard schematics.

VIHmin = 2.0 V, VILmax = 0.8 V for the input pins TDO, CPUCLOCK, /BRKOUT.

When having multiple JTAG TAP controllers in the chain, make sure that the TriCore is the first device in the chain. It is not possible to have another TriCore, XC16x or XC2000 in the same chain.
See “Application Note Debug Cable TriCore” (app_tricore_ocds.pdf) for a description of
- The TriCore Debug Cables and their features
- The JTAG and DAP connectors
- Availably accessories: adapters, converters and cables

<table>
<thead>
<tr>
<th>Pins</th>
<th>Connection</th>
<th>Description</th>
<th>Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TMS</td>
<td>Test Mode Select</td>
<td>None.</td>
</tr>
<tr>
<td>2</td>
<td>VCCS</td>
<td>VCC Sense</td>
<td>Connect to Chip I/O voltage VCC.</td>
</tr>
<tr>
<td>3</td>
<td>TDO</td>
<td>Test Data Out</td>
<td>If there are multiple devices on the JTAG chain, connect TDO to the TDI signal of the next device in the chain.</td>
</tr>
<tr>
<td>4, 6, 12</td>
<td>GND</td>
<td>System Ground Plan</td>
<td>Connect to digital ground.</td>
</tr>
<tr>
<td>5</td>
<td>CPUCLOCK</td>
<td>CPU clock</td>
<td>Optional.</td>
</tr>
<tr>
<td>7</td>
<td>TDI</td>
<td>Test Data In</td>
<td>No other devices in the JTAG chain are allowed between the Debug Cable and the TriCore.</td>
</tr>
<tr>
<td>8</td>
<td>RESET</td>
<td>Reset</td>
<td>Connect to /PORST and connect /PORST to VCC via a 10 kΩ pull-up resistor. Do not connect to /TRST.</td>
</tr>
<tr>
<td>9</td>
<td>TRST</td>
<td>Test Reset</td>
<td>Connect to /TRST. Do not connect to /PORST.</td>
</tr>
<tr>
<td>10</td>
<td>BRKOUT</td>
<td>Break output</td>
<td>None.</td>
</tr>
<tr>
<td>11</td>
<td>TCLK</td>
<td>Test clock</td>
<td>None.</td>
</tr>
<tr>
<td>13</td>
<td>BRKIN</td>
<td>Break input</td>
<td>None.</td>
</tr>
<tr>
<td>14</td>
<td>OCDSE</td>
<td>OCDS enable</td>
<td>Required for older TriCore devices only, e.g. TC1765, TC1775 and TC19xx. Connect to VCC via a 10 kΩ pull-up resistor.</td>
</tr>
<tr>
<td>15</td>
<td>NC</td>
<td>Not Connected</td>
<td>Connect to Ground.</td>
</tr>
<tr>
<td>16</td>
<td>GND</td>
<td>KEY pin.</td>
<td>Connect to Ground.</td>
</tr>
</tbody>
</table>
Trace Connector

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

AMP 40 Connector

- 2x20 connector
- Pin-to-pin spacing: 50 mil = 0.05 inch = 1.27 mm
- Connector example:

<table>
<thead>
<tr>
<th>AMP/ Tyco 104550-5</th>
<th>standard</th>
</tr>
</thead>
</table>

Signal description:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUCLK</td>
<td>1</td>
<td>2</td>
<td>GND</td>
</tr>
<tr>
<td>N/C</td>
<td>3</td>
<td>4</td>
<td>EMUSTAT0</td>
</tr>
<tr>
<td>EMUSTAT1</td>
<td>5</td>
<td>6</td>
<td>EMUSTAT2</td>
</tr>
<tr>
<td>EMUSTAT3</td>
<td>7</td>
<td>8</td>
<td>EMUSTAT4</td>
</tr>
<tr>
<td>N/C</td>
<td>9</td>
<td>10</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>11</td>
<td>12</td>
<td>GND</td>
</tr>
<tr>
<td>N/C</td>
<td>13</td>
<td>14</td>
<td>EMUPC0</td>
</tr>
<tr>
<td>EMUPC1</td>
<td>15</td>
<td>16</td>
<td>EMUPC2</td>
</tr>
<tr>
<td>EMUPC3</td>
<td>17</td>
<td>18</td>
<td>EMUPC4</td>
</tr>
<tr>
<td>EMUPC5</td>
<td>19</td>
<td>20</td>
<td>EMUPC6</td>
</tr>
<tr>
<td>EMUPC7</td>
<td>21</td>
<td>22</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>23</td>
<td>24</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>25</td>
<td>26</td>
<td>GND</td>
</tr>
<tr>
<td>N/C</td>
<td>27</td>
<td>28</td>
<td>EMUBREAK0</td>
</tr>
<tr>
<td>EMUBREAK1</td>
<td>29</td>
<td>30</td>
<td>EMUBREAK2</td>
</tr>
<tr>
<td>N/C</td>
<td>31</td>
<td>32</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>33</td>
<td>34</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>35</td>
<td>36</td>
<td>VCC</td>
</tr>
<tr>
<td>N/C</td>
<td>37</td>
<td>38</td>
<td>N/C</td>
</tr>
<tr>
<td>N/C</td>
<td>39</td>
<td>40</td>
<td>GND</td>
</tr>
</tbody>
</table>
### ERF8 22-pin Power.org Connector

- 2x11 edge rate socket strip connector
- Pin-to-pin spacing: 31.5 mil = 0.031 inch = 0.8 mm
- Connector example:

<table>
<thead>
<tr>
<th>Samtec ASP-137969-01</th>
<th>ASP for Power.org connector (base type is ERF8-DV)</th>
</tr>
</thead>
</table>


**Signal description:**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RXP0</td>
<td>1</td>
<td>2</td>
<td>VREF</td>
</tr>
<tr>
<td>RXN0</td>
<td>3</td>
<td>4</td>
<td>TCK</td>
</tr>
<tr>
<td>GND</td>
<td>5</td>
<td>6</td>
<td>TMS</td>
</tr>
<tr>
<td>RXP1</td>
<td>7</td>
<td>8</td>
<td>TDI</td>
</tr>
<tr>
<td>RXN1</td>
<td>9</td>
<td>10</td>
<td>TDO</td>
</tr>
<tr>
<td>GND</td>
<td>11</td>
<td>12</td>
<td>TRST-</td>
</tr>
<tr>
<td>RXP2</td>
<td>13</td>
<td>14</td>
<td>CLKP</td>
</tr>
<tr>
<td>RXN2</td>
<td>15</td>
<td>16</td>
<td>CLKN</td>
</tr>
<tr>
<td>GND</td>
<td>17</td>
<td>18</td>
<td>TRIG</td>
</tr>
<tr>
<td>RXP3</td>
<td>19</td>
<td>20</td>
<td>CRCERR</td>
</tr>
<tr>
<td>RXN3</td>
<td>21</td>
<td>22</td>
<td>RESET-</td>
</tr>
</tbody>
</table>

### Samtec 60 Connector

- 2x30 high-speed socket strip connector
- Pin-to-pin spacing: 19.7 mil = 0.0197 inch = 0.5 mm
Connector examples:

<table>
<thead>
<tr>
<th>Connector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samtec QSH-030-01-L-D-A</td>
<td>standard</td>
</tr>
<tr>
<td>Samtec QSH-030-01-L-D-A-K</td>
<td>with pick &amp; place pad</td>
</tr>
<tr>
<td>Samtec QSH-030-01-L-D-A-K-TR</td>
<td>with pick &amp; place pad, tape &amp; reel</td>
</tr>
</tbody>
</table>

Signal description:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Pin</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET-</td>
<td>1</td>
<td>2</td>
<td>BREAK2</td>
<td></td>
</tr>
<tr>
<td>BRKIN-</td>
<td>3</td>
<td>4</td>
<td>BREAK1</td>
<td></td>
</tr>
<tr>
<td>TRST-</td>
<td>5</td>
<td>6</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>7</td>
<td>8</td>
<td>VCCIO/OCDS1</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>9</td>
<td>10</td>
<td>BREAK0</td>
<td></td>
</tr>
<tr>
<td>OCDS1EN-</td>
<td>11</td>
<td>12</td>
<td>STAT4</td>
<td></td>
</tr>
<tr>
<td>TDI</td>
<td>13</td>
<td>14</td>
<td>STAT3</td>
<td></td>
</tr>
<tr>
<td>TCK</td>
<td>15</td>
<td>16</td>
<td>STAT2</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>17</td>
<td>18</td>
<td>STAT1</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>19</td>
<td>20</td>
<td>STAT0</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>21</td>
<td>22</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>23</td>
<td>24</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>25</td>
<td>26</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>BRKOUT-</td>
<td>27</td>
<td>28</td>
<td>TDO</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>29</td>
<td>30</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>31</td>
<td>32</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>33</td>
<td>34</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>35</td>
<td>36</td>
<td>EMUPC7</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>37</td>
<td>38</td>
<td>EMUPC6</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>39</td>
<td>40</td>
<td>EMUPC5</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>41</td>
<td>42</td>
<td>EMUPC4</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>43</td>
<td>44</td>
<td>TMS</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>45</td>
<td>46</td>
<td>N/C</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>47</td>
<td>48</td>
<td>EMUPC3</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>49</td>
<td>50</td>
<td>EMUPC2</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>51</td>
<td>52</td>
<td>EMUPC1</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>53</td>
<td>54</td>
<td>EMUPC0</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>55</td>
<td>56</td>
<td>VCCIO/OCDS2</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>57</td>
<td>58</td>
<td>OCDS2EN-</td>
<td></td>
</tr>
<tr>
<td>N/C</td>
<td>59</td>
<td>60</td>
<td>EMUCLK</td>
<td></td>
</tr>
</tbody>
</table>
### Technical Data for Debugger

#### Operation Voltage

<table>
<thead>
<tr>
<th>Adapter</th>
<th>OrderNo</th>
<th>Voltage Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debugger for TriCore Automotive PRO</td>
<td>LA-2707</td>
<td>0.9 .. 5.5 V</td>
</tr>
<tr>
<td>Debug-Bundle MPC5xxx/TriCore Automotive PRO</td>
<td>LA-2713</td>
<td>1.6 .. 5.5 V</td>
</tr>
<tr>
<td>Debug-Bundle TriCore/RH850 Automotive PRO</td>
<td>LA-2714</td>
<td>1.6 .. 5.5 V</td>
</tr>
<tr>
<td>Debugger for TriCore Automotive</td>
<td>LA-3737</td>
<td>1.6 .. 5.5 V</td>
</tr>
<tr>
<td>Debugger-Bundle MPC5xxx/TriCore Automotive</td>
<td>LA-3738</td>
<td>1.6 .. 5.5 V</td>
</tr>
<tr>
<td>Debugger-Bundle TriCore/RH850 Automotive</td>
<td>LA-3753</td>
<td>1.6 .. 5.5 V</td>
</tr>
</tbody>
</table>

#### Mechanical Dimensions

<table>
<thead>
<tr>
<th></th>
<th>OCDS Uni-Dir Debug Cable V0</th>
<th>Bidirectional Cables and Automotive Debug Cables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length:</td>
<td>4.2 cm</td>
<td>5.5 cm</td>
</tr>
<tr>
<td>Width:</td>
<td>5.2 cm</td>
<td>6.25 cm</td>
</tr>
<tr>
<td>Height:</td>
<td>1.6 cm</td>
<td>1.7 cm</td>
</tr>
<tr>
<td>Length of cable:</td>
<td>approx. 37.5 cm</td>
<td>approx. 37.8 cm</td>
</tr>
</tbody>
</table>

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).
## Operation Frequency

<table>
<thead>
<tr>
<th>Module</th>
<th>CPU</th>
<th>TRACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA-7928</td>
<td>PXB4260</td>
<td>130.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC10GP</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1100</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1115</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1130</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1161</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1162</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1163</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1164</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1165</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1166</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC11IA</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC11IB</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC11IC</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1762</td>
<td></td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1764</td>
<td></td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1765</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1766</td>
<td></td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1766ED</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1775</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1792</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1796</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1796ED</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1796L</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1910</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1912</td>
<td>180.0 MHz</td>
</tr>
<tr>
<td>LA-7928</td>
<td>TC1920</td>
<td>180.0 MHz</td>
</tr>
</tbody>
</table>

## Operation Voltage

<table>
<thead>
<tr>
<th>Adapter</th>
<th>OrderNo</th>
<th>Voltage Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessor TriCore OCDS-L2 5V Samtec 60</td>
<td>LA-7879</td>
<td>3.0 .. 5.5 V</td>
</tr>
<tr>
<td>Converter AMP 40 to Samtec 60</td>
<td>LA-7919</td>
<td>3.0 .. 5.5 V</td>
</tr>
<tr>
<td>Preprocessor TriCore OCDS-L2 3.3V Samtec 60</td>
<td>LA-7928</td>
<td>1.8 .. 3.3 V</td>
</tr>
</tbody>
</table>
Mechanical Dimensions

Dimension

LA-7928  PP-TRICORE-3.3V

TOP VIEW

CABLE

SIDES VIEW

ALL DIMENSIONS IN 1/1000 INCH

LA-7879  PP-TRICORE-5V

TOP VIEW

CABLE

SIDES VIEW

ALL DIMENSIONS IN 1/1000 INCH
This chapter covers the following topics:

- “Parallel Off-chip Trace - OCDS-L2 Flow Trace (Analyzer)”, page 126

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 58.

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

```plaintext
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

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

Recording is stopped when the TriCore halts, e.g. when a breakpoint was hit.

6. View the Results

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).

<table>
<thead>
<tr>
<th>Order number</th>
<th>LA-7879</th>
<th>LA-7879</th>
<th>LA-7928</th>
</tr>
</thead>
<tbody>
<tr>
<td>Housing</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>PCB version</td>
<td>MERF1-4</td>
<td>MERF5-6</td>
<td>MERF7-</td>
</tr>
<tr>
<td>Connector</td>
<td>AMP40</td>
<td>SAMTEC60</td>
<td>SAMTEC60</td>
</tr>
<tr>
<td>Counter</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Frequency</td>
<td>80 MHz</td>
<td>133 MHz</td>
<td>180 MHz</td>
</tr>
<tr>
<td>Termination</td>
<td>passive (clock)</td>
<td>passive (clock)</td>
<td>active 50Ohm Thevenin</td>
</tr>
<tr>
<td>Threshold</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Trace control</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>JTAG on board</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Trace depth</td>
<td>x2</td>
<td>x2</td>
<td>x4</td>
</tr>
<tr>
<td>Number of cable</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

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 “Trace Method Analyzer” (general_ref_a.pdf) for details.

This results in inaccurate execution times. Especially when TrOnchip.TCompress is enabled, 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.

©1989-2020 Lauterbach GmbH
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” (mcgs_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

---

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:

<table>
<thead>
<tr>
<th>event</th>
<th>TR0</th>
<th>TR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EMU]BREAK0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[EMU]BREAK1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>[EMU]BREAK2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

Example 5: Trace Enable

When address 0xA0001400 is reached, write a sample into the trace memory.
Troubleshooting for OCDS-L2 Trace

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 (ICRTC-support@lauterbach.com).

Error Diagnosis

Error messages are displayed in the upper left corner of the Trace.List window:

| NOTE: | Advanced trace analysis commands like Trace.STATistic.Func, Trace.STATistic.TASK display accurate results only if the trace recording works error-free. |

©1989-2020 Lauterbach GmbH
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:

   ![Trace Find window](image)

   Select **Expert** and enter “flowerror” in the item field. The item entry is not case sensitive. Use the Find First/Find Next button to jump to the next flowerror within the trace. Find All will open a separate window with a list of all flowerrors.

2. Using command **Trace.FindAll**, FLOWERROR

   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.
3. Using command **Trace.Chart.sYmbol**

This command will start a statistical analysis. An additional symbol (ERROR) is shown if errors where found.

![Image of TriCore Debugger and Trace]

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 (ICRTC-support@lauterbach.com).

- Setup/hold time violation.
  - 3 ns setup / 1 ns hold time needed.
  - sample point: falling edge.