

Qorivva MPC5xxx/SPC5xx Debugger and NEXUS Trace

Release 02.2025





MANUAL

Qorivva MPC5xxx/SPC5xx Debugger and NEXUS Trace

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents	
ICD In-Circuit Debugger	
Processor Architecture Manuals	
Qorivva MPC5xxx/SPC5xx	
Qorivva MPC5xxx/SPC5xx Debugger and NEXUS Trace	1
History	9
Introduction	10
Available Tools	10
JTAG/OnCE Debugger	10
On-chip Trace	11
High-Speed Serial Off-chip Trace (Aurora NEXUS)	11
Parallel Off-chip Trace (parallel NEXUS)	12
Co-Processor Debugging (eTPU/GTM/SPT)	12
Multicore Debugging	12
Software-only Debugging (HostMCI) via XCP	13
Software Installation	13
Hardware Installation	14
JTAG Debugger	14
Parallel Nexus Debugger and Trace	15
Aurora Nexus Debugger and Trace	16
ESD Protection Considerations	17
Demo and Start-up Scripts	17
Debug Cable / Nexus Adapter Versions and Detection	18
Brief Overview of Documents for New Users	19
Target Design Requirement/Recommendations	21
General (ICD Debugger)	21
Quick Start	22
Run Program from On-chip SRAM	22
Run Program from FLASH	24
Connect to Running Program (hot plug-in)	25
FAQ	26
Debugging	27

Breakpoints	27
Software Breakpoints	27
On-chip Breakpoints	27
Breakpoints on Program Addresses	28
Breakpoints on Data Addresses	29
Breakpoints on Data Access at Program Address	30
Breakpoints on Data Value	30
Counting Debug Events with Core Performance Monitor	31
Memory Access	32
Access Classes	32
Access Classes to Memory and Memory Mapped Resources	32
Access Classes to Other Addressable Core and Peripheral Resources	33
Cache Debugging Support	35
Memory Coherency	35
Memory Coherency During run-time Memory Access	35
Viewing Cache Contents	36
MESI States and Cache Status Flags	37
Using Cache Lines as SRAM Extension	37
Architecture-Specific Status Bar Information: Debug Status	38
Debug Status While Core Stopped	38
Debug Status While Running	39
Support for Peripheral Modules	40
Displaying Peripheral Module Registers	40
Peripheral Registers Modified by TRACE32	41
Debugging and Tracing Through Reset	42
Multicore Debugging	44
SMP Debugging	45
AMP Debugging	46
Watchdog Timer Support	47
e200 Core Watchdog (TCR/TSR)	47
On-chip Watchdog (SWT)	47
Chip External Watchdog	48
Censorship Unlock	49
Censorship unlock on MPC56XX and SPC56X processors	49
Censorship unlock on MPC57XX, SPC57X/SPC58X and S32R processors	49
Recovering a censored processor (MPC57XX, SPC57X/SPC58X and S32R)	51
Non-secure boot (S32R294)	53
Non-secure boot by script	53
Non-secure boot if fuses blown	53
Troubleshooting Debug	54
Tracing	55
e200 PCFIFO On-chip Trace	55
MPC57XX/SPC57X/SPC58X NEXUS On-chip Trace (trace-to-memory)	56

External Trace Ports (Parallel NEXUS/Aurora NEXUS)	57	
Basic Setup for Parallel Nexus	57	
Basic Setup for Aurora Nexus	58	
Tracing the Program Flow	58	
Tracing of Data (read/write) Transactions	59	
Example: Data Trace with Address Range	59	
Tracing of Context Switches	60	
Trace Context Switches using Data Trace Messaging (DTM)	60	
Trace Context Switch using Ownership Trace Messaging (OTM)	60	
Trace Based Run-time Measurement / Timestamping	61	
Trace Based Run-time Measurement for off-chip Parallel NEXUS	61	
Trace Based Run-time Measurement for off-chip Aurora NEXUS	62	
Trace Based Run-time Measurement for on-chip Trace / Trace-to-memory	62	
Correlation of the Trace Timestamp with Other Tool Timestamps	62	
Implications of Using the Processor Generated Timestamps	62	
Processors with on-chip timestamp support	63	
Trace Filtering and Triggering with Debug Events	63	
Overview	63	
Example: Selective Program Tracing	65	
Example: Event Controlled Program/Data Trace Start and End	66	
Example: Event Controlled Trace Recording	67	
Example: Event Controlled Trigger Signals	67	
Example: Event Counter	68	
Tracing Peripheral Modules / Bus Masters	68	
Example: Filter by Address Range	68	
Example: Event Controlled Trace Start and End	68	
Trace Filtering and Triggering Features Provided by TRACE32	69	
Troubleshooting Trace	69	
Tracing VLE or Mixed FLE/VLE Applications	69	
FLASH Programming Support	71	
FLASH Programming Scripts	71	
Requirements due to FLASH ECC Protection	73	
Programming the RCHW or Boot Header	74	
Programming the Shadow Row	74	
Programming Serial Boot Password and Censorship Word	76	
TEST / UTEST / OTP FLASH Programming	77	
Programming an OTP Sector	77	
Programming an UTEST Sector which is not set to OTP	78	
Brownout Depletion Recovery	79	
Troubleshooting FLASH	79	
Command Reference: SYStem Commands	81	
SYStem.BdmClock	Set BDM clock frequency	81
SYStem.CONFIG.state	Display target configuration	82

SYStem.CONFIG	Configure debugger according to target topology	83
SYStem.CONFIG.DEBUGPORTTYPE	Set debug cable interface mode	88
Hardware Requirements for cJTAG Operation		88
SYStem.CONFIG.EXTWDTDIS	Disable external watchdog	89
SYStem.CONFIG.PortSHaRing	Control sharing of debug port with other tool	90
SYStem.CPU	Select the target processor	90
SYStem.LOCK	Lock and tristate the debug port	91
SYStem.MemAccess	Select run-time memory access method	91
SYStem.Mode	Select operation mode	93
Command Reference: SYStem.Option Commands		94
SYStem.Option.BISTRUN	Debug with BIST enabled	94
SYStem.Option.CoreStandBy	On-the-fly breakpoint and trace setup	94
SYStem.Option.DCFREEZE	Data cache state frozen while core halted	94
SYStem.Option.DCREAD	Read from data cache	95
SYStem.Option.DISableResetEscalation	Control reset escalation disabling	95
SYStem.Option.DISableShortSequence	Short reset sequence handling	96
SYStem.Option.DisMode	Disassembler operation mode	96
SYStem.Option.DUALPORT	Implicitly use run-time memory access	97
SYStem.Option.FASTACCESS	Special operation mode for fast run control	98
SYStem.Option.FREEZE	Freeze system timers on debug events	98
SYStem.Option.HoldReset	Set reset hold time	99
SYStem.Option.ICFLUSH	Invalidate instruction cache before go and step	99
SYStem.Option.ICREAD	Read from instruction cache	99
SYStem.Option.IMASKASM	Disable interrupts while single stepping	100
SYStem.Option.IMASKHLL	Disable interrupts while HLL single stepping	100
SYStem.Option.KEYCODE	Inhibit censorship protection	100
SYStem.Option.LPMDebug	Enable low power mode debug handshake	102
SYStem.Option.LockStepDebug	Enable lock-step core register access	103
SYStem.Option.MMUSPACES	Separate address spaces by space IDs	103
SYStem.Option.NexusMemoryCoherency	Coherent NEXUS mem-access	104
SYStem.Option.NoDebugStop	Disable JTAG stop on debug events	105
SYStem.Option.NoJtagRdy	Do not evaluate JTAG_RDY signal	105
SYStem.Option.NOTRAP	Use brkpt instruction for software breakpoints	106
SYStem.Option.OVERLAY	Enable overlay support	107
SYStem.Option.PC	Set fetch address debug actions	107
SYStem.Option.RESetBehavior	Set behavior when target reset detected	108
SYStem.Option.ResBreak	Halt the core while reset asserted	108
SYStem.Option.ResetDetection	Configure reset detection method	109
SYStem.Option.ResetMode	Select reset mode for SYStem.Up	110
SYStem.Option.SLOWRESET	Relaxed reset timing	110
SYStem.Option.STEPSOFT	Use alternative method for ASM single step	111
SYStem.Option.TDOSElect	Select TDO source of lock step core pair	111
SYStem.Option.VECTORS	Specify interrupt vector table address	111

SYStem.Option.WaitBootRom	Wait for BootROM completion	112
SYStem.Option.WaitReset	Set reset wait time	112
SYStem.Option.WATCHDOG	Debug with software watchdog timer	114
Command Reference: MMU Commands		116
MMU.DUMP	Page wise display of MMU translation table	116
MMU.List	Compact display of MMU translation table	118
MMU.SCAN	Load MMU table from CPU	120
MMU.Set	Set an MMU TLB entry	122
Command Reference: BenchMarkCounter		123
BMC.<counter>.ATOB	Enable event triggered counter start and stop	123
BMC.<counter>.FREEZE	Freeze counter in certain core states	126
BMC.FREEZE	Freeze counters while core halted	127
BMC.Trace	Trace performance monitor events	127
Command Reference: TrOnchip		128
TrOnchip.CONVert	Adjust range breakpoint in on-chip resource	128
TrOnchip.EDBRAC0	Assign debug events to target software	129
TrOnchip.EVTEN	Enable EVTI and EVTO pins	130
TrOnchip.RESet	Reset on-chip trigger settings	131
TrOnchip.Set	Enable special on-chip breakpoints	131
TrOnchip.VarCONVert	Set single address breakpoint for scalar	132
TrOnchip.state	View on-chip trigger setup window	133
Command Reference: Onchip		134
Onchip.TBARange	Set on-chip trace buffer address range	134
Command Reference: NEXUS		135
NEXUS.BTM	Enable program trace messaging	135
NEXUS.CLIENT<x>.BUSSEL	Set NXMC target RAM	135
NEXUS.CLIENT<x>.MODE	Set data trace mode of nexus client	135
NEXUS.CLIENT<x>.SELECT	Select a nexus client for data tracing	136
NEXUS.CLIENT3.SPTACQMASTER	Trace individual SPT masters	136
NEXUS.CoreENable	Enable core tracing for dedicated cores in SMP	136
NEXUS.DDR	Enable NEXUS double data rate mode	137
NEXUS.DMADTM	Enable DMA data trace messaging	137
NEXUS.DTM	Enable data trace messaging	138
NEXUS.DTMARK	Data trace mark	138
NEXUS.DTMWhileHalted	Data trace messaging while core halted	139
NEXUS.DQM	Enable data acquisition messaging	139
NEXUS.FRAYDTM	Enable FlexRay data trace messaging	139
NEXUS.HTM	Enable branch history messaging	140
NEXUS.OFF	Switch the NEXUS trace port off	140
NEXUS.ON	Switch the NEXUS trace port on	141
NEXUS.OTM	Enable ownership trace messaging	142
NEXUS.PCRCONFIG	Configure NEXUS PCR for tracing	142

NEXUS.PINCR	Define DCI PINCR register value	143
NEXUS.PortMode	Set NEXUS trace port frequency	143
NEXUS.PortSize	Set trace port width	144
NEXUS.POTD	Periodic ownership trace disable	144
NEXUS.PTCM	Enable program trace correlation messages	145
NEXUS.PTMARK	Program trace mark	145
NEXUS.RefClock	Enable Aurora reference clock	146
NEXUS.Register	Display NEXUS trace control registers	146
NEXUS.RESet	Reset NEXUS trace port settings	146
NEXUS.RFMHISTBUGFIX	Double RFM workaround	146
NEXUS.SmartTrace	Enable smart trace analysis	147
NEXUS.Spen<messagetype>	Enable message suppression	147
NEXUS.STALL	Stall the program execution when FIFO full	147
NEXUS.state	Display NEXUS port configuration window	148
NEXUS.SupprTHReshold	Set fill level for message suppression	148
NEXUS.TimeStamps	Enable on-chip timestamp generation	148
NEXUS.WTM	Enable watchpoint messaging	149
Nexus specific TrOnchip Commands		150
TrOnchip.Alpha	Set special breakpoint function	150
TrOnchip.Beta	Set special breakpoint function	150
TrOnchip.Charly	Set special breakpoint function	151
TrOnchip.Delta	Set special breakpoint function	151
TrOnchip.DISable	Disable NEXUS trace register control	151
TrOnchip.Echo	Set special breakpoint function	151
TrOnchip.ENable	Enable NEXUS trace register control	152
TrOnchip.EVTI	Allow the EVTI signal to stop the program execution	152
TrOnchip.EVTO	Use EVTO signal for runtime measurement	152
TrOnchip.EXTErnal	Enable trace trigger input of NEXUS adapter	153
TrOnchip.Out0	Select OUT0 pin signal source	153
TrOnchip.Out1	Select OUT1 pin signal source	154
TrOnchip.TOOLIO2	Select TOOLIO2 pin signal source	155
TrOnchip.TRaceControl	Trace control with special debug events	156
Debug and Trace Connectors		157
14-pin JTAG/OnCE Connector (JTAG)		157
AUTO26 Connector (JTAG)		157
10-pin ECU14 Connector (with converter LA-3843)		158
38-pin Mictor Connector (NEXUS parallel)		158
50-pin SAMTEC ERF8 Connector (NEXUS parallel)		159
51-pin GlenAir / ROBUST Connector (NEXUS parallel)		160
34-pin SAMTEC ERF8 Connector (Aurora NEXUS)		161
Mechanical Dimensions		162
Technical Data		171

History

- 10-Oct-22 New command [SYStem.Mode.Prepare](#).
- 20-Jul-22 For the [MMU.SCAN ALL](#) command, CLEAR is now possible as an optional second parameter.
- 24-Jun-22 New command [SYStem.Option.WaitBootRom](#).

Introduction

This document describes the processor specific settings and features for TRACE32-ICD for the following CPU families:

- NXP/Freescale Qorivva MPC55XX, MPC56XX, MPC57XX and S32R (PowerPC series)
- STMicroelectronics SPC56X, SPC57X and SPC58X series

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

NOTE:

The processor specific information in this document is collected thoroughly from processor reference manuals, data sheets and other sources. Lauterbach can however not guarantee that the processor specific information provided in this document is correct. Please refer to the processor reference manual and/or manufacturer.

Processor specific information includes but is not limited to:

- existence of a processor
- number and types of cores
- availability of certain debug features on a processor or core
- existence and sizes of memory
- availability of on-chip and off-chip trace features

Available Tools

This chapter gives an overview over available Lauterbach TRACE32 tools for MPC5XXX/SPC5XX processors.

JTAG/OnCE Debugger

Debugging MPC5XXX/SPC5XX requires a Lauterbach Debug Cable together with a Lauterbach PowerDebug Module. The following debug cables are available:

- LA-3206: Debugger for MPC5xxx AUTO26 (PACK)
- LA-2708: Debugger for MPC5xxx Automotive PRO
- LA-3736: Debugger for MPC5xxx Automotive
- LA-7753: JTAG Debugger MPC5xxx/SPC5xx



The following debug modules are supported:

- LA-3500: POWER DEBUG INTERFACE / USB 3
- LA-3503 POWER DEBUG E40
- LA-3505: POWER DEBUG PRO
- LA-7699: POWER DEBUG II Ethernet
- LA-7708: POWER DEBUG INTERFACE / USB 2
- LA-7707/LA-7690: POWER TRACE / ETHERNET 256 / 512MB
- LA-7705: POWER DEBUG Ethernet

LA-7753 is additionally supported by:

- LA-7702: POWER DEBUG INTERFACE
- LA-7704: POWER DEBUG INTERFACE / USB

The DEBUG INTERFACE (LA-7701) does not support this processor series.

For a comparison of the Debug Cables see:

https://www.lauterbach.com/differences_between_standard_and_automotive_debug_cables.pdf

On-chip Trace

On-chip tracing requires no extra Lauterbach hardware, it can be configured and read out with a regular **JTAG/OnCE Debugger**. Depending on the on-chip trace module implemented in the processor, a trace license might or might not be required. See **e200 PCFIFO on-chip trace** and **MPC57XX/SPC57X NEXUS on-chip trace** for details.

High-Speed Serial Off-chip Trace (Aurora NEXUS)

Lauterbach offers an off-chip trace solution for processors with Aurora NEXUS trace port. Aurora is a high-speed serial interface defined by Xilinx.

Tracing requires the Aurora NEXUS Preprocessor for Qorivva MPC57xx/SPC5XX (LA-3911) and a POWER TRACE II / POWER TRACE III module. A POWERTRACE / ETHERNET module can be used with reduced speed and limited functionality.

See **Basic Setup for Aurora Nexus** for more information.



Parallel Off-chip Trace (parallel NEXUS)

The parallel NEXUS trace port can be used with a PowerTrace/Ethernet or PowerTrace II / PowerTrace III module and one of the following NEXUS adapters:

- LA-7630 NEXUS AutoFocus adapter:
up to 16 MDOs, I/O Voltage 1.0-5.2V, Trace clock up to 200 MHz SDR (up to 100 MHz in DDR)
Debug port sharing support, ext. Watchdog control, cJTAG support
- LA-7610 NEXUS Adapter MPC55XX (obsolete):
up to 12 MDOs, I/O Voltage 2.6-3.6V, Trace clock up to 120 MHz SDR.
Debug port sharing support, ext. Watchdog control
Not supported with PowerTrace III or newer
- LA-7612 NEXUS Adapter MPC551X (obsolete):
up to 8 MDOs, I/O Voltage 3V or 5V, Trace clock up to 110 MHz SDR.
Not supported with PowerTrace III or newer

See [Basic Setup for Parallel Nexus](#) for more information.

Co-Processor Debugging (eTPU/GTM/SPT)

Debugging the MPC5XXX coprocessors eTPU/eTPU2, GTM and SPT is included free of charge, i.e. there is no additional license required.

For details about coprocessor debugging, see the specific Processor Architecture Manuals:

- [“eTPU Debugger and Trace”](#) (debugger_etpu.pdf)
- [“GTM Debugger and Trace”](#) (debugger_gtm.pdf)

Multicore Debugging

Lauterbach offers multicore debugging and tracing solutions, which can be done in two different setups: Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing (AMP). For details see chapter [Multicore Debugging](#).

Concurrent debugging of multiple e200 cores requires a *License for Multicore Debugging (MULTICORE)*.

Software-only Debugging (HostMCI) via XCP

TRACE32 PowerView also supports debugging and tracing without using TRACE32 PowerTools hardware. The debug accesses are done via a 3rd party XCP slave. The following licenses are required to unlock this feature:

- LA-8892L: 1 User Floating License PPC Front-End
- LA-9012L: 1 User Floating License XCP MPC5xxx Debug Back-End
- LA-8902L: 1 User Floating License Multicore Debugging (optional)
- LA-9013L: 1 User Floating License XCP MPC5xxx Trace License (optional)

For more information see below documents:

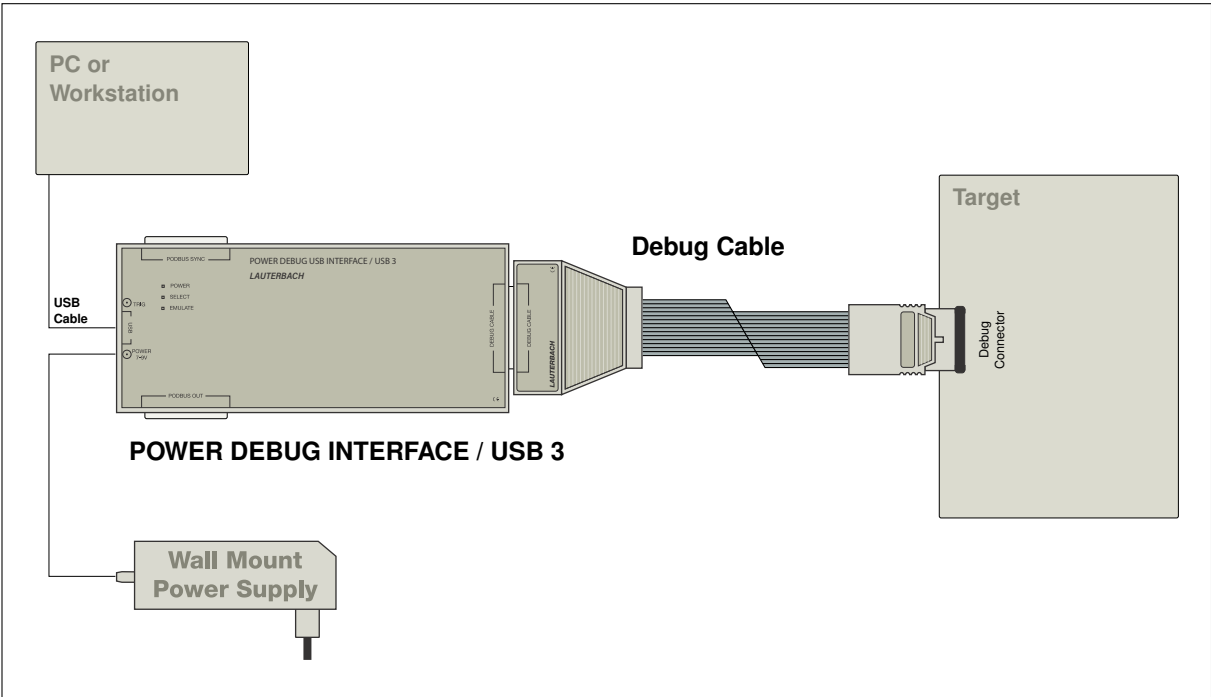
- [“XCP Debug Back-End”](#) (backend_xcp.pdf)
- [“Software-only Debugging \(Host MCI\)”](#) in T32Start, page 28 (app_t32start.pdf)

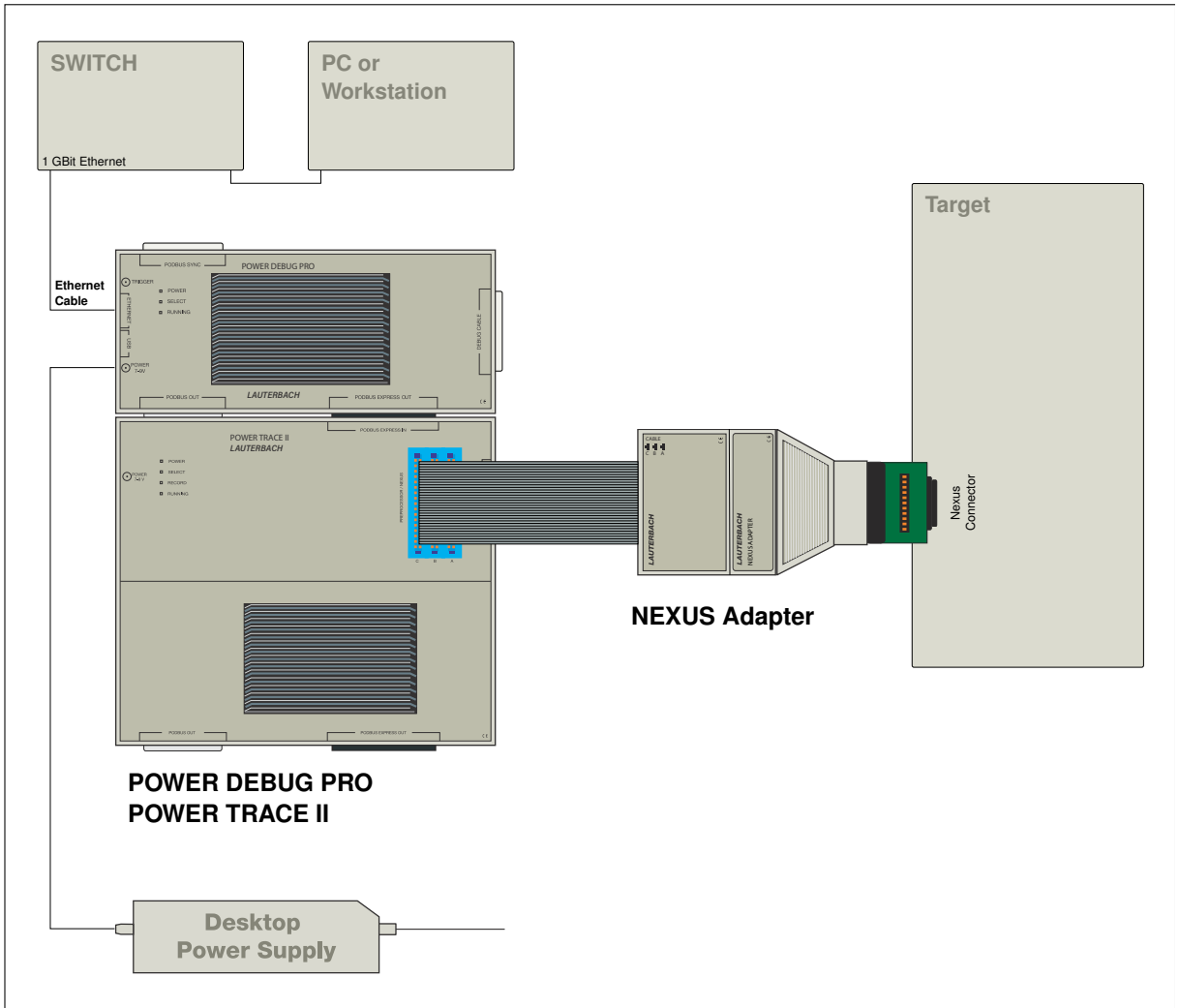
Software Installation

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

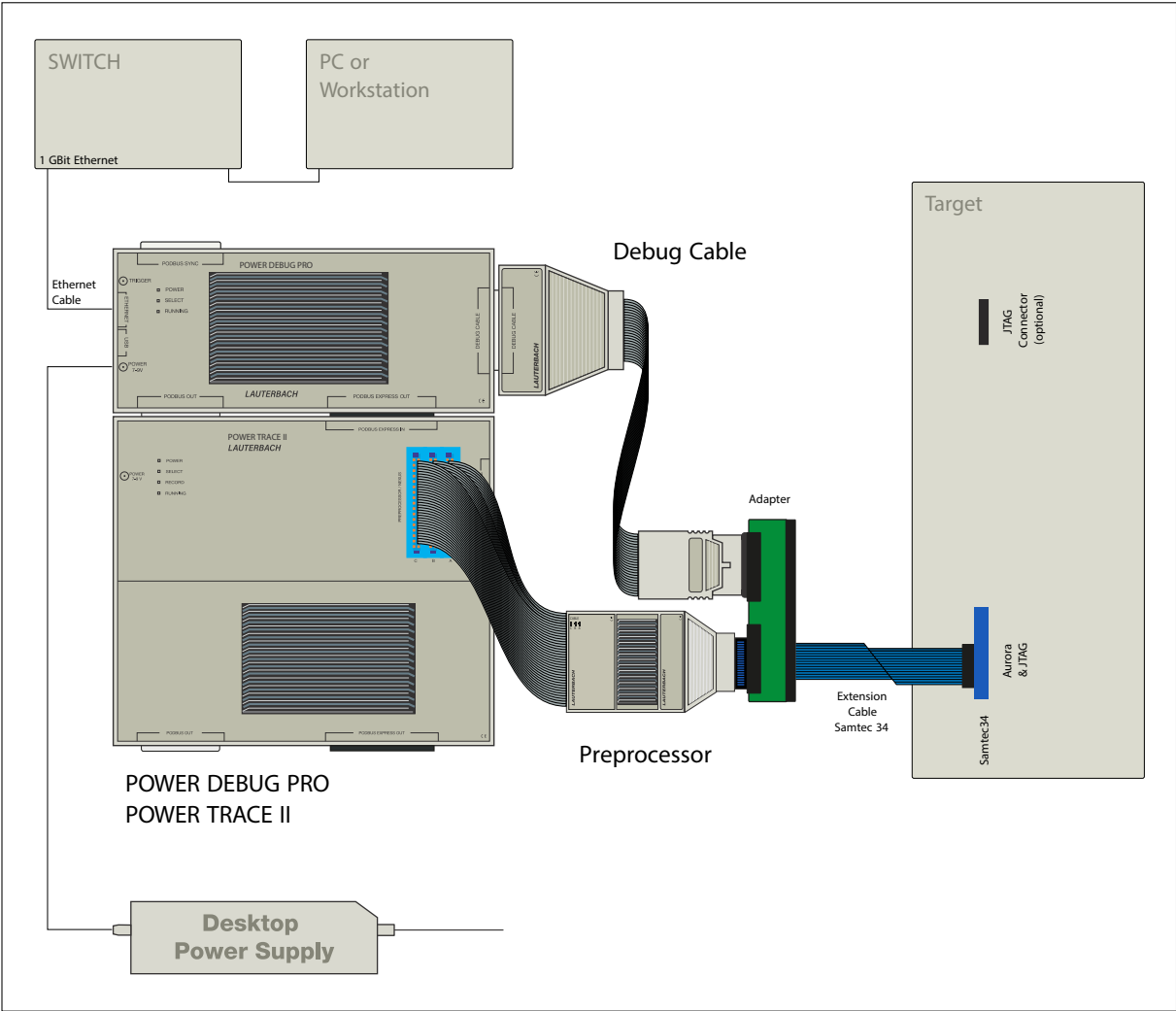
- An installer is available for a complete TRACE32 installation under Windows. See [“MS Windows”](#) in TRACE32 Installation Guide, page 21 (installation.pdf).
- For a complete installation of TRACE32 under Linux, see [“PC_LINUX”](#) in TRACE32 Installation Guide, page 23 (installation.pdf).

JTAG Debugger





When the NEXUS Adapter (for parallel NEXUS trace) is used, both debug and trace signals are connected through the NEXUS adapter and NEXUS connector on the target. Do not connect a debug cable in parallel to the NEXUS adapter.



WARNING:	<p>To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF.</p> <p>Recommendation for the software start:</p> <ol style="list-style-type: none">1. Disconnect the Debug Cable from the target while the target power is off.2. Connect the host system, the TRACE32 hardware and the Debug Cable.3. Power ON the TRACE32 hardware.4. Start the TRACE32 software to load the debugger firmware.5. Connect the Debug Cable to the target.6. Switch the target power ON.7. Configure your debugger e.g. via a start-up script. <p>Power down:</p> <ol style="list-style-type: none">1. Switch off the target power.2. Disconnect the Debug Cable from the target.3. Close the TRACE32 software.4. Power OFF the TRACE32 hardware.
-----------------	--

Demo and Start-up Scripts

Lauterbach provides ready-to-run start-up scripts for known hardware that is based on MPC5xxx.

To search for PRACTICE scripts, do one of the following in TRACE32 PowerView:

- Type at the command line: **WELCOME.SCRIPTS**
- or choose **File** menu > **Search for Script**.

You can now search the demo folder and its subdirectories for PRACTICE start-up scripts (*.cmm) and other demo software.

You can also manually navigate in the `~/demo/powerpc/` subfolder of the system directory of TRACE32. The demo scripts can be started through the menu **MPC5XXX > Tools > Start Demo**.

Debug Cable / Nexus Adapter Versions and Detection

The following table shows how to detect which JTAG debug cable or NEXUS adapter is connected:

Debug cable and/or Nexus adapter version	Condition in PRACTICE script language
LA-3206 LA-2708 (Automotive debug cable PRO)	<pre>PRINT ((ID.CABLE() &0xFFFE) == 0x4150) \ (ID.CABLE() == 0x4178)</pre>
LA-3736 (Automotive debug cable)	<pre>PRINT ID.CABLE() == 0x4155</pre>
LA-7753 rev. 1 (OnCE debug cable, JTAG only, no reset detection)	<pre>PRINT (ID.CABLE() &0xEFFF) == 0x604F</pre>
LA-7753 rev. 2 (OnCE debug cable, JTAG and cJTAG, supports reset detection)	<pre>PRINT ID.CABLE() == 0x3535</pre>
LA-7630 (Nexus Adapter, max 16 MDO / 2 MSEO, 1-5V, SDR and DDR)	<pre>PRINT POWERNEXUS() && (ID.CABLE() == 0x0002)</pre>
LA-7610 (Nexus Adapter, max 12 MDO / 2 MSEO 3.3V, SDR only)	<pre>PRINT POWERNEXUS() && (ID.CABLE() == 0x0100)</pre>
LA-7612 (Nexus Adapter, max 8 MDO / 1 MSEO, 5V, SDR only)	<pre>PRINT POWERNEXUS() && (ID.CABLE() == 0x0101)</pre>
LA-3911 (High speed serial prepro- cessor for Aurora NEXUS)	<pre>PRINT POWERTRACE() && !POWERNEXUS()</pre>

Required Debugger Software Versions

The table below shows the minimum required software version to work with certain debug hardware:

Debug Hardware	Minimum required software version
LA-3206 LA-2708 (Automotive debug cable PRO)	TRACE32 Release 2018.02
LA-3736 (Automotive debug cable)	TRACE32 Release 2012.08
LA-7753 rev. 2 (OnCE debug cable, JTAG and cJTAG, supports reset detection)	TRACE32 Release 2009.08

Brief Overview of Documents for New Users

Architecture-independent information:

- [“Debugger Tutorial”](#) (debugger_tutorial.pdf): Get familiar with the basic features of a TRACE32 debugger.
- [“General Commands”](#) (general_ref_<x>.pdf): Alphabetic list of debug commands.
- [“OS Awareness Manuals”](#) (rtos_<os>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

Architecture-specific information:

- [“Processor Architecture Manuals”](#): These manuals describe commands that are specific for the processor architecture supported by your Debug Cable. To access the manual for your processor architecture, proceed as follows:
 - Choose **Help** menu > **Processor Architecture Manual**.
- [“XCP Debug Back-End”](#) (backend_xcp.pdf): This manual describes how to debug a target over a 3rd-party tool using the XCP protocol.

Further information:

- **[“Training MPC5xxx/SPC5xx Nexus Tracing”](#)** (training_nexus_mpc5500.pdf): Training for the NEXUS trace
- **[“Onchip/NOR FLASH Programming User’s Guide”](#)** (norflash.pdf): Onchip FLASH and off-chip NOR FLASH programming.
- **[“Training Basic SMP Debugging”](#)** (training_debugger_smp.pdf): SMP debugging.
- **[“eTPU Debugger and Trace”](#)** (debugger_etpu.pdf): Debugging and tracing the eTPU/eTPU2.
- **[“GTM Debugger and Trace”](#)** (debugger_gtm.pdf): Debugging and tracing the Generic Timer Module (GTM).

General (ICD Debugger)

- Locate the **JTAG/OnCE or Trace connector** as close as possible to the processor to minimize the capacitive influence of the trace length and cross coupling of noise onto the JTAG signals. Don't put any capacitors (or RC combinations) on the JTAG lines.
- Connect TDI, TDO, TMS and TCK directly to the CPU. Buffers on the JTAG lines will add delays and will reduce the maximum possible JTAG frequency. If you need to use buffers, select ones with little delay. Most CPUs will support JTAG above 30 MHz, and you might want to use high frequencies for optimized download performance.
- Ensure that JTAG $\overline{\text{RESET}}$ is connected directly to the $\overline{\text{RESET}}$ of the processor. This will provide the ability for the debugger to drive and sense the status of $\overline{\text{RESET}}$. The target design should only drive $\overline{\text{RESET}}$ with open collector/open drain.
- For optimal operation, the debugger should be able to reset the target board completely (processor external peripherals, e.g. memory controllers) with $\overline{\text{RESET}}$.
- In order to start debugging right from reset, the debugger must be able to control CPU $\overline{\text{RESET}}$ and CPU $\overline{\text{TRST}}$ (JCOMP) independently. There are board design recommendations to tie CPU $\overline{\text{TRST}}$ (JCOMP) to CPU $\overline{\text{RESET}}$, but this recommendation is not suitable for JTAG debuggers.

Debug cable with blue ribbon cable	The T32 internal buffer/level shifter will be supplied via the VCCS pin. Therefore it is necessary to reduce the VCCS pull-up on the target board to a value smaller 10 Ω .
------------------------------------	--

Run Program from On-chip SRAM

Follow these steps to run a program from the on-chip SRAM:

1. Select the target processor, or use automatic CPU detection.

```
SYStem.CPU MPC5554  
; or  
SYStem.DETECT CPU
```

2. Multi-core processors: Select the core that starts running directly form reset

```
;MPC55XX/56XX: select core_0  
SYStem.CONFIG.CORE 1. 1.  
  
;MPC5746M: select core_2  
SYStem.CONFIG.CORE 3. 1.
```

3. Start debug session. Debugger resets processor and halts the core at the reset address.

```
SYStem.Up
```

4. Cores with MMU: After **SYStem.Up**, the core's MMU holds only a single TLB that maps the reset address. In order to run an application from SRAM, set up the required TLBs manually.

```
;initialize MPC55XX MMU (same as BAM)  
MMU.Set TLB1 0. 0x00000000 0x00000000 0x00000000  
MMU.Set TLB1 1. 0xC0000500 0xFFF0000A 0xFFF0003F  
MMU.Set TLB1 2. 0xC0000700 0x20000000 0x2000003F  
MMU.Set TLB1 3. 0xC0000400 0x40000000 0x4000003F  
MMU.Set TLB1 4. 0xC0000500 0xC3F00008 0xC3F0003F  
MMU.Set TLB1 5. 0xC0000700 0x00000000 0x0000003F
```

5. Cores with MMU: In order to run an application from SRAM, set up the required TLBs manually. For run-time memory access, the debugger requires a static translation table. As the core is halted and MMU set up, we can take the translation form the TLBs:

```
;copy core TLBs to debugger translation table  
MMU.SCAN TLB1  
  
;enable debugger based address translation  
TRANSlation.ON
```

6. MPC5XXX on-chip SRAM must be initialized (ECC) before usage.

```
Data.Set EA:0x40000000--0x4000FFFF%Quad 0x1122334455667788
```

7. Load the program.

```
Data.LOAD.Elf demo.elf           ; ELF specifies the format,  
                                   ; demo.elf is the file name
```

8. Run program, e.g. until function main.

```
Go main
```

9. Display ASM/HLL core at current instruction pointer

```
List
```

Run Program from FLASH

Follow these steps to program an application to flash and run it:

1. Prepare FLASH programming. mpc5xxx.cmm detects the target processor and calls the appropriate flash script

```
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY
```

2. Program application to FLASH. The command FLASH.ReProgram only erases and programs when required. The option **/NoClear** of the second Data.LOAD command keeps already loaded debug symbols.

```
;activate flash programming (unused sectors are erased)
FLASH.ReProgram ALL /Erase

;load file(s)
Data.LOAD.Elf project.x
Data.LOAD.S3 data.s3 /NoClear

;commit data to flash
FLASH.ReProgram off
```

3. The FLASH memory is now up-to-date. Reset the processor, so that the processor can load the RCHW form FLASH.

```
SYStem.Up
```

4. Cores with MMU: For run-time memory access, the debugger requires a static translation table. As the core's MMU is not set up right now, copying the translation from the core is not possible. As projects usually use 1:1 translation, a manual declaration can be performed.

```
;set up 1:1 address translation and enable
TRANSLation.Create 0x00000000--0xFFFFFFFF 0x00000000
TRANSLation.ON
```

5. Run program, e.g. until function main.

```
Go main
```

6. Display ASM/HLL core at current instruction pointer

```
List
```


Connect to Running Program (hot plug-in)

Follow these steps to attach the debugger to a running system:

1. Select the target processor, or use automatic CPU detection.

```
SYStem.CPU MPC5554  
; or  
SYStem.DETECT CPU
```

2. Load debug symbols.

```
Data.LOAD.ELF project.x /NoCODE
```

3. Start debug session without resetting core.

```
SYStem.Mode.Attach
```

4. Cores with MMU: For run-time memory access, the debugger requires a static translation table. As the core's MMU is not accessible while the core is running, copying the translation from the core is not possible. As projects usually use 1:1 translation, a manual declaration can be performed.

```
;set up 1:1 address translation and enable  
TRANSlation.Create 0x00000000--0xFFFFFFFF 0x00000000  
TRANSlation.ON
```

5. Observe variables or memory.

```
Var.View %E my_var your_var  
Data.Dump E:0x40000100
```

6. Set breakpoints or halt core.

```
Break.Set my_func /Onchip
```

```
Break
```

7. Display ASM/HLL core at current instruction pointer

```
List
```

FAQ

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

Breakpoints

There are two types of breakpoints available: Software breakpoints and on-chip breakpoints.

Software Breakpoints

To set a software breakpoint, before resuming the CPU, the debugger replaces the instruction at the breakpoint address with a **TRAP** instruction.

On-chip Breakpoints

To set breakpoints on code in read-only memory, only the on-chip instruction address breakpoints are available. With the command **MAP.BOnchip** <range> it is possible to declare memory address ranges for use with on-chip breakpoints to the debugger. The number of breakpoints is then limited by the number of available on-chip instruction address breakpoints.

- **On-chip breakpoints:** Total amount of available on-chip breakpoints.
- **Instruction address breakpoints:** Number of on-chip breakpoints that can be used to set Program breakpoints into ROM/FLASH/EEPROM.
- **Data address breakpoints:** Number of on-chip breakpoints that can be used as Read or Write breakpoints.
- **Data value breakpoint:** Number of on-chip data value 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

Core type:	On-chip Breakpoints	Instruction Address Breakpoints	Data Address Breakpoints	Data Value Breakpoints
e200z0 e200z0h	4 instruction 2 read/write no counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	none
e200z0Hn3	4 instruction 2 read/write 2 data value no counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	2 single breakpoints (associated with data address BPs)

Core type:	On-chip Breakpoints	Instruction Address Breakpoints	Data Address Breakpoints	Data Value Breakpoints
e200z1 e200z3 e200z6 e200z650 e200z750	4 instruction 2 read/write 2 counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	none
e200z335	4 instruction 2 read/write 2 data value 2 counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	2 single breakpoints (associated with data address BPs)
e200z446 e200z4d e200z760	8 instruction 2 read/write 2 data value 2 counters	8 single breakpoints -- or -- 2 breakpoint ranges and 4 single breakpoints	2 single breakpoints -- or -- 1 breakpoint range	2 single breakpoints (associated with data address BPs)
e200z210 e200z215 e200z225 e200z420 e200z425 e200z720 e200z4201 e200z4203 e200z4204 e200z4251 e200z7260	8 instruction 4 read/write 2 data value no counters	8 single breakpoints -- or -- 4 breakpoint ranges	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints (associated with data address BPs)

You can see the 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 a new on-chip breakpoint.

Breakpoints on Program Addresses

The debugger sets software and on-chip breakpoints to the effective address. If a breakpoint is set on a program address, the debugger will first try to set a software breakpoint. If writing the software breakpoint fails (translation error or bus error), then an on-chip breakpoint will be set instead. If a memory range must

not be written by the debugger, it can be declared for on-chip breakpoint usage using [MAP.BOnchip](#). Alternatively, it is also possible to force a single breakpoint to on-chip using the command [Break.Set](#) with option `/Onchip`:

```
Map.BOnchip 0xFFFFC000--0xFFFFFFFF ;use on-chip breakpoints in FLASH
Break.Set 0xFFFFF064                ;debugger sets on-chip breakpoint

Break.Set my_func1                   ;debugger sets on-chip or sw breakp.
Break.Set my_func1 /Onchip           ;debugger sets on-chip breakpoint
```

Two on-chip program address breakpoints can be combined to an address range:

```
Break.Set 0x00000000--0x00002000 /Onchip
Break.Set IVOR0_Handler--IVOR15_Handler /Onchip
```

Breakpoints can be configured to stop if the break event occurred a given number of times. If the core implements DBCNT (see [On-chip breakpoint table](#)), and on-chip breakpoint implementation is selected, the on-chip counter will be used.

```
;stop on the 20th call of function foo
Break.Set foo /Onchip /COUNT 20.
```

Breakpoints on Data Addresses

Data address breakpoints cause a debug event when a certain address or address range is read or written by the core. A data address breakpoint to a single address has a granularity of 1 byte.

```
Break.Set 0xC3F80004 /Read           ;break when core reads from 0xC3F80004
Break.Set 0xC3F80004 /Write          ;break when core writes to 0xC3F80004
Break.Set 0xC3F80004 /ReadWrite      ;break on read or write access

Break.Set 0xC3F80000--0xC3F80023 /Write ;break address range

Var.Break.Set counter /Write         ;break on variable write access
```

Equal to program address breakpoints, data address breakpoints can be configured to stop if the break event occurred a given number of times:

```
;stop on the 8th write to arrayindex
Break.Set arrayindex /Write /COUNT 20.
```

Data address breakpoint limitations:

1. The source of the data access (read and/or write) must be the core, as the data address breakpoints are part of the core. Any other accesses from on-chip or off-chip peripherals (DMA etc.) will not be recognized by the data address breakpoints.
2. The data being targeted must be qualified by an address in memory. It is not possible to set a data address breakpoint to GPR, SPR etc.

Breakpoints on Data Access at Program Address

A normal data access breakpoint as described above hits on all data accesses to the memory address or address range, independent of the program address which caused the access. It is also possible to set a data address breakpoint which only hits if the access is performed from a specified program address. The specified program address must be a load or store instruction.

```
;Break if the instruction at address 0x40001148 reads from variable count
Break.Set 0x40001148 /MemoryRead count

;Break if the instruction at address 0x40001148 writes to range
Break.Set 0x40001148 /MemoryWrite 0xFFFFF000--0xFFFFFFFF
```

The program address can also be an address range or a range of debug symbols:

```
;Break on all accesses to count from code of the address range
Break.Set 0x40000100--0x400001ff /MemoryReadWrite count

;Break if variable nMyIntVar is written by an interrupt handler
;(debug symbols IVORxx_Handler loaded from debug symbols)
Break.Set IVOR0_Handler--IVOR15_Handler /MemoryWrite nMyIntVar

;Break if variable nTestValue is written within function test_func
Break.Set sYmbol.RANGE(test_func) /MemoryWrite nTestValue

;Break if variable nTestValue is written outside of test_func
Break.Set sYmbol.RANGE(test_func) /EXclude /MemoryWrite nTestValue
```

Breakpoints on Data Value

Most e200 cores (see [On-chip breakpoint table](#)) implement two on-chip breakpoints on data value

For e200 cores without on-chip data value breakpoints, TRACE32 supports them by software emulation. When a data value breakpoint is set, the debugger will use one of the data address breakpoints. When the core hits that breakpoint, the target application will stop and the debugger will evaluate if the data value matches. If the value matches, the debugger will stop execution, if it does not match, the debugger will restart the application. Using software emulated data value breakpoints will cause the target application to slow down.

In case of the NEXUS debugger and trace, breakpoints on data value can be realized using the complex trigger unit. See [“Complex Trigger Unit for Nexus MPC5xxx”](#) (app_ctu_mpc5xxx.pdf).

Examples for setting data value breakpoints:

```
;Break when the value 0x1233 is written to the 16-bit word at 0x40000200
Break.Set 0x40000200 /Write /Data.Word 0x1233

;Break when a value not equal 0x98 is written to the 8-bit variable xval
Break.Set xval /Write /Data.Byte !0x98

;Break when decimal 32-bit value 4000 is written
;to variable count within function foo
Break.Set sYmbol.RANGE(foo) /MemoryWrite count /Data.Long 4000.
```

Counting Debug Events with Core Performance Monitor

The same debug events that are used for the above breakpoint examples can also be used as watchpoints, which can be used as input event to the core performance monitor. For more information about the core performance monitor, see [BMC](#).

The example below shows how to count the number of times, a certain function has been called:

```
;Set Alpha..Echo breakpoints to functions of interest
Break.Set my_func /Program /Onchip /Alpha
Break.Set othr_func /Program /Onchip /Beta

;Configure BMC (only CNT2 and CNT3 can count debug events)
BMC.state
BMC.CNT2.EVENT ALPHA
BMC.CNT3.EVENT BETA
```

Counting data accesses is similar. The following example calculates the percentage of variable writes with a certain value (compared to all writes to this variable):

```
;Set up debug events
Var.Break.Set xval /Write /Onchip /Alpha
Var.Break.Set xval /Write /Onchip /Data 0x98 /Beta

;Configure BMC (only CNT2 and CNT3 can count debug events)
BMC.state
BMC.CNT2.EVENT ALPHA
BMC.CNT3.EVENT BETA

;Show ratio
BMC.CNT3.RATIO X/CNT2
```

Access Classes

Access classes are used to specify how TRACE32 PowerView accesses memory, registers of peripheral modules, addressable core resources, coprocessor registers and the [TRACE32 Virtual Memory](#).

Addresses in TRACE32 PowerView consist of:

- An access class, which consists of one or more letters/numbers followed by a colon (:)
- A number that determines the actual address

Here are some examples:

Command:	Effect:
List.auto P:0x1000	Opens a List window displaying program memory
Data.dump D:0xFF800000 /LONG	Opens a DUMP window at data address 0xFF800000
Data.Set SPR:415. %Long 0x00003300	Write value 0x00003300 to the SPR IVOR15
PRINT Data.Long (ANC:0xFFF00100)	Print data value at physical address 0xFFF00100

Access Classes to Memory and Memory Mapped Resources

The following memory access classes are available:

Access Class	Description
P	Program (memory as seen by core's instruction fetch)
F	Program, disassembly shows std. PowerPC instructions
V	Program, disassembly shows VLE encoded instructions
D	Data (memory as seen by core's data access)
IC	L1 Instruction Cache (or L1 Unified cache)
DC	L1 Data Cache
L2	L2 Cache
NC	No Cache (access with caching inhibited)
EEC	Emulation memory in MPC57XX/SPC57X emulation devices

In addition to the access classes, there are access class attributes.

The following access class attributes are available:

Access Class Attributes	Description
E	Use real-time memory access. This attribute has no effect if SYStem.MemAccess is set to Disabled).
A	Given address is physical (bypass MMU)
U	TS (translation space) == 1 (user memory)
S	TS (translation space) == 0 (supervisor memory)

Examples of usage:

Command:	Effect:
List.auto SP:0x1000	Opens a List window displaying supervisor program memory
Data.Set ED:0x3330 0x4F	Write 0x4F to address 0x3330 using real-time memory access
Data.dump EEEC:0x0C000000	Opens dump window on emulation memory using real-time memory access

If an access class attribute is specified without an access class, TRACE32 PowerView will automatically add the default access class of the used command. For example, **List.auto** U:0x100 is complemented to **List.auto** UP:0x100.

Access Classes to Other Addressable Core and Peripheral Resources

The following access classes are used to access registers which are not mapped into the processor's memory address space.

Access Class	Description
SPR	Special Purpose Register (SPR) access
PMR	Performance Monitor Register (PMR) access
DCR	Device Control Register (DCR) access
TLB	Access to the core's TLB entries
DBG	NEXUS register and special debug register access

SPR, PMR and DCR registers are addressed by specifying the register number after the access class.

The access class TLB gives access to the TLB entries of the e200 core. The TLB contents are provided in the way they are represented in the MAS registers. The most significant byte is used to address the TLB table:

TLB access mask	Description
TLB:0x8100iiiM legacy access: TLB:0x0000iiiM	Access to TLB1 table (MMU). iii: TLB index M: Byte offset to TLB content as represented in MAS registers (0..3=MAS1, 4..7=MAS2, 8..11=MAS3, 12..15=unused)
TLB:0x8200iiiM	Access to TLB2 table (MPU). iii: TLB index M: Byte offset to TLB content as represented in MAS registers (0..3=MAS0, 4..7=MAS1, 8..11=MAS2, 12..15=MAS3)

The TLB access class is supplementary and allows reading TLBs as well as bit field modification of TLB entries. For general MMU/MPU setup, it is recommended to use the command [MMU.Set](#).

The access class DBG, which covers a wide variety of accesses, has a special encoding. The encoding as listed below is only valid for the MPC5XXX debugger.

DBG access mask	Description
DBG:0x01ttN0RR	Access to NEXUS registers of non-core NEXUS clients, e.g. NPC, NAR, NXDM, NXFR, NXSS, NXMC, SPU and GTMDI tt: TAP access command (ACCESS_AUX_...) N: NEXUS_ENABLE command (usually zero) RR: NEXUS register ID
DBG:0x03ttN0RR	Same as above, but for NEXUS clients on the Buddy Device
DBG:0x02tt0CRR	Access to eTPU NEXUS registers tt: TAP access command (ACCESS_AUX_...) C: eTPU client selection RR: eTPU register ID
DBG:0x04ttttttt	Access to DWPU tag RAM (32-bit wise) tttttt: Tag RAM address (one tag RAM access increment = 256-byte PD memory block)
DBG:0x00000004 ... DBG:0x0000007E	e200 core NEXUS register access (address = register index * 2)
DBG:0x400 (CDACNTL) DBG:0x401 (CDADATA)	e200 core cache debug register access

NOTE: The registers mapped through the DBG access class are automatically configured by the debugger. Manual changes are likely to disturb debugger/trace functionality and in most cases will be overwritten by the debugger. Use the NEXUS commands to configure tracing instead of directly writing to the NEXUS registers.

Memory Coherency

If a core is halted in debug mode, the debugger maintains cache coherency when the default access classes are used. The default access classes are written in bold letters in the table below. The other access classes allow the intentional modification certain memory without maintaining coherency.

The following table describes which memory will be accessed depending on the used access class:.

Access Class	Unified Cache	Memory (uncached)
D:	read(1)/updated	read(4)/updated
P: / F: / V:	read(2)/updated	read(4)/updated
DC:	read/updated	read(4)/not updated
IC:	read/updated	read(4)/not updated
NC:	no access	read/updated

Access Class	D-Cache	I-Cache	Memory (uncached)
D:	read(1)/updated	not updated	read(4)/updated
P: / F: / V:	not updated	read(2)/updated(3)	read(4)/updated
DC:	updated	no access	read(4)/not updated
IC:	no access	updated	read(4)/not updated
NC:	no access	no access	updated

- (1): if **SYStem.Option.DCREAD** is ON (default: ON)
- (2): if **SYStem.Option.ICREAD** is ON (default: OFF)
- (3): only if **SYStem.Option.ICFLUSH** is OFF (default: OFF)
- (4): reading from memory only if not found in cache

Memory Coherency During run-time Memory Access

Some e200 cores only support run-time access to uncached memory. The affected cores are e200 cores which implement data or unified cache and which support operating the cache in copy-back mode (e200z6, e200z650, e200z750, e200z760). For cores without data cache and cores that only support write-through (like most MPC57XX/SPC57X/SPC58X), there are no restrictions to run-time memory access.

If one of the affected cores is in use, one of the following changes to the core configuration can be made to get either read-only or both read/write access.

- To gain read-only access to all memory, configure the cache mode to write-through. This is done via the WM or DCWM field of the L1CSR register.
- To gain read-only access to certain memory spaces, configure or create a TLB entry for this address range and set the W (write-through) bit.
- To gain read/write access to all memory, the data/unified cache has to be disabled. This is done via the L1CSR register (cache enable bit).
- To gain read/write access to certain memory spaces, configure or create a TLB entry for this address range and set the I (caching inhibited) bit.

Please note that these changes will impact the processor performance. Global configuration settings (like done via L1CSR) have more impact than settings for small address ranges. Therefore it is recommended to control the access via TLB settings and keep the page sizes for read and/or write accesses as small as possible. For example, keep the stack memory range caching enabled, as the stack does usually not need to be accessed via run-time memory access.

Viewing Cache Contents

The cache contents can be viewed using the **CACHE.DUMP** command.

Cache	Command
L1 instruction cache L1 unified cache	CACHE.DUMP IC
L1 data cache	CACHE.DUMP DC

The meaning of the data fields in the **CACHE.DUMP** window is explained in the table below. Please note that an uninitialized cache will contain random data, therefore the data fields of the **CACHE.DUMP** window will show random values as well.

Data field	Meaning
address	Physical address of the cache line. The address is composed of cache tag and set index.
set, way	Set and way index of the cache
v, d	Status bits of the cache line v(alid), d(irty)
#	MESI state

I or II	Way locked. MPC55XX with unified cache: --: not locked I-: locked for instructions globally through L1CSR[WID] -D: locked for instructions globally through L1CSR[WID] ID: locked by lock bit in cache line MPC5XXX with I/D-Cache (harvard): -: not locked L: locked
sa ua	Supervisor (sa)/user (ua) access protection: rw: read-write ro: read-only na: no access MPC57XX/SPC57X/58X only.
lo	Lockout state. Cache lines with tag errors or data errors will have this lockout indicator set. MPC57XX/SPC57X/58X only.
u	LRU information. Shows which cache way will be replaced next. MPC57XX/SPC57X/58X only.
00 04 08 ...	Address offsets within cache line corresponding to the cached data
address (right field)	Debug symbol assigned to address

MESI States and Cache Status Flags

The data cache logic of Power Architecture cores is described as states of the MESI protocol. The debugger displays the cache state using the cache line status flags valid, dirty and shared. The debugger also displays additional status flags (e. g. locked) which can not be mapped to any of the MESI states.

State translation table:

MESI state	Flag
M (modified)	V(alid) && D(irty)
E (exclusive)	V(alid) && NOT D(irty)
S (shared)	V(alid) && S(hared)
I (invalid)	NOT V(alid)

Using Cache Lines as SRAM Extension

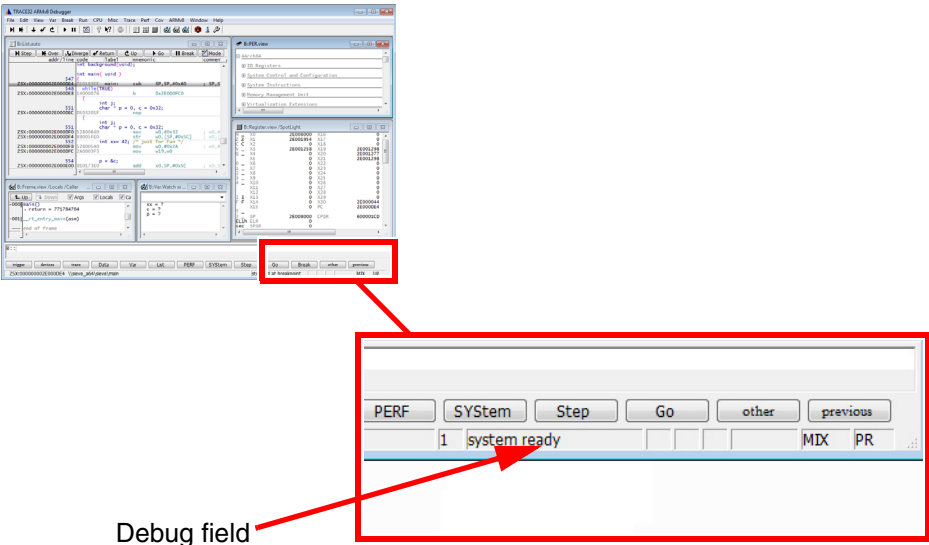
Some e200 cores (e200z6, e200z650, e200z750, e200z760) allow using locked cache lines as additional SRAM. The cache lines are enabled and locked to an unused address. In this case the debugger might fail to display the contents in cache (bus error) if the debugger is not configured / used as described below.

If the cache lines are used as data memory, ensure that **SYStem.Option.DCREAD** is set to ON (default).

If the cache lines are used as program memory (only possible for e200z6, e200z650), set **SYSystem.Option.ICREAD** to on. Program code in a locked cache line can only be modified using access class IC.

Architecture-Specific Status Bar Information: Debug Status

The debug status field in the status bar shows information about the state of the core or SoC. This chapter lists the architecture specific states. For a description of generic states, see “**Status Bar**” (ide_user.pdf).



Debug Status While Core Stopped

If the core was running previously, PowerView will usually display the stop reason. Some stop reasons may only be shown if the halt event could not be assigned to a breakpoint set in the **Break.List** window.

Stop reason	Description
stp by EVTI or BKPT	The core was stopped either by an EVTI event or by fetching the hard BKPT instruction. This can occur if TrOnchip.EVTI is ON, as in this case the debugger can in some cases not differentiate those events.
stopped by BKPT	The core was stopped by fetching a hard-coded BKPT instruction.
stopped by EVTI	The core was stopped by an EVTI trigger event.
stopped by dni	The core was stopped by a DNI debug event (caused by executing a DNI instruction).
stopped by pmi	The core was stopped by a Performance Monitor Interrupt debug event.

stopped by mpu	The core was stopped by a Memory Protection Unit debug event.
stopped by ude	The core was stopped by an Unconditional debug event.
stopped by interrupt	The core was stopped by an Interrupt Taken debug event, if TrOnchip.Set IRPT was set to ON.
stopped by return	The core was stopped by a Return debug event, if TrOnchip.Set RET was set to ON.
stopped by dacr	The core was stopped by a Data Address Compare Read debug event.
stopped by dacw	The core was stopped by a Data Address Compare Write debug event.
stopped by dcomp1	The core was stopped by Data Compare debug event #1.
stopped by dcomp2	The core was stopped by Data Compare debug event #2.
stopped by devt1	The core was stopped by an External Debug Event 1 debug event. Usually triggered by the break switch.
stopped by devt2	The core was stopped by an External Debug Event 2 debug event. Usually triggered by the break switch.
stopped by dcnt1	The core was stopped by an Debug Counter 1 debug event. Only for cores that have hardware support for counting debug events.
stopped by dcnt2	The core was stopped by an Debug Counter 2 debug event. Only for cores that have hardware support for counting debug events.

Debug Status While Running

If the core is running, PowerView shows the state with a green background. The state “running”, from the debugger point of view, means that the core is not halted in debug mode, so access to many core resources (esp. registers) is not possible. The state “running” does not necessarily imply that the core is executing instructions.

Run state	Description
running	Core running and not halted for debug.
running (reset)	The core is currently in reset. The debugger shows this state if OSR[RESET] is set.
running (chkstop)	The core is in checkstop state. Usually it can not be halted by the debugger.

running (wait)	The core is stalled and waiting for an interrupt, because the WAIT instruction was executed. The debugger shows this state if OSR[WAIT] is set.
running (stop)	The core is in STOP mode. The debugger shows this state if OSR[STOP] is set.
running (sleep)	The core is in HALT mode. The debugger shows this state if OSR[HALT] is set.
running (lpm-st/sl)	SYStem.Option.LPMDebug is set to PASSIVE or ACTIVE and the chip is in a low power-mode.
running (lpm-sleep)	SYStem.Option.LPMDebug is set to HANDSHAKE and the chip entered SLEEP mode.
running (lpm-stop)	SYStem.Option.LPMDebug is set to HANDSHAKE and the chip entered STOP mode.
running (lock-step)	The core is the checker core of a lock-step core pair. This core can not be halted by the debugger.

Support for Peripheral Modules

TRACE32 supports access to the memory mapped registers of all peripheral modules. The peripheral register description files (*.per, so-called PER-files) for the on-chip peripherals are included in TRACE32. PER files for recent processors are usually not included in updates, but are available upon request.

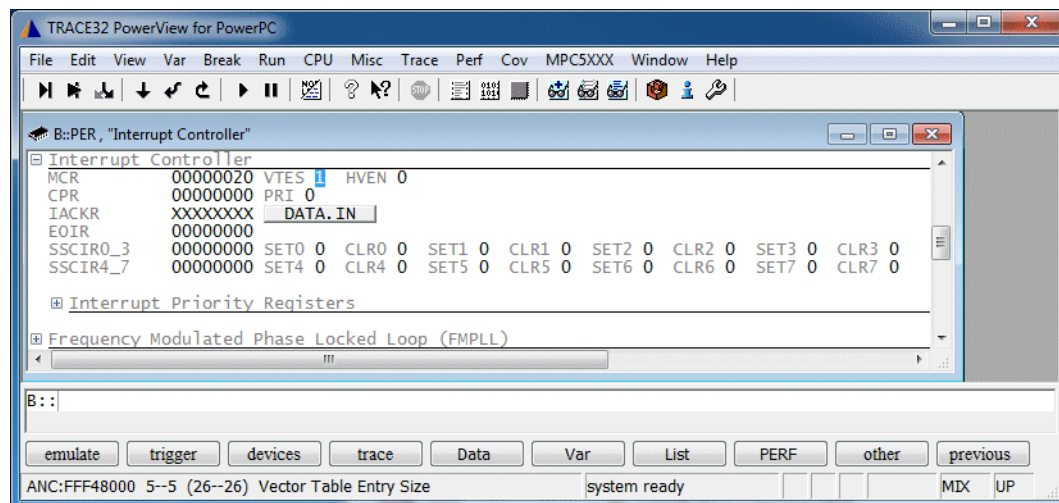
For external peripherals and/or custom peripherals, it is possible to create additional PER files with custom content. See “**Peripheral Files Programming**” (per_prog.pdf) for details.

Displaying Peripheral Module Registers

Open the peripheral registers view either using the command **PER.view**, or open the window by menu: CPU -> Peripherals. In order to show a certain module directly, open it by menu using MPC5XXX -> On-chip peripherals.

If a configuration register is selected in the **PER.view** window, the following information is displayed in the **Cursor** field of the TRACE32 status bar:

- **Access class** - here A for physical address and NC for no cache.
- Address - here 0xffff48000.
- Bit position (TRACE32 bit index) - here 5.
- Bit position (Power Architecture bit numbering) - here 26.
- Full name - here Vector Table Entry Size.



Cursor field

Peripheral Registers Modified by TRACE32

Some memory mapped registers of the on-chip peripherals have to be modified to allow proper debug control of the processor:

Register	Feature / Action	Dependencies
SWT[CR]	Watchdog. Must be disabled e.g. for FLASH programming.	SYSTEM.Option.WATCHDOG

RGM[FRET] RGM[DRET]	Reset escalation. If the application causes processor resets, reset escalation will cause that the processor disables itself after some resets until the next power cycle.	SYStem.Option.DISableResetEscalation
RGM[FESS]	Short reset sequence. If the processor is configured to perform the short sequence, peripherals and cores are not reset, which can cause a variety of debugging issues. E.g. FLASH programming can fail.	SYStem.Option.DISableShortSequence
SIU.PCR[] SIUL.PCR[]	Modify PCR registers so that their pin function is set to NEXUS signals (MDO, MSEO, MCKO)	NEXUS.PCRCONFIG

Debugging and Tracing Through Reset

Overview

In order to debug or trace through a reset, the debug and trace register configuration must be preserved when a reset occurs.

Depending on the used processor or processor version, both debug and trace registers will be preserved natively, or debug and/or trace registers will be loaded with reset values when the reset occurs.

If the core(s) on the target reset debug and/or trace register values when a reset occurs, TRACE32 offers a workaround to re-configure debug and trace registers. The workaround is enabled with the command [SYStem.Option.ResetDetection](#). The available reset detection mechanisms depend on the used debug cable or nexus adapter. See [SYStem.Option.ResetDetection](#) for details.

The debugger supports a number of different actions when a reset is detected, see [SYStem.Option.RESetBehavior](#) for details.

On some processors, the I/Os used for the NEXUS port are set to GPIO functionality by default. To enable trace-through-reset functionality, these processors also require [SYStem.Option.ResetDetection](#), together with [NEXUS.PCRCONFIG](#).

MPC563X, MPC564X, MPC567X, SPC563X, SPC564X and MPC57XX, SPC57XX

Most processors of the MPC56XX/SPC56XX and MPC57XX/SPC57XX series natively support debugging through reset, because the debug and trace registers are not cleared upon reset. Breakpoints and trace settings are not affected and are still in effect after a reset.

If the debugger is operated together with another tool using the same JTAG port (e.g. calibration tool), the other tool might disable the trace port when a reset occurs. Using **SYStem.Option.ResetDetection** here will work around this behavior and re-enable trace-through-reset functionality.

MPC555x, MPC553x, MPC556x, MPC560x, SPC560x

The cores of this processors will reset all on-chip debug and trace registers upon reset, so **SYStem.Option.ResetDetection** is required to debug / trace through resets.

When reset is detected, the debugger will then reconfigure all on-chip breakpoints, debug and trace registers before the CPU starts executing code from the reset address. Reconfiguring takes some time, so there will be a delay from releasing reset until commands are executed.

On-chip breakpoints can be used to stop the core after reset, e.g. at the start of the user program. If set to the reset address, the CPU can also be halted immediately after reset. Example:

```
SYStem.Option.RD_RSTINOUT          ; listen for reset on RSTOUT  
  
Break.Set 0xFFFFFFFFC /Onchip      ; set on-chip breakpoint to reset  
                                   ; address
```

Impact of SYStem.Option.ResetDetection on Reset Flags

If **SYStem.Option.ResetDetection** is used and the debugger detects a reset, the debugger will immediately assert reset in order to re-connect to the processor.

This the external reset will override or at least change one or more the reset flags.

The exact behavior also depends on the target. If the debugger's reset is connected to PORST (power-on reset/destructive reset pin), all original reset flags are cleared, while with ESR0 (or similar pin that only causes a functional reset), the bit for external reset is set in addition to original reset flags.

Multicore Debugging

One or more cores can be assigned to a TRACE32 PowerView instance. The cores are referred to by a core-index which is hard coded in the debugger software.

core-index	MPC55xx MPC563xM MPC564xA MPC5674F SPC563M SPC564A	MPC5510 MPC5643L MPC5668G MPC567xK SPC56EL SPC56HL SPC56AP	MPC5676R MPC574xR MPC5777C	MPC5747C MPC5748C MPC5746G MPC5747G MPC5748G	MPC5744B MPC5745B MCP5746B MPC5744C MPC5745C MPC5746C
core_0	1	1	1	1 (z4_a)	1 (z4)
core_1	-	2	2	2 (z4_b)	2 (z2)
core_2	-	-	-	3 (z2)	-
HSM	-	-	-	4	3
eTPU A/B/C	2/3/4	-	3/4/5	-	-
GTM	-	-	-	-	-
SPT	-	-	-	-	-

core-index	MPC5746M MPC5777M SPC57xM SPC58xG SPC58xE SPC58xN SPC58xH	MPC574xK SPC574K	MPC577xK S32R264 S32R274 S32R294 S32R372		
core_0	1	1	3 (z4)		
core_1	2	-	1 (z7_a)		
core_2	3 (IOP)	3 (IOP)	2 (z7_b)		
HSM	4	4	-		
eTPU A/B/C	-	-	-		
GTM	5	5	-		
SPT	-	-	4		

TRACE32 supports either controlling each core with a separate PowerView instance (**AMP debugging**) or controlling multiple cores with a single PowerView instance (**SMP debugging**). SMP debugging is only possible for cores of the same architecture (e.g. e200 core_0 and e200 core_1).

TRACE32 also supports mixed AMP/SMP operation. E.g. MPC5746M can be controlled with two PowerView instances, one for core_2 (IOP) and one controlling core_0 and core_1 in SMP mode.

In TRACE32 terminology, SMP debugging means to control more than one core in a single PowerView instance. Use this method for cores which run the same kernel / instance of the operating system. Cores controlled in a single PowerView instance share the following resources:

- Debug symbols
- OS Awareness
- Run control (Go, Step, Break) and breakpoints
- Debug and trace settings

If it is desired to have control over any of the above resources separately for each core, [AMP debugging](#) must be used.

Follow these steps to set up the debugger for SMP debugging:

1. Select the target processor, or use automatic CPU detection.

```
SYStem.DETECT CPU
```

2. Assign cores to this PowerView instance. Look up the core-index of each core in [above list](#). The order of the cores must match to the core order used by the kernel.

```
;Kernel: logical_core_0 = core_0, logical_core_1 = core_1  
  
;CORE.ASSIGN <logical_core_0> <logical_core_1> [...]  
CORE.ASSIGN 1 2
```

3. Start debug session and continue as usual.

```
SYStem.Up | SYStem.Mode.Attach
```

All core context dependent windows ([Register.view](#), [List](#), [Data.dump](#), etc.) show the data as seen from the currently selected core. Select a core using the command [CORE.select](#) *<logical_core_index>*.

```
Register  
  
CORE 0      ;Register window shows registers of core_0  
CORE 1      ;Register window shows registers of core_1
```

If any of the cores hits a breakpoint, PowerView automatically selects the core that hit the breakpoint. The currently selected core displayed in the status bar and can be changed by right-clicking on the core field.

It is also possible to show more than one core context at the same time, using the option `/Core <logical_core_index>`. All windows with core-dependent information support this option.

```
Register /CORE 0
Register /CORE 1

List /CORE 0
List /CORE 1
```

Example scripts for SMP debugging can be found in the demo folder, e.g.

- `~/demo/powerpc/hardware/mpc56xx/mpc5643l-dualcore/smp_demo.cmm`
- `~/demo/powerpc/hardware/mpc56xx/mpc5676r-dualcore/smp_demo.cmm`
- `~/demo/powerpc/hardware/spc56xx/spc56ap/smp_demo.cmm`

Further demo scripts available for download and upon request.

AMP Debugging

In AMP debugging mode, a separate PowerView instance is started for each core. The individual instances are completely independent of each other, but it is possible to synchronize run-control and system mode changes (see [command SYNch](#)).

An easy way to start multiple PowerView instances is to use **T32Start**. It is also possible to start further instances from a PRACTICE script.

The following steps demonstrate the setup for AMP debugging, assuming that the application is already programmed to FLASH:

1. Select the target processor, or use automatic CPU detection.

<code>;core_0 setup script:</code>	<code>;core_1 setup script:</code>
<code>SYStem.CPU MPC5517</code>	<code>SYStem.CPU MPC5517</code>

2. Assign target cores to the individual instances. Look up the core-index of each core in [above list](#). Use either **SYStem.CONFIG.CORE <core_index> <chip_index>** or **CORE.ASSIGN <core_index>**. The parameter `<chip_index>` must be the same for all cores on the same chip.

<code>SYStem.CONFIG.CORE 1. 1.</code>	<code>SYStem.CONFIG.CORE 2. 1.</code>
---------------------------------------	---------------------------------------

3. **SYStem.CONFIG.SLAVE** must be OFF for the core that starts running right from reset. Set to ON for all other cores (that are released later by the first core).

<code>SYStem.CONFIG.SLAVE OFF</code>	<code>SYStem.CONFIG.SLAVE ON</code>
--------------------------------------	-------------------------------------

4. Load debug symbols on both instances.

```
Data.LOAD appl.x /NoCODE
```

```
Data.LOAD appl.x /NoCODE
```

5. Start debug session: SYStem.Up for the core that runs right from reset. SYStem.Mode.Attach for all cores that are started later.

```
SYStem.Up
```

```
SYStem.Mode.Attach
```

6. Core_0 is halted at the reset address and core_1 remains in reset, In order to halt core_1 as soon as it is released from reset, issue the Break command.

```
Break
```

7. Start core_0. Core_1 will halt at its reset address after being released by core_0.

```
Go
```

```
WAIT !RUN() ; wait until cpu stops
```

Example scripts for AMP debugging can be found in the demo folder, e.g.

- `~/demo/powerpc/hardware/mpc56xx/mpc564xc-dualcore/`
- `~/demo/powerpc/hardware/spc56xx/spc56el-dualcore/`

Further demo scripts are available for download and upon request.

Watchdog Timer Support

e200 Core Watchdog (TCR/TSR)

The e200 core watchdog is configured and controlled through the TCR and TSR special purpose registers. There is no dedicated command to control this watchdog, but it is indirectly controlled with **SYStem.Option.FREEZE**, which halts the time base (TBU/TBL/DEC) while the core is halted in debug mode.

On-chip Watchdog (SWT)

The on-chip SWT modules can be controlled via **SYStem.Option.WATCHDOG**. By default setting, TRACE32 will disable the SWT every time the core halts in debug mode. Each TRACE32 PowerView instance will control only the SWT module(s) that is/are assigned to the core(s) it controls.

If it is intended to debug with SWT enabled, please ensure that the application sets the SWT_CR[FRZ] bit when it sets up the SWT. The FRZ bit configures the SWT to automatically halt when the core halts for the debugger.

On MPC57XX, SPC57X and SPC58X multicore processors, the SOC implementation's behavior is to that SWT_CR[FRZ] function only works if all cores under debug start and halt at the same time. We recommend to either keep the SWT permanently disabled. If this is not possible, set **SETUP.StepAllCores** to ON to ensure that the SWT can be halted under any circumstances. This behavior is not specific to the SWT, but affects all on-chip peripheral modules that implement such a Freeze function (e.g. PIT, STM).

Chip External Watchdog

TRACE32 also includes features to control on-board/chip external watchdog timers. There are several possibilities.

1. If the watchdog can be disabled by e.g. GPIO or serial communication, the debugger can write the required sequences automatically at the begin of a debug session using **Data.STARTUP** or **Data.STANDBY**.
2. If the watchdog must be serviced, by toggling a GPIO, **Data.TIMER** can be used to perform the required accesses while the core is halted.
3. Some debug cables and NEXUS adapters support controlling a chip external watchdog with a dedicated pin. This pin is controlled using **System.CONFIG.EXTWDTDIS**. the pin can be configured to either deliver a static level or a dynamic level that changes together with the core state. The table below shows which debug hardware supports this watchdog disable pin. Please note that this pin is only driven if the target is powered and the debug session is active.

Debug Cable / NEXUS Adapter	Pin
LA-7753 Debug Cable JTAG/OnCE	Not available
LA-2708 LA-3736 Debug Cable Automotive	Pin 14 on 26-pin connector (AUTO26) Pin 27 on Mictor-38 connector (with converter LA-3874) Pin 28 on 50-pin Samtec connector (with converter LA-3875)
LA-7610 NEXUS Adapter MDO12	Pin 27 on Mictor-38 connector Pin 50 on GlenAir51 connector (with LA-7611) (only supported by serial number C05030057285 and higher)
LA-7612 NEXUS Adapter MDO8	Not available
LA-7630 NEXUS Adapter AutoFocus	Pin 27 on Mictor-38 connector (with LA-7631) Pin 28 on 50-pin Samtec connector (with LA-7636) Pin 50 on GlenAir51 connector (with LA-7632)

4. Some watchdog timers can only be disabled if a pin is driven high before the target is powered. The table below lists which tools can support this scenario.

Debug Cable / NEXUS Adapter	Pin
LA-7753 Debug Cable JTAG/OnCE LA-7610 NEXUS Adapter MDO12 LA-7612 NEXUS Adapter MDO8	Not available
LA-7630 NEXUS Adapter AutoFocus	Pin "Ox0", located on the header connector of the converters LA-7631 and LA-7636. This pin will driver permanently HIGH using these commands: TrOnchip.OUT0 WDTC TrOnchip.TOOLIO2 OFF/HIGH
LA-2708 LA-3736 Debug Cable Automotive	Support for this feature on request. Please contact technical support.

Censorship Unlock

TRACE32 supports censorship unlock via JTAG commands, if the processor supports those commands. The unlock is supported by all processors but MPC55XX.

In order to unlock the processor when starting the debug session, provide the censorship password to the debugger using the command **SYStem.Option.KEYCODE**. Size and format of the password depends on the processor in use.

Censorship unlock on MPC56XX and SPC56X processors

MPC56XX/SPC56X processors only support unlocking during **SYStem.Up**. Attaching to a censored processor without resetting it is not possible.

Censorship unlock on MPC57XX, SPC57X/SPC58X and S32R processors

On MPC57XX, SPC57X, SPC58X and S32R processors, the censorship unlock is possible without asserting reset and also if functional reset is asserted. For the unlock during **SYStem.Up**, the ability to successfully unlock the processor depends on which reset pin is connected to the debugger's reset output.

If the debugger's reset output is connected to ESR0, **SYStem.Up** will perform a functional reset and the unlock will succeed. If the debugger's reset output is connected to PORST, the processor will perform a destructive reset and the unlock during **SYStem.Up** will fail.

On MPC5777C, RESET_B is the processor's only reset input and has the effect of a destructive reset. In the context of this chapter, assume RESET_B==PORST. MPC5777C can only use method 2 and, if the processor's RSTOUT is connected to the debugger's RSTOUT, method 4.

In the case the debugger's reset output is connected to PORST, several workarounds are available to gain access to the processor:

1. If the processor includes the *Debug and Calibration Interface (DCI)*, perform SYStem.Up using a functional reset via JTAG instead of asserting the reset output. This method can not be used together with **SYStem.Mode.StandBy**. It is not possible to halt the core immediately after a power-on reset.

```
SYStem.CPU <cpu>
CORE.ASSIGN <cores>
SYStem.Option.KEYCODE <password>
SYStem.Option.ResetMode FUNCtional
SYStem.Up
```

2. Use workaround to unlock and halt the core after reset release with fixed delay. The delay time set with **SYStem.Option.WaitReset** is critical. If the time is too short, the debugger sends the debug request before the processor is unlocked and SYStem.Up will fail. If the time is too long, the processor will execute many instructions before the debug request is sent. With a little fine-tuning it is usually possible to halt the core within <100 instructions, i.e. before it leaves the BAF code. This method supports debugging/tracing through power cycles.

```
SYStem.CPU <cpu>
CORE.ASSIGN <cores>
SYStem.Option.KEYCODE <password>
SYStem.Option.ResBreak OFF
SYStem.Option.WaitReset <delay> default
SYStem.Option.ResetMode PIN
SYStem.Up
```

3. Use workaround to unlock and halt the core after reset is released. It is required that the reset is visible to the debugger on the debug/trace connector's RESET pin (bidirectional reset signal). If the processor has separate RESET in and out pins, use method 4. Usually it is possible to halt the core within a few 100 instructions, i.e. before it leaves the BAF code. This method supports debugging/tracing through power cycles.

```
SYStem.CPU <cpu>
CORE.ASSIGN <cores>
SYStem.Option.KEYCODE <password>
SYStem.Option.ResBreak OFF
SYStem.Option.WaitReset 0s RESET
SYStem.Option.ResetMode PIN
SYStem.Up
```

4. Use workaround to unlock and halt the core after reset is released (using ESR0/RSTOUT feedback). It is required to connect the processor's ESR0 to the debug cable's RSTOUT input. Only supported for the [Automotive Debug Cable](#) and the [Parallel Nexus Adapters](#). Usually it is possible to halt the core within a few 100 instructions, i.e. before it leaves the BAF code. This method supports debugging/tracing through power cycles.

```
SYStem.CPU <cpu>
CORE.ASSIGN <cores>
SYStem.Option.KEYCODE <password>
SYStem.Option.ResBreak OFF
SYStem.Option.WaitReset 0s RSTOUT
SYStem.Option.ResetMode PIN
SYStem.Up
```

Methods 2.4 can not halt the core at the reset address. If debugging should start directly from the reset address, it is possible to perform a functional reset following the first reset. Please note that this may have side effects, because some code has already been executed. Also there is no real advantage, as the BAF code is usually not of interest.

```
;destructive reset with delay and unlock
SYStem.CPU <cpu>
CORE.ASSIGN <cores>
SYStem.Option.KEYCODE <password>
SYStem.Option.ResBreak OFF
SYStem.Option.WaitReset <parameters>
SYStem.Option.ResetMode PIN
SYStem.Up

;follow-up functional reset
SYStem.Option.ResBreak ON
SYStem.Option.ResetMode FUNCTIONal
SYStem.Up
```

Recovering a censored processor (MPC57XX, SPC57X/SPC58X and S32R)

If a bad application image is flashed to the processor, the processor can lock up or go through reset escalation. In either case it can be required to perform a power cycle to connect and recover the processor.

The effects of a bad application image may not become effective directly after programming the application image to flash. In some cases the problem will appear only after a target power cycle or reset without debugger. Reasons for this include the debugger initializing SRAM (via flash script execution or other setup script), the debugger disabling the watchdog (see [SYStem.Option.WATCHDOG](#)) and/or disabling the reset escalation (see [SYStem.Option.DISableResetEscalation](#)).

In order to recover a censored processor, first set up the debugger using either method 2, 3 or 4 from above chapter. If the setup is complete, halt the core at power-on reset.

The core can be halted at power-on reset either through menu or by calling a script:

Method	Action
Menu	Click on MPC5XXX (or SPC5XX), select Tools - Halt core at power-on reset
Script	DO ~/demo/powerpc/etc/standby/haltatpoweronreset.cmm

Using either method, a dialog will appear to guide you through the target power cycle. Here's a full example using the method 2 from above chapter:

```
SYStem.CPU <cpu>
CORE.ASSIGN <cores>
SYStem.Option.KEYCODE <password>
SYStem.Option.ResBreak OFF
SYStem.Option.WaitReset <delay> default
SYStem.Option.ResetMode PIN

DO ~/demo/powerpc/etc/standby/haltatpoweronreset.cmm
```

In order to successfully recover a censored processor, it is particularly essential to find a good value for <delay>. If the time is too short, the debugger sends the debug request before the unlock took place. However is the time is too long, the processor may execute the problematic code before the debug request can halt the core. It is recommended to find a suitable timing using a good target before trying recovery. The <delay> parameter of **SYStem.Option.WaitReset** accepts time values in µs resolution.

Non-secure boot (S32R294)

The processor must be in non-secure boot mode in order to load and run an image from QSPI flash.

New processors are configured to secure boot mode from factory. It can be put into non-secure boot mode either by script or by blowing fuses.

Non-secure boot by script

Program the application image to QSPI flash. Afterwards run the script

```
~/demo/powerpc/hardware/s32/s32r294/non_secure_boot.cmm
```

The processor will halt at the first instruction of the user application in Instruction RAM. Example:

```
;Set up QSPI flash programming
DO ~/demo/powerpc/flash/s32r294-qspi.cmm PREPAREONLY
FLASH.ReProgram ALL /Erase
Data.LOAD.Binary <application image>.bin 0x20000000
FLASH.ReProgram off

;Perform non-secure boot by script:
DO ~/demo/powerpc/hardware/s32/s32r294/non_secure_boot.cmm

;Core is now halted at first instruction of user application
```

Non-secure boot if fuses blown

If the fuses for non-secure boot are blown, it is important to halt the processor not before BootROM execution has completed. This can be achieved using **SYStem.Option.ResBreak** OFF. Doing so will halt the processor usually at the first instruction of the user application. If the BootROM execution should finish very early after reset, a few instructions of the user application may already be executed. Example:

```
SYStem.CPU S32R294
CORE.ASSIGN 1 2 3
SYStem.Option.ResBreak OFF
SYStem.Up
;Core is now halted at first instruction of user application
```

Troubleshooting Debug

The table below lists typical problems that occur during debugging.

Error Message	Reason
target power fail	Target has no power or debug cable is not connected. Check if the JTAG VCC pin is driven by the target.
emulation pod configuration error	<ul style="list-style-type: none">• The installed debugger software version is too old and therefore does not recognize and support the connected processor.• A JTAG communication error prevented a correct CPU detection. See the message AREA for more information.
target reset fail	<ul style="list-style-type: none">• The core is permanently in reset. Try SYStem.Option.SLOWRESET and check signal level of the JTAG RESET pin.• If the processor is multicore, make sure that the core is released from reset either by SOC or by first active core after reset. Make sure you select a core released from reset using SYStem.CONFIG.CORE. E.g. on many MPC57XX, the first core to run is core_2 (IOP).
emulation debug port fail	<ul style="list-style-type: none">• If the error occurs after a Break command, it usually means that the target application crashed. The core is unable to complete the next instruction fetch, which causes that it does not respond to the debugger's halt requests. Start a new debug session (SYStem.Up), set a breakpoint to the machine check interrupt handler and try to locate the problem. If a NEXUS trace is available, the problem can usually be found quickly using by analyzing the trace recording after the problem occurred.• If the error occurs during a memory access, the used address or address range possibly pointed to an unimplemented memory location or to a peripheral module that is disabled or otherwise not properly set up (e.g. clocks). In this case the message AREA shows the address that caused the problem.

Processors of the MPC5XXX/SPC5XX series implement a variety of trace modules. Depending on the module, the trace information is either stored on the processor or sent out through an external trace port. This section lists all available trace modules, their configuration options and examples.

e200 PCFIFO On-chip Trace

The PCFIFO is a FIFO which stores the last eight branch target addresses. It is implemented in e200 cores of the MPC55XX, MPC56XX and SPC56X processors, excluding e200z0 and e200z1 cores.

TRACE32 supports the PCFIFO for all processors and regardless of the used debug solution (JTAG and NEXUS). Using the PCFIFO does not require a trace license. It is also possible to use PCFIFO and NEXUS trace in parallel.

The PCFIFO has no configurable options. It is always enabled. None of the trace related command groups (SYStem, NEXUS, TrOnchip, Onchip) has an effect on the PCFIFO operation.

Usage: This command opens a window that shows the program flow. The program flow is reconstructed based on the PCFIFO data.

```
Onchip.List
```

Statistic analysis, RTOS tracing and run-time measurements are not possible with the PCFIFO.

Availability per Core	
e200z3 e200z4d e200z446 e200z6 e200z650 e200z750 e200z760	PCFIFO available
e200z0* e200z1 e200z2* e200z42* e200z72*	PCFIFO not available

Availability per Chip	
MPC55xx MPC563xM / SPC563M MPC564xA / SPC564A MPC564xL / SPC56EL MPC567xK / SPC56HK MPC5676R MPC5674F MPC5777C	PCFIFO available
MPC564xB/C SPC56EB/C	PCFIFO available in core_0
MPC57xx (**) SPC57x SPC58x S32Rxxx	PCFIFO available in core_0 (**): Excluding MPC5777C

MPC57XX/SPC57X/SPC58X NEXUS On-chip Trace (trace-to-memory)

Many processors of the MPC57XX/SPC57X series implement a feature to store the nexus messages of cores and peripheral trace clients into an on-chip trace memory, the so-called trace-to-memory feature.

Using the on-chip trace with a debug cable requires the on-chip trace license LA-7968X. The on-chip trace license is not required if a NEXUS adapter (LA-7630, LA-7610, LA-7612) is in use. The on-chip trace license is also not required when the Aurora NEXUS preprocessor (LA-3911) is connected.

In order to use trace-to-memory, address space and address range of the on-chip trace memory has to be set using [Onchip.TBARange](#). The usable address ranges depend on processor and device type:

Production Device	Buffer size	Onchip.TBARange
MPC5726L SPC572L	8 kByte	A:0x0D000000--0x0D001FFF
MPC5746M MPC574xK MPC5777M MPC574xR SPC574K7x SPC57EM80 SPC57HM90	16 KByte	A:0x0D000000--0x0D003FFF
SPC58xE SPC58xG	16 KByte	A:0x0D004000--0x0D007FFF
SPC58xN	32 KByte	A:0x0D000000--0x0D007FFF

In order to trace to emulation memory, use access class EEC:

Emulation Device	Buffer size	TBARange
MPC5746M (cut 1) MPC574xK MPC574xR SPC574K7x SPC57EM80 SPC58xE SPC58xG	1 MByte	EEC:0x0C000000--0xC0FFFFFF
MPC5746M (cut 2) MPC5777M SPC57HM90 SPC58xN	2 MByte	EEC:0x0C000000--0xC1FFFFFF

Processors not listed in the above tables do not support trace-to-memory.

The address range assignment can be performed through the MPC5XXX menu as well: “Onchip Trace - Set Onchip Trace Buffer”. The configuration can also be scripted depending on connected processor and debug tool. See [Onchip.TBARange](#) for an example.

The configuration of trace methods and clients is done through the [NEXUS](#) and [TrOnchip](#) command groups.

External Trace Ports (Parallel NEXUS/Aurora NEXUS)

External trace ports collect the NEXUS trace messages from cores and peripheral trace clients and send those messages to an external trace module. Depending on the processor, the messages are sent through the parallel NEXUS AUX interface (MDO, MSEO, MCKO) or through a high-speed serial connection (XILINX Aurora protocol). The Nexus adapter or the Aurora NEXUS preprocessor received the trace messages and stores them in the memory of the PowerTrace module.

Basic Setup for Parallel Nexus

The trace port settings must be done before start of the debug session. Most processors support two trace port widths, full port mode and reduced port mode. In some cases, full port mode is only supported in certain packages. Some small packages do not provide a trace port at all. Some processors only provide a trace port in special development packages. Set the desired port mode using [NEXUS.PortSize](#).

The trace port frequency divider must be selected, so that the resulting trace port frequency does not exceed the maximum specified frequency in the processor's data sheet. Many processors support trace port frequencies up to 88 MHz, but for some processors the limit is significantly lower. The MCKO divider is configured using [NEXUS.PortMode](#).

Example for MPC5676R: System frequency max. 180 MHz. MCKO frequency according to data sheet max. 82 MHz. Therefore the MCKO divider must be set to 1/3. The resulting trace port frequency is 60MHz.

```
SYStem.CPU MPC5676R

NEXUS.PortSize MD016
NEXUS.PortMode 1/3

SYStem.Up
```

Basic Setup for Aurora Nexus

The processors of the MPC5XXX/SPC5XX series require an external clock source for the Aurora NEXUS block. The Aurora NEXUS preprocessor can provide this clock for frequencies up to 3125 MHz. It is enabled using the command [NEXUS.RefClock](#).

The number of lanes is set using [NEXUS.PortSize](#), the bit rate per lane is set using [NEXUS.PortMode](#). The default settings of the debugger are usually valid for the selected processor and do not need to be changed.

Example for MPC5777M

```
SYStem.CPU MPC5777M

NEXUS.PortSize 4Lane
NEXUS.PortMode 1250Mbps
NEXUS.RefClock ON

SYStem.Up
```

All trace port settings must not be changed during an active debug session. Usually the processor must be reset (e.g. [SYStem.Up](#)) to bring up the trace port with new settings.

Tracing the Program Flow

Tracing of the program flow is enabled by default. The e200 cores support several kinds of program tracing: *branch trace messaging* and *history trace messaging*.

Branch Trace Messaging (BTM)

This is the default method set in TRACE32. The processor is configured to send a trace message for every executed branch instruction. As the debugger stores a timestamp with every received message, this method has the highest accuracy for timing measurements. The drawback is the high trace port bandwidth and trace memory consumption (i.e. short recording time). The high amount of messages can cause overflows of the on-chip message FIFO, i.e. loss of trace data.

Setup of for branch trace messaging:

```
NEXUS.BTM ON
NEXUS.HTM OFF
```

History Trace Messaging (HTM)

In *history tracing* mode, the e200 core only sends trace messages for indirect branches. Information about direct branches and amount of executed instructions is sent in occasional resource full messages. This method significantly reduces the amount of generated trace messages. Message FIFO overflows can be prevented and the recording time is increased. The program flow information is complete in HTM, so even in this mode, TRACE32 can reconstruct the full program flow without loss. The drawback of this mode is the reduced runtime measurement accuracy.

Setup of branch history tracing:

```
NEXUS.BTM ON
NEXUS.HTM ON
```

In order to remedy the loss of runtime measurement accuracy, many newer processors can additionally generate messages for function calls. With this method, the accuracy of function-level runtime measurements is identical to classical branch trace.

Setup of branch history tracing + function call tracing:

```
NEXUS.BTM ON
NEXUS.HTM ON
NEXUS.PTCM.BL_HTM ON
```

Tracing of Data (read/write) Transactions

General data tracing is enabled using the command **NEXUS.DTM**. This command enables the data trace for the full address space. The amount of generated trace messages is usually too high to be sent through the trace port and the on-chip message FIFO will overflow.

The amount of generated trace messages can be reduced by defining address ranges for which data trace is generated. Up to four address ranges are possible.

Example: Data Trace with Address Range

Use TraceData to limit the data trace to an address range. Depending on the core, 2 or 4 address ranges are possible. TraceData has no impact on program trace messaging setting.

```
;Enable data trace for read/write accesses to all peripherals
Break.Set 0xC0000000--0xFFFFFFFF /ReadWrite /TraceData
```

```
;In addition to full program trace, enable data trace for read accesses  
;to the array flags  
NEXUS.BTM ON  
Var.Break.Set flags /Read /TraceData
```

Another method of reducing trace data is [event-triggered trace filtering](#).

Tracing of Context Switches

NEXUS supports two methods of tracing context (= process, task, thread etc) changes. One method is using the data trace, the other method uses Ownership Trace Messaging (OTM).

In order to have the debugger process and display task information properly, it is required to set up OS Awareness. If the operating system in use implements the OSEK Runtime Interface (ORTI), which is often true for this processor series, see [“Configuration”](#) in OS Awareness Manual OSEK/ORTI, page 7 (rtos_orti.pdf) for instructions. For other operating systems, please check the instructions in the appropriate [OS Awareness Manual](#).

Trace Context Switches using Data Trace Messaging (DTM)

If the core implements NEXUS Class 3, the preferred method to trace context switches is to enable data trace for the memory location that holds the ID of the currently active process. If the OS Awareness is set up, the address of the memory location can be retrieved using the function [TASK.CONFIG\(\)](#).

Example for tracing context switches using DTM, for an operating system using the OSEK Runtime Interface (ORTI):

```
;set up TROS awareness  
TASK.ORTI my_rtos.orti  
  
;enable Data trace to current context information  
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

Trace Context Switch using Ownership Trace Messaging (OTM)

Data trace messaging is supported by cores implementing NEXUS Class 3 or higher. If the core implements NEXUS Class 2, then context switches can only be traced using Ownership Trace Messaging (OTM).

The processor sends an ownership trace message when the core writes to a certain SPR (special purpose register). Depending on the core in use, one or two registers can be used to issue an OTM message, the PID0 register and NPIDR. The active register is selected by the command [NEXUS.OTM](#).

This method requires cooperation of the operating system. Some operating systems will support this feature by default or by configuration, while other operating systems may require code instrumentation or implementing predefined hooks.

Look for instructions in the appropriate [OS Awareness Manual](#) for further information.

Example for tracing context switches using OTM, for an operating system using the OSEK Runtime Interface (ORTI):

```
;set up TROS awareness
TASK.ORTI my_rtos.orti

;enable ownership trace messaging via register PID0
NEXUS.OTM PID0
```

Trace Based Run-time Measurement / Timestamping

In order to enable trace based time measurements, the debugger needs a timestamp for every trace message. Depending on the target processor and trace solution in use, there are up to two kinds timestamp available:

- Tool generated timestamps. These timestamps are only available for off-chip tracing. For this processor series, tool generated timestamps are available for all PowerTrace modules. One timestamp is stored for each trace record.
- Processor generated timestamps. If the processor/core supports generation of timestamps, then these timestamps can be used for on-chip and off-chip tracing. One timestamp is generated for each NEXUS trace message.

This chapter shows how the timestamps work under several scenarios, implications of using on-chip timestamps and which processors support on-chip timestamps.

On-chip timestamps are controlled with the command [NEXUS.TimeStamps](#).

Trace Based Run-time Measurement for off-chip Parallel NEXUS

For parallel NEXUS trace ports, the NEXUS adapter stores one NEXUS trace message per trace record. Additionally there is only little and rather constant delay between event and message output, therefore the **tool generated timestamps** will yield in very exact results. The resolution of the timestamps mostly depends on the time the trace port needs to send out a single message, i.e. the trace port width (number of MDOs). As a rough estimate, the resolution will be 2~3 MCKO clock cycles for trace ports with ≥ 12 MDOs, and 8~16 MCKO clock cycles for trace ports with 2 or 4 MDOs.

If a better resolution is required, **processor generated timestamps** can be used for parallel NEXUS trace ports, if supported by the processor. The resolution of the processor generated timestamp is $1/\langle \text{core frequency} \rangle$. Please be aware that processor generated timestamps add 25~35% to the required trace port bandwidth, which may cause message loss due to on-chip message FIFO overflows.

Trace Based Run-time Measurement for off-chip Aurora NEXUS

Due to the nature of NEXUS via high-speed serial trace port, up to three messages can be stored in one trace record. Due to this, run-time measurements with **tool generated timestamps** will result in less precision on sub-function level or when trace filtering is performed. Run-time measurements on function-level, as well as average measurements usually do not lose precision. It is recommended to not use trace filtering if a run-time measurement is performed.

If a better resolution is required, **processor generated timestamps** can be used for Aurora NEXUS trace ports, if supported by the processor. The resolution of the processor generated timestamp is $1/\langle \text{core frequency} \rangle$. Please be aware that processor generated timestamps add 25~35% to the required trace port bandwidth, which may cause message loss due to on-chip message FIFO overflows.

Trace Based Run-time Measurement for on-chip Trace / Trace-to-memory

In this configuration, the generated trace data is directly stored on the processor, either in the small on-chip trace buffer or the bigger emulation memory (if emulation device available). The debugger transfers the recorded trace data through JTAG to the PC after the recording took place. Therefore **tool generated timestamps** are not available.

Run-time measurements using the on-chip trace buffer are only possible if the processor supports **processor generated timestamps**. Processor generated timestamps increase the amount of generated trace data by 25~35%. Unlike the off-chip trace solutions, the trace bandwidth to the on-chip trace buffer is usually sufficient enough to transfer the timestamp information without message loss. Due to the limited amount of buffer size (up to 2MBytes on emulation devices), recording the timestamp information will noticeably reduce the amount of trace events stored in the on-chip trace buffer.

Correlation of the Trace Timestamp with Other Tool Timestamps

If the **tool generated timestamp** is used, then this timestamp is automatically correlated to all other hardware and software timestamps of TRACE32 tools. The trace recording can be immediately correlated to recordings of PowerProbe, Integrator and to the Logic Analyzer Probe integrated into the PowerTrace modules and to software features like the **SNOOPer**. It is even possible to correlate trace recordings made by multiple PowerTrace modules that are synchronized via the PodBus interface.

If **processor generated timestamps** are used, this absolute correlation of all debug tool timestamps is not possible. If a common event is known, then the recordings can be correlated manually (see **Trace.ZERO**). As the clock sources of tool generated and processor generated timestamps are not synchronized, the time offset can increase with higher distance of the measurement from the common event.

Implications of Using the Processor Generated Timestamps

As already mentioned above, processor generated timestamps increase the amount of generated trace data by 25~35%, which will reduce the absolute run-time that fits into the (on/off-chip) trace buffer and can also cause overflows of the on-chip message FIFOs.

Processor generated timestamps can be used to make run-time measurements in a filtered trace recording even if an Aurora NEXUS trace port is used.

As this processor series does not generate timestamp overflow messages, trace events must be frequent enough so that not more than one timestamp overflow can occur between two events. Assuming a 300MHz core frequency, the 30-bit timestamp will overflow every 3.58 seconds.

Processors with on-chip timestamp support

The table below shows which processors support generation of timestamps.

Processor	Processor generated timestamp support
MPC55XX MPC56XX MPC574xP MPC5777C SPC57xK SPC58XB SPC58xC SPC58xE SPC58xG SPC58xH	Not supported
MPC5746M, rev2 MPC574xC MPC574xD MPC574xG MPC577xK S32R274 S32R294 S32R372 SPC57EM, rev2 SPC58NN	Supported for all cores
MPC5777M, rev. 2 SPC57HM, rev. 2	Supported for core_2 (IOP)

Trace Filtering and Triggering with Debug Events

Overview

Any debug event available on e200 cores can be either used to halt the core, or act as watchpoint. A watchpoint hit event again can be configured to either trigger a *watchpoint hit message*, or to act as input event for selective tracing. TRACE32 offers a variety of features based on watchpoints.

Watchpoints are set using the command **Break.Set**, similar to breakpoints that halt the core, but additionally include an option to define the desired behavior:

Break.Set <address>|<range> /<action>

Define trace filter or trigger

The list below shows all available trace filtering and trigger actions:

<action>	Behavior
TraceEnable	Configure the trace source to only generate a trace message if the specified event occurs. Complete program flow or data trace is disabled. If more than one TraceEnable action is set, all TraceEnable actions will generate a trace message. See Example .
TraceON TraceOFF	If the specified event occurs, program and data trace messaging is started (TraceON) or ends (TraceOFF). In order to perform event based trace start/end to program trace and data trace separately, use Alpha-Echo actions. Functionality of TraceOFF is only given when used on combination with TraceON. Using TraceOFF stand-alone is not supported by the processor.
TraceTrigger	Stop the sampling to the trace on the specified event. A trigger delay can be configured optionally using Analyzer.TDelay .
BusTrigger	If the specified event occurs, a trigger pulse is generated on the podbus trigger line. This trigger signal can be used to control other podbus devices (e.g. PowerProbe) or to control external devices using the trigger connector of the PowerDebug/PowerTrace module (see TrBus).
BusCount	The specified event is used as input for the counter of the PowerDebug/PowerTrace module. See Count for more information.
WATCH	Set a watchpoint on the event. The CPU will trigger the EVTO pin if the event occurs and generate a watchpoint hit message if the trace port is enabled.
Alpha - Echo	Declares an event for several special control / trigger actions: <ul style="list-style-type: none"> • Configure event triggered trace start/end for program and data separately. See TrOnchip.Alpha for details. • Configure Trace/Trigger events for additional nexus trace clients. See TrOnchip.Alpha for details. • Configure event triggered performance counter start/stop. See BMC.<counter>.ATOB for details. • Configure trigger events for the complex trigger unit. See “Complex Trigger Unit for Nexus MPC5xxx” (app_ctu_mpc5xxx.pdf) for details.

NOTES:	<ul style="list-style-type: none"> • TraceEnable action on data address uses the data address selectors of the NEXUS module, no watchpoints are used in this case. • Actions on data address (excluding TraceEnable) can not differentiate between read and write access. Only /ReadWrite is allowed.
---------------	---

Example: Selective Program Tracing

TraceEnable enables tracing exclusively for the selected events. All other program and data trace messaging is disabled.

```
;Only generate a trace message when the instruction  
;at address 0x00008230 is executed.  
Break.Set 0x00008230 /Program /TraceEnable
```

TraceEnable can also be applied on data trace. In this case, filtering is performed using the data trace selectors of the NEXUS module, which supports differentiation between read and write accesses:

```
;Only generate a trace message when the core writes to variable flags[3].  
Var.Break.Set flags[3] /Write /TraceEnable
```

TraceEnable can be used for high precision time-distance measurements:

```
;Get start and end address of function to be measured  
  &a1=sYmbol.BEGIN(func_to_measure)  
  &a2=sYmbol.EXIT(func_to_measure)  
  
;Only generate trace messages on the addresses used for measurement  
  Break.Set &a1 /Program /TraceEnable  
  Break.Set &a2 /Program /TraceEnable  
  
;run application  
  Trace.Init  
  Go  
  WAIT 5.s  
  Break  
  
;statistic analysis  
  Trace.STATistic.AddressDURation &a1 &a2  
  
;plot time distance over time (can take some time for analysis)  
  Trace.PROFILECHART.DURATION /FILTERA ADDRESS &a1 /FILTERB ADDRESS &a2
```

NOTES:

- TraceEnable on program events needs only very low bandwidth and enables very long recording times, but disables all but the enabled trace events.
- The time measurement commands from above example also work with a normal trace recording without TraceEnable, but with a shorter recording time.
- For parallel NEXUS, using TraceEnable increases the precision of the timing measurement.
- For Aurora NEXUS, TraceEnable can be used for time measurements only if the processor generates trace messages with timestamps. See [NEXUS.TimeStamps](#) for details. If the processor does not support timestamp generation, do not use TraceEnable for time measurements.

Example: Event Controlled Program/Data Trace Start and End

Program and data trace can be enabled and disabled based on debug events. TraceON and TraceOFF control both program and data trace depending on NEXUS.BTM/DTM setting. TraceON and TraceOFF control the message source, i.e. the core's NEXUS module:

```
;Enable program/data trace when func2 is entered
;Disable program/data trace when last instruction of func2 is executed.
Break.Set sYmbol.BEGIN(func2) /Program /TraceON
Break.Set sYmbol.EXIT(func2) /Program /TraceOFF
```

```
;Enable program/data trace when variable flags[3] is accessed
Var.Break.Set flags[3] /ReadWrite /TraceON
```

```
;Disable program/data trace data address 0x40000230 is read or written
with 16-bit value 0x1122
Break.Set 0x40000230 /ReadWrite /Data.Word 0x1122 /TraceOFF
```

```
;Enable program/data trace only when a specific task is active
;NOTE: RTOS support must be set up correctly
&magic=TASK.MAGIC("my_task") ;get magic value for the task of interest
Break.Set task.config(magic) /ReadWrite /Data &magic /TraceON
Break.Set task.config(magic) /ReadWrite /Data !&magic /TraceOFF
```

It is also possible to enable/disable program and data trace messaging separately:

```
;Enable/disable only program trace based on events,  
;full data trace messaging  
NEXUS.DTM ReadWrite  
Break.Set func2 /Program /Onchip /Alpha  
TrOnchip.Alpha ProgramTraceON  
Var.Break.Set flags[8] /ReadWrite /Onchip /Beta  
TrOnchip.Beta ProgramTraceOFF
```

```
;In addition to full program trace, enable/disable data trace messaging  
;only for func2  
NEXUS.BTM ON  
Break.Set sYmbol.BEGIN(func2) /Program /Onchip /Alpha  
TrOnchip.Alpha DataTraceON  
Break.Set sYmbol.EXIT(func2) /Program /Onchip /Beta  
TrOnchip.Beta DataTraceOFF
```

Example: Event Controlled Trace Recording

Debug/trace events can also be used to trigger and stop the trace recording (i.e. message sink):

```
;Generate a trigger for the trace recording module when  
;the specified event occurs. Trace recording stops delayed after  
;another 10% of the trace buffer size was recorded.  
;  
Break.Set sieve /Program /TraceTrigger  
Trace.TDelay 10%
```

Example: Event Controlled Trigger Signals

TRACE32 can generate a trigger signal based on debug/trace events. The trigger signal can be used to control PowerProbe or PowerIntegrator, as well as with external tools (using the trigger connector)

```
;Generate PODBUS trigger signal on data access event with data value  
Var.Break.Set flags[9] /ReadWrite /Data.Byte 0x01 /BusTrigger  
  
;forward signal to trigger connector  
TrBus.Connect Out  
TrBus.Mode High
```

Example: Event Counter

There is also a built-in event counter which can be used to count debug/trace events or to measure the event frequency:

```
;Measure the execution frequency of function sieve
Break.Set sieve /Program /BusCount
Count.Mode Frequency
Count.Gate 1.s           ;measure for 1 second
Go                       ;run application
Count.Go                 ;start measurement
PRINT "sieve freq = "+FORMAT.DECIMAL(1.,Count.VALUE()/1000.)+"Hz"
Count.state              ;open event counter window
```

Tracing Peripheral Modules / Bus Masters

Many processors support tracing of peripheral bus master trace clients, e.g. DMA or FlexRay controllers. The clients are controlled with the **NEXUS.CLIENT<x>** commands.

As for the core's data trace, the amount of generated trace messages is usually too high to be sent through the trace port and the on-chip message FIFO will overflow. Therefore it is necessary to set filters to reduce the amount of trace messages. The MPC5xxx processor series' peripheral bus master trace clients support two filtering methods, explained in below examples.

Example: Filter by Address Range

The MPC5xxx peripheral bus master trace clients support two freely configurable address ranges. The client will only generate trace messages, if the read or write address is inside one of those address ranges. The range only applies to the selected clients. Other clients and the cores can be configured independently.

This example shows how to enable DMA trace only for a given address range:

```
;select DMA trace client
NEXUS.CLIENT1.SELECT DMA_0

;set Alpha event on address range and write access
Break.Set D:0x40001000--0x400017FF /Write /Onchip /Alpha

;Assign Alpha event to CLIENT1, function TRACEDATA
TrOnchip.Alpha TraceDataClient1
```

Example: Event Controlled Trace Start and End

The MPC5xxx peripheral bus master trace clients support two freely configurable address ranges. The client will only generate trace messages, if the read or write address is inside one of those address ranges.

Configure the trace of the DMA controller, so that DMA trace starts when the DMA controller writes to 0x1000 and stops when DMA controller wrote 0x1040.

```
;select DMA trace client
NEXUS.CLIENT1.SELECT DMA_0

; define events for DMA data trace on/off
Break.Set D:0x40001000 /Write /Onchip /Alpha
Break.Set D:0x40001040 /Write /Onchip /Beta

; assign events to data trace on/off for client 1
TrOnchip.Alpha TraceONClient1
TrOnchip.Beta TraceOFFClient1
```

Trace Filtering and Triggering Features Provided by TRACE32

For processors with parallel Nexus trace port, TRACE32 implements the Complex Trigger Unit (CTU). The CTU is a state machine which uses Nexus trace messages and other signals as input, and can be used to perform a filtered trace recording, generate output signals or to halt the program execution. It's main usage on MPC55XX is to provide break-on-data-value functionality, which is not supported by the e200z6 debug logic.

See [“Complex Trigger Unit for Nexus MPC5xxx”](#) (app_ctu_mpc5xxx.pdf) for features and programming examples of the CTU.

Troubleshooting Trace

Tracing VLE or Mixed FLE/VLE Applications

On processors with support for both VLE (variable length encoding) and FLE (std PowerPC instruction encoding), the debug symbols must provide information about which address ranges are compiled for which encoding. The debugger will show wrong information in the [List](#) window, and there will be flow errors in the [Trace](#) analysis, if one of below situations occur:

- If debug symbols have not been loaded, the disassembler is probably not configured to display the currently used instruction set. Use [SYStem.Option.DisMode <VLE | FLE>](#) to configure manually, or use [SYStem.Option.DisMode AUTO](#) (default) to display the instruction set according to the current state of the CPU. It is always possible to use [SYStem.Option.DisMode <VLE | FLE>](#) to manually force a specific decoding.
- If debug symbols have been loaded, the debugger will use the information from the debug symbols if [SYStem.Option.DisMode AUTO](#) is selected.
- If the debug symbols are loaded, but the disassembly is still wrong, the information about the used instruction set may be wrong. The command [sYmbol.LisT.ATTRibute](#) opens a window that displays all address ranges of the debug symbols and if the instructions are FLE or VLE. The

information from **MMU.DUMP.TLB1** can help to compare the current VLE setup in the MMU with the information from the debug symbols. If the information in **sYmbol.List.ATTRibute** is wrong, please check your linker configuration.

- If the debugger lists VLE/FLE instructions as expected for the address ranges shown in **sYmbol.List.ATTRibute** but there are still sporadic errors in the disassembly, then the used linker is not properly configured to link VLE code. In this case it was observed that all debug symbols were aligned to 4 byte boundaries, while the actual code was aligned at 2 byte boundaries. Check linker version and linker configuration.
- Early compiler versions supporting VLE often had buggy VLE debug symbols. Check if a compiler update is available.

As a workaround, it is also possible to override the information loaded from the debug symbols using **sYmbol.NEW.ATTRibute**.

```
; Syntax:
; sYmbol.NEW.ATTRibute <FLE|VLE> <start-address>
; sYmbol.NEW.ATTRibute <FLE|VLE> <range>

; Example: override
Data.LOAD auto project.elf
sYmbol.NEW.ATTRibute FLE 0x00000000--0x0003ffff
sYmbol.NEW.ATTRibute VLE 0x00040000--0x00ffffff

;add new attribute FLE for BAM, which is not covered by debug symbols
sYmbol.NEW.ATTRibute FLE 0xFFFFF000
```

FLASH Programming Scripts

Reference scripts for the programming of the on-chip FLASH of MPC5XXX/SPC5XXX devices can be found in the TRACE32 installation folder under `~~/demo/powerpc/flash/*.cmm`. The FLASH programming binaries included with TRACE32 are generated from FLASH libraries provided by Freescale and/or ST. The script names follow the following convention:

Manufacturer	Filename
Freescale	<code>~~/demo/powerpc/flash/mpc5*.cmm</code>
STMicroelectronics	<code>~~/demo/powerpc/flash/spc5*.cmm</code>
Freescale/STM JDP Joint Development Program	<code>~~/demo/powerpc/flash/jpc5*.cmm</code>

For automatic selection of the right flash script, use the included selector script: `mpc5xxx.cmm`.

All scripts can be used without change in any project. For flexibility, the flash scripts support some parameters. All parameters are optional:

Parameter	Behavior
(no parameter)	Set up processor and debugger for flash programming, then open a file select dialog. The file selected by the user will be programmed to flash. This function is also available through the menu: MPC5XXX -> Tools -> Program FLASH
PREPAREONLY	Only set up processor and debugger for flash programming. No user interaction occurs.
SKIPCONFIG	Set up processor and debugger for flash programming, but skip basic debugger / target setup for advanced configuration. No user interaction occurs.
PORTSHARING=ON	Enable debug port sharing with other tools, e.g. ETAS ETK. If this parameter is passed to the flash script, the flash script calls <code>SYStem.CONFIG PortSHaring ON</code>

This example shows how to call a flash script from your own script. It is important to reset the processor (**SYStem.Up**) after flash programming, because the processor setup for flash programming can differ from the one needed for the application to run (e.g. VLE vs. FLE):

```
;prepare flash programming
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY

;activate flash programming (erasing unused sectors)
FLASH.ReProgram ALL /Erase

;load file(s)
Data.LOAD.Elf project.x
Data.LOAD.S3 data.s3

;commit data to flash
FLASH.ReProgram off

;after flash programming: reset processor
SYStem.Up
```

To improve the flash programming speed, set up PLL and JTAG clock after calling the flash script. The PLL setting and possible maximum JTAG frequency strongly depends on the target design and processor. Use carefully, because overclocking can damage the processor.

```
;prepare flash programming
DO ~/demo/powerpc/flash/mpc5676r.cmm PREPAREONLY

Data.Set ANC:0xC3F80000 %LONG 0x06000000 ;PLL for fast programming
SYStem.BdmClock 20.MHz ;for faster download

;program FLASH
FLASH.ReProgram ALL /Erase
Data.LOAD.Elf project.x
FLASH.ReProgram off

;after flash programming: reset processor
SYStem.Up
```


This example shows how to perform advanced debugger/target configuration. The bold lines show some possible advanced settings (use only when appropriate!):

```
SYStem.RESet
SYStem.CPU MPC5676R
SYStem.Option.WATCHDOG OFF
SYStem.CONFIG.PortSHaring ON ;calibration tool connected
SYStem.CONFIG.EXTWDTDIS HIGH ;disable external watchdog
SYStem.Up

;prepare flash programming
DO ~/demo/powerpc/flash/mpc5676r.cmm PREPAREONLY SKIPCONFIG

;activate flash programming (erasing unused sectors)
FLASH.ReProgram ALL /Erase
```

Requirements due to FLASH ECC Protection

The on-chip FLASH of MPC55XX/56XX implements an ECC error detection/protection. Therefore, the minimum program size is two consecutive 32-bit words, aligned on a 0-modulo-8 byte address. The resulting 64-bit units **must not be programmed more than once** after each erase cycle. If such a unit is programmed more than once, the unit can become inaccessible due to ECC detection.

Multiple programming of 64-bit units can occur, if

- a file is programmed (e.g. ELF, SRECORD), which contains memory blocks which are not aligned to 0-modulo-8 byte addresses,
- several files are programmed with overlapping address ranges.

It is recommended to use **FLASH.ReProgram** instead of **FLASH.Program** to program flash. **FLASH.ReProgram** will merge all data loaded by **Data.LOAD** or **Data.Set** before programming the device. This way TRACE32 will ensure that no 64-bit unit will be programmed more than once. The provided flash example scripts also use **FLASH.ReProgram**.

If the FLASH **already contains ECC errors**, please make sure to call **FLASH.Erase** once before calling **FLASH.ReProgram**. Starting with Build 20739, **FLASH.ReProgram** supports programming FLASH with ECC errors without prior call of **FLASH.Erase**.

The FLASH can contain ECC errors, if

- the FLASH memory is damaged (in this case **FLASH.Erase** might fail too)
- 64-bit units were programmed more than once
- problems during programming (e.g. power fail, software issues)
- it is a **new device**, which was never programmed/erased after factory tests

Programming the RCHW or Boot Header

The RCHW or boot header holds information about the start address for the core(s) amongst other configuration settings. Depending on the target processor, the RCHW/boot header is either read by the BAM or BAF code, or directly evaluated by the processor's reset logic. If a SYStem.Up is performed, the processor sets the PC to the BAM/BAF start address in the BAM/BAF case, while in the reset logic case, the PC will be directly set to the start address in the RCHW or boot header.

The RCHW or boot header typically is (and should be) part of the flash image, but it can also be generated using the debugger, e.g. if the start address can be determined using debug symbols. The following example shows how to manually program the RCHW on a MPC55XX processor. Addresses and contents vary depending on the target processor in use.

```
FLASH.ReProgram ALL /ERASE
Data.LOAD.ELF * E:0x00--(&flashsize-1)
Data.Set 0x00 %Long 0x005A0000 ;boot identifier
Data.Set 0x04 %Long Var.ADDRESS("_start") ;start address
FLASH.ReProgram off
```

NOTE:	Programming the boot identifier without (or with an invalid) start address can render the processor unusable and the debugger will fail to connect to the processor. In order to regain access, pull the FAB pin of the processor HIGH. This will configure the processor to boot from internal ROM instead of trying to fetch the illegal boot address from flash. Once the FAB pin is high, the debugger can connect and reprogram the FLASH.
--------------	---

Programming the Shadow Row

The flash programming example scripts included with TRACE32 declare the shadow row sector, but the sector is set to flash algorithm NOP instead of TARGET. This way the sector is protected against accidental erase or programming.

In order to enable programming or erasing the shadow row, the flash declaration has to be changed to algorithm TARGET. Example:

```
;prepare flash programming
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY

;enable shadow row programming (change type NOP to TARGET)
FLASH.CHANGETYPE <shadow_row_base>++0x3FF TARGET

;program FLASH
FLASH.ReProgram ALL
...
```

Once the shadow row sector is set to type TARGET, it can be erased and programmed. **The censorship word is however still protected.** Every time the shadow row is erased, the debugger will force restore the default censorship word.

The next chapter describes how to override this extra protection and change the censorship word in flash.

Programming Serial Boot Password and Censorship Word

After enabling shadow row programming as described above, the serial boot password can be programmed.

The censorship word has an extra protection. By default, the debugger will force that the censorship word gets programmed to 0x55AA55AAFFFFFFFF (for C90LC-Flash: 0x55AA55AA55AA55AA). In order to program the censorship word, use the following sequence:

```
;prepare flash programming and enable shadow row
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY
FLASH.CHANGETYPE <shadow_row_base>++<size> TARGET

;programming sequence to enable censored mode
FLASH.AUTO <shadow_row_base>++<size> /CENSORSHIP
Data.Set <censorship_address> %QUAD 0x55AA1234FFFFFFFF ;for C90FL/H7F
Data.Set <censorship_address> %QUAD 0x55AA123455AA1234 ;for C90LC
Data.Set <password_address> %QUAD 0xFEEDFACECAFEBEEF ;set password
FLASH.AUTO off
```

The next sequence shows how to disable censorship:

```
;prepare flash programming and enable shadow row
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY
FLASH.CHANGETYPE <shadow_row_base>++<size> TARGET

;programming sequence to uncensor device
FLASH.AUTO <shadow_row_base>++<size> /CENSORSHIP
Data.Set <censorship_address> %QUAD 0x55AA55AAFFFFFFFF ;for C90FL/H7F
Data.Set <censorship_address> %QUAD 0x55AA55AA55AA55AA ;for C90LC
Data.Set <password_address> %QUAD 0xFEEDFACECAFEBEEF ;default password
FLASH.AUTO off
```

NOTE:

The censorship word must have at least one bit set to 0 and one set to 1, in each half word (16 bit block).
A valid password must have at least one bit set to 0 and one set to 1, in each half word (16 bit block). The password restriction only applies to 64-bit passwords. Processors with 256-bit passwords do not have that restriction.

Newer processors (MPC56XX, SPC56X and later) have a feature to inhibit censorship via JTAG (using the serial password). See [S**ystem.Option.KEYCODE**](#) for details.

TEST / UTEST / OTP FLASH Programming

Many processors implement one or more test sectors. Those sectors can contain factory data and configuration (DCF) records, but can also hold user-specific data, like e.g. serial numbers. The TEST/UTEST sectors are usually OTP, either by factory default or by user configuration.

In order to protect these sectors from accidental programming/overwriting, they are marked with option **/OTP** in the flash scripts. Example for MPC5746M/SPC57EM (excerpt from jpc574xm.cmm):

```
FLASH.RESet
...
FLASH.Create 4. 0x01300000--0x0133FFFF TARGET Quad 0x030c
FLASH.Create 4. 0x01340000--0x0137FFFF TARGET Quad 0x030d
; UTEST address space
FLASH.Create 6. 0x00400000--0x00403FFF TARGET Quad 0x0500 /OTP /INFO "UTEST"
```

Programming an OTP Sector

FLASH sectors declared with option **/OTP** remain disabled (all writes are ignored) during normal FLASH programming, to prevent accidental programming. Any FLASH command that makes use of the erase feature (i.e. **FLASH.Erase**, **FLASH.AUTO**, **FLASH.ReProgram**) will omit sectors declared as OTP.

OTP sectors must be programmed using **FLASH.Program** with option **/OTP**. It is highly recommended to program normal sectors and OTP separately:

```
;Step 0: Prepare flash programming
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY

;Step 1: Program normal sectors (OTP sectors are ignored)
FLASH.ReProgram ALL /Erase
Data.LOAD.Elf project.x
Data.LOAD.S3 data.s3
FLASH.ReProgram off

;Step 2: Program OTP
FLASH.Program 0x00400000--0x00403FFF /OTP
Data.LOAD.Binary <file> <start_address>
;and/or alternatively:
Data.Set %Quad <address> %Quad <dcf_record1> [<dcf_record2> ...]
FLASH.Program off
```

On some processors of the SPC58X series, the UTEST sector has a 128 bit ECC granularity. Because of this, an even number of DCF records (quad words) must be programmed. If the number of DCF records is not a multiple of 2, a data alignment error will occur and FLASH will not be programmed. A dummy DCF record can be added in order to get an even number of DCF records. Example:

```
;Step 0 and 1 as above.
;Step 2: Program OTP (128 bit ECC granularity)
FLASH.Program 0x00400000--0x00403FFF /OTP
Data.Set %Quad <address> %Quad <dcf-record1> <dcf-record2>
;and/or alternatively:
Data.Set %Quad <address> %Quad <dcf-record> <dummy-dcf-record>
FLASH.Program off
```

A dummy DCF record *usually* can be one of the following. Please make sure to check with the processor reference manual and/or processor manufacturer for the appropriate method.

- The same DCF record as written to the first 64 bits of the 128 bit block (i.e. the same DCF record is written twice).
- A DCF record with an invalid chips select (refer to the processor reference manual for invalid/unused DCF chip selects).

As an additional measure of safety, the programming script could be extended to only program the OTP if it is still unprogrammed. Doing so can help to assure that no illegal combinations of DCF records will be programmed. Example:

```
Data.Set 0x00400000--0x00403FFF %Quad 0xFFFFFFFFFFFFFFFF /DIFF
IF FOUND()
(
    PRINT "OTP already programmed, aborting."
)
ELSE
(
    FLASH.Program 0x00400000--0x00403FFF /OTP
    Data.LOAD.Binary otpodata.bin
    FLASH.Program off
)
```

Programming an UTEST Sector which is not set to OTP

The UTEST sector can be either OTP or erasable, depending on factory configuration or custom configuration. Please check the processor reference manual for details.

If the UTEST sector is not configured to OTP, erasing the sector can be enabled by re-declaring UTEST without option /OTP, e.g.

```
;prepare flash programming
DO ~/demo/powerpc/flash/mpc5xxx.cmm PREPAREONLY

;delete original UTEST declaration
FLASH.Delete 0x00400000--0x00403FFF

;re-declare as normal sector (copy parameters from flash script)
FLASH.Create 6. 0x00400000--0x00403FFF TARGET Quad 0x0500
```

NOTE:	If an OTP sector is declared without option /OTP in the flash script, TRACE32 can not provide any protection against accidental programming or overwriting of OTP sector contents. In the worst case, accidental overwriting (of e.g. DCF records or factory configuration data) can permanently damage the processor.
--------------	---

Brownout Depletion Recovery

If a brownout occurs during an erase operation on the C90FL flash, the flash blocks being erased can be left in an indeterminate state (invalid ECC values). A brownout is defined as an accidental power loss or supply voltage drop or unexpected reset. For more information see Freescale AN4521. Brownout depletion recovery is implemented for all affected devices. The recovery is performed automatically on-demand during flash operations (ReProgram, Erase or AUTO). No additional command or user action is required.

Troubleshooting FLASH

File Contains Addresses Outside the FLASH Sectors

A frequently occurring problem is that the file to be programmed to FLASH contains address ranges outside the addresses of the FLASH sectors (e.g. SRAM or unimplemented memory space). This can interrupt the flash algorithm or cause the processor to reset. In order to investigate such problems, use the following commands to find out which address ranges are contained in the loaded file.

```
; find out which addresses are contained in loaded file
Data.LOAD.auto * /VM
sYmbol.List.Map
```

If the **sYmbol.List.Map** window shows an address range outside of FLASH, the problem is usually a bug in the linker script and should be fixed there. In the case there is data linked to SRAM, there is often a chance that the application does not run reliably because of missing data initialization. If section linked to SRAM is not assumed to be an error, an alternative solution is to give the Data.LOAD command an address range.

```
; load only a part of the contained addresses (because we know it's ok)
FLASH.ReProgram ALL
Data.LOAD.auto * 0--0xFFFFFFFF
FLASH.ReProgram off
```

ECC Errors in FLASH

The FLASH programming commands **FLASH.ReProgram** and **FLASH.AUTO** will read from FLASH sectors that are going to be modified. Although the debugger tries to recover automatically from ECC errors, sometimes it is required to use **FLASH.Erase** to recover from ECC errors. In order to maintain high flash programming/update speeds and to not unnecessarily increase programming cycles, it is recommended to use FLASH.Erase only as fallback instead of adding it to the start of every new Flash.ReProgram cycle.

SYStem.BdmClock

Set BDM clock frequency

Format:	SYStem.BdmClock <i><rate></i>
<i><rate></i> :	1kHz ... 50MHz

Selects the frequency for the debug interface. For multicore debugging, it is recommended to set the same JTAG frequency for all cores.

NOTE:

The recommended maximum JTAG frequency is 1/4th of the core frequency with default PLL configuration after reset. The JTAG frequency can be increased after configuring the PLL.

Please make sure to decrease the JTAG frequency to 1/4th of the reset core frequency before a target reset (e.g. [SYStem.Up](#)).

See processor data sheet for additional restrictions of the max. JTAG frequency.

Format:	SYStem.CONFIG.state [/<tab>]
<tab>:	DebugPort Jtag XCP

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

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

<tab>	Opens the SYStem.CONFIG.state window on the specified tab. For tab descriptions, see below.
DebugPort	Lets you configure the electrical properties of the debug connection, such as the communication protocol or the used pinout.
Jtag	Informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip.
XCP	Lets you configure the XCP connection to your target. For descriptions of the commands on the XCP tab, see “ XCP Debug Back-End ” (backend_xcp.pdf).

Format:	SYStem.CONFIG <i><parameter></i> <i><number_or_address></i> SYStem.MultiCore <i><parameter></i> <i><number_or_address></i> (deprecated)
<i><parameter></i> (DebugPort):	CORE <i><core_index></i> <i><chip_index></i> DEBUGPORT DebugCable0 Analyzer0 XCP0 Slave ON OFF TriState ON OFF
<i><parameter></i> (JTAG):	DRPRE <i><bitcount></i> DRPOST <i><bitcount></i> IRPRE <i><bitcount></i> IRPOST <i><bitcount></i> TAPState 7 12 TCKLevel 0 1 CJTAGFLAGS <i><flags></i> CJTAGTCA <i><tca></i>

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 processor in the JTAG chain. The information is required before the debugger can be activated e.g. by a **SYStem.Up**. See example below.

TriState has to be used if (and only if) more than one debugger are connected to the common JTAG port at the same time. TAPState and TCKLevel define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

NOTE:	When using the TriState mode, nTRST/JCOMP must have a pull-up resistor on the target. In TriState mode, a pull-down is recommended for TCK, but targets with pull-up are also supported.
--------------	--

Debug port parameters:

CORE	The parameter <code><core_index></code> defines which core the PowerView instance controls. Counting starts with 1, i.e. 1: core_0, 2: core_1, 3: core_2... The parameter <code><chip_index></code> is important for multi-chip targets. All PowerView instances which control cores on the same physical chip must use the same <code><chip_index></code> value. Although any value > 0 is allowed, it is recommended to use 1 for single-chip targets. See Multicore Debugging for details and examples.
DEBUGPORT	Use this command to select the debug port to use for JTAG communication. On TRACE32 PowerTools hardware, the debug port can be either DebugCable0 or Analyzer0 , if a Nexus adapter is connected. If TRACE32 is operated on software-only mode, DebugPort XCP0 allows debugging via a 3rd party XCP slave.
Slave	(default: OFF) If more than one PowerView instance is using the same JTAG port, all except one must have this option active. Only one debugger - the “master” - is allowed to control the signals nTRST/JCOMP and (n)RESET).
TriState	(default: OFF) If more than one debugger share the same JTAG port, this option is required. The debugger switches to tristate mode after each JTAG access. Then other debuggers can access the port.

JTAG parameters (see [daisy chain example](#) below):

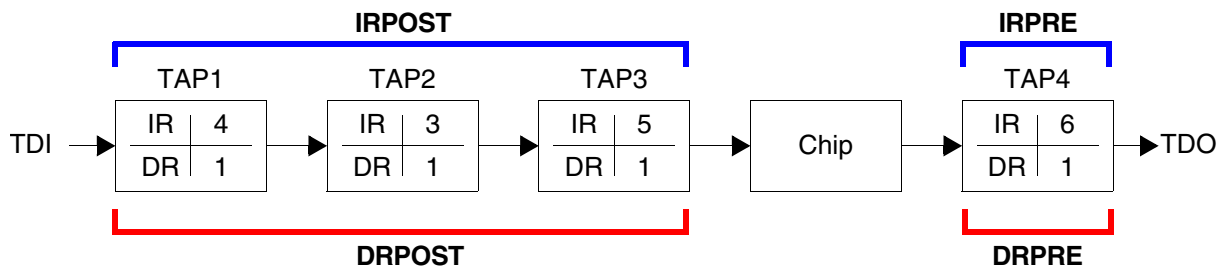
DRPRE	(default: 0) <code><number></code> of TAPs in the JTAG chain between the core of interest and the TDO signal of the debugger. If each core in the system contributes only one TAP to the JTAG chain, DRPRE is the number of cores between the core of interest and the TDO signal of the debugger.
DRPOST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

cJTAG parameters (do not change debugger default settings):

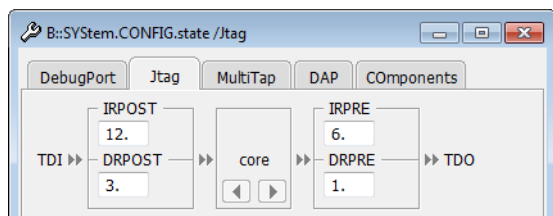
CJTAGFLAGS <flags>	<p>Activates bug fixes for “cJTAG” implementations.</p> <p>Bit 0: Disable scanning of cJTAG ID.</p> <p>Bit 1: Target has no “keeper”.</p> <p>Bit 2: Inverted meaning of SREDGE register.</p> <p>Bit 3: Old command opcodes.</p> <p>Bit 4: Unlock cJTAG via APFC register.</p> <p>Default: 0</p>
CJTAGTCA <value>	Selects the TCA (TAP Controller Address) to address a device in a cJTAG Star-2 configuration. The Star-2 configuration requires a unique TCA for each device on the debug port.

Daisy-Chain Example



IR: Instruction register length **DR:** Data register length **Chip:** The chip you want to debug

Daisy chains can be configured using a PRACTICE script (*.cmm) or the **SYStem.CONFIG.state** window.



Example: This script explains how to obtain the individual IR and DR values for the above daisy chain.

```
SYStem.CONFIG.state /Jtag      ; optional: open the window

SYStem.CONFIG IRPRE  6.        ; IRPRE: There is only one TAP.
                                ; So type just the IR bits of TAP4, i.e. 6.

SYStem.CONFIG IRPOST 12.       ; IRPOST: Add up the IR bits of TAP1, TAP2
                                ; and TAP3, i.e. 4. + 3. + 5. = 12.

SYStem.CONFIG DRPRE  1.        ; DRPRE: There is only one TAP which is
                                ; in BYPASS mode.
                                ; So type just the DR of TAP4, i.e. 1.

SYStem.CONFIG DRPOST 3.        ; DRPOST: Add up one DR bit per TAP which
                                ; is in BYPASS mode, i.e. 1. + 1. + 1. = 3.
                                ; This completes the configuration.
```

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

Format: SYStem.CONFIG.DEBUGPORTTYPE [JTAG CJTAG]

Default: JTAG.

This command is used to configure the debug port type used by the debugger.

Hardware Requirements for cJTAG Operation

Debug modules: These debug modules do **not** support cJTAG operation:

- LA-7702: PowerDebug module (without USB or Ethernet connection)
- LA-7704: PowerDebug USB1 module

If the debug cable or Nexus adapter supports cJTAG operation depends on the production date. The production date is encoded in the serial number (CYYMMxxxxxx, YY=year, MM=month).

- LA-7753 (14-pin JTAG/OnCE header): Must be from 09/2009 or later
- LA-2708, LA-3736 (AUTO26 header): supported by all versions
- LA-7630 (Nexus AutoFocus adapter):
 - 01/2013 and newer: full support
 - 06/2009...12/2012: restricted support (TDI and TDO signal of Nexus adapter must be disconnected from any target signal when using cJTAG)
 - 05/2009 and older: not supported
- LA-7610 (Nexus adapter 3.3V): not supported
- LA-7612 (Nexus adapter 5V): not supported

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

Default for Automotive/Automotive PRO Debug Cable: High.
Default for XCP and NEXUS AutoFocus Adapter: OFF.

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

OFF	The WDTDIS pin is not driven. (only XCP and parallel NEXUS Adapter)
High	The WDTDIS pin is permanently driven high.
Low	The WDTDIS pin is permanently driven low.
HighwhenStopped	The WDTDIS pin is driven high when program is stopped. (not XCP)
LowwhenStopped	The WDTDIS pin is driven low when program is stopped. (not XCP)
Trigger	The WDTDIS pin is driven by the Complex Trigger Unit. (only parallel NEXUS Adapter)
SLAVE	The WDTDIS state of the XCP slave is not changed. (XCP only)

Format:	SYSystem.CONFIG.PortSHaRing [ON OFF Mode <mode>] SYSystem.Option.ETK [ON OFF] (deprecated) SYSystem.Option.GSI [1 2 OFF] (deprecated)
<mode>:	ETK GSI1 GSI2

Configures 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.
Mode	Select protocol mode. Default: ETK (also supports XETK). Modes GSI1 and GSI2 used for DSPACE GSI with either one or two data address breakpoints reserved for other tool.

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

NOTE:	<ul style="list-style-type: none">Supported by: LA-2708, LA-3736 (JTAG Debugger for MPC5xxx Automotive) LA-7630 (NEXUS Debug/Trace for MPC5xxx/SPC5xxx) LA-7610 (NEXUS Debugger and Trace for MPC5500)Not supported by: LA-7753 (JTAG Debugger MPC5xxx/SPC5xxx) LA-7612 (NEXUS Debugger and Trace for MPC551x)
--------------	---

SYSystem.CPU

Select the target processor

Format:	SYSystem.CPU <cpu_name>
<cpu_name>:	MPC55XX MPC5554 ... <cpu_name_with_wildcards>

Selects the target processor or target core.

If you are unsure about the processor, try **SYSystem.DETECT CPU** for automatic detection.

SYStem.LOCK

Lock and tristate the debug port

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

Default: OFF.

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

SYStem.MemAccess

Select run-time memory access method

Format:	SYStem.MemAccess <mode>
<mode>:	Denied Enable StopAndGo NEXUS XCP

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 run-time memory access has to be activated for each window by using the access class E: (e.g. [Data.dump](#) E:0x100) or by using the format option %E (e.g. [Var.View](#) %E var1). It is also possible to activate this non-intrusive memory access for all memory ranges displayed on the TRACE32 screen by setting [SYStem.Option.DUALPORT ON](#).

Denied	Memory access is disabled while the CPU is executing code.
Enable	Memory access is enabled while the CPU is executing code. Available for Instruction set simulators and virtual targets (MCD).
NEXUS	<p>Memory access is done via the on-chip NEXUS block. This option is available for both the NEXUS and JTAG-only debugger solution. Memory accesses via the NEXUS block can not snoop caches.</p> <p>MPC55XX/56XX, SPC56X: While the core is running, data in cache can not be modified with this access. If the cache operates in copy-back mode, reading cached data is also not possible.</p> <p>MPC57XX/SPC57X/SPC58X: Data cache is write-through, so reading cached data is always possible, updating cached contents also is supported on most of these processors.</p> <p>MPC5777C: See SYStem.Option.NexusMemoryCoherency</p>

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.

XCP

Only available for [software-only debugging \(HostMCI\)](#).

Memory access is done via a built-in memory access feature of the 3rd party XCP slave. This memory access method is faster than performing the memory access by sending JTAG commands to the XCP slave, but the behavior (e.g. in terms of cache coherency) may differ depending on the XCP slave hardware, firmware or configuration.

NOTE:

- These processors do not support run-time memory access via NEXUS: MPC5601D, MPC5602D, MPC5601P, MPC5602P

Format: **SYStem.Mode** <mode>

SYStem.Attach (alias for SYStem.Mode Attach)

SYStem.Down (alias for SYStem.Mode Down)

SYStem.Up (alias for SYStem.Mode Up)

<mode>: **Down | NoDebug | Prepare | Go | Attach | StandBy | Up**

Selects how the debugger connects to the processor and performs the selected connection.

Down	Disables the debugger. The state of the processor remains unchanged.
NoDebug	Resets the target with debug mode disabled. In this mode no debugging is possible. The processor state keeps in the state of NoDebug.
Prepare	Connect to processor while core is unpowered. Available only for emulation devices. PREPARE mode is used to initialize, read or write emulation memory on the buddy die, before the production die is powered.
Go	Resets the target with debug mode enabled and prepares the processor for debug mode entry. Now, the processor can be stopped with the break command or any break condition.
Attach	Connect to the processor without resetting target/processor. Use this command to connect to the processor without changing it's current state.
StandBy	Debugging/Tracing through power cycles. The debugger will wait until power-on is detected, then bring the processor into debug mode, set all debug and trace registers and start the processor. In order to halt the processor at the first instruction, place a temporary on-chip breakpoint range (Break 0--0xFFFFFFFFC /Onchip)
Up	Resets the target/processor and sets the processor to debug mode. After execution of this command the processor is stopped and prepared for debugging.

SYStem.Option.BISTRUN

Debug with BIST enabled

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

MPC5777C only. By default (OFF), the processor will disable BIST if it detects that a debugger is connected while reset is asserted.

If set to ON, the debugger will connect to the processor only after reset is released. This mode impacts the debugger’s ability to debug and trace the processor from reset and through power cycles.

SYStem.Option.CoreStandBy

On-the-fly breakpoint and trace setup

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

On multi-core processors, only one of the cores starts to execute code right after reset. The other cores remain in reset or disabled state. In this state it is not possible to set breakpoints or configure the core for tracing. This option works around this limitation and makes breakpoints and tracing available on these cores.

NOTE:

This option is not required for MPC5676R when operated in SMP mode or when **SYnch.MasterSystemMode** and **SYnch.SlaveSystemMode** are set to **ON** in AMP debugging mode.

SYStem.Option.DCFREEZE

Data cache state frozen while core halted

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

Default: ON. This command configures how the debugger will maintain cache coherence for the debugger’s memory accesses while the core is halted in debug mode. The setting has no impact on the run-time memory access.

If ON, the debugger will maintain cache coherency by reading or writing directly to the cache arrays and memory. This method guarantees that the cache tags and status bits (valid, dirty) of remain unaffected by the memory accesses of the debugger. The debugger will also maintain cache coherency if the memory access is done through the NEXUS block (access class “E:”, or **SYSystem.Option.DUALPORT ON**) is used while the core is halted. This is the recommended setting.

If OFF, the debugger will maintain cache coherency by allowing the data/unified cache to be updated while reading or writing memory through debug commands. Cache coherency is **not** maintained if memory is accessed through the NEXUS block (access class “E:”, or **SYSystem.Option.DUALPORT ON**).

Setting DCFREEZE to OFF is reserved for a specific use case which requires an optimal data throughput while maintaining cache coherency (while core is halted). Do not to set DCFREEZE to OFF unless advised by Lauterbach.

SYSystem.Option.DCREAD

Read from data cache

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

Default: ON.

If enabled, **Data.dump** windows for access class D: (data) and variable windows display the memory values from the d-cache, if valid. If data is not available in cache, physical memory will be read.

SYSystem.Option.DISableResetEscalation

Control reset escalation disabling

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

Default: ON.

A processor that implements the *reset escalation feature* disable itself after a certain number of resets. Once a processor is fully escalated, a power cycle is required to regain debug access to the cores. The debugger disables the reset escalation by default, to facilitate the debug and development process. In order to test the behavior of the *reset escalation* in the application, set this option to OFF.

Please note that debugger-generated resets (e.g. **SYSystem.Up**) also contribute to the number of resets that trigger the *reset escalation*.

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

Some processors support a feature called short reset sequence, which is enabled through the RGM_FESS register. When the short reset sequence is enabled, a part of the reset phases (e.g. the BIST) are skipped at reset (including reset asserted by the debugger, e.g. for **SYStem.Up**).

Having an incomplete reset can cause problems to the debugger, for example, flash programming can fail. Therefore, by default setting (ON), the debugger disable the short reset sequence for external and JTAG resets.

Some boot loaders can not cope with the debugger’s behavior, because they blindly assume that the short sequence is enabled without checking the actual setting of the RGM_FESS register. This can cause the application to crash (e.g. by accessing uninitialized SRAM).

The ideal solution is to modify the boot loader so that it evaluates RGM_FESS when deciding if the short sequence is enabled or not.

Set this option to OFF for boot loaders which have problems when the debugger disables the short sequence. The target will require a power cycle in order to recover from the debugger’s intervention.

Another use case of this option is to debug the reset scenario with short sequence enabled.

NOTE:	<ul style="list-style-type: none">• The debugger will print a warning to the message area / status line when SYStem.Up is performed when this option is set to OFF and short sequence is enabled.• If you should experience debugging / flash programming problems while this option is set to OFF, you have to turn it ON again and perform another SYStem.Up.
--------------	--

Format:	SYStem.Option.DisMode <mode>
<mode>:	ACCESS AUTO FLE VLE

MPC5XXX/SPC5XX with VLE instruction set support only.

This command sets the operation mode for the disassembler.

AUTO (default)	The information provided by the compiler output file is used for the disassembler selection. If no information is available, it has the same behavior as the option ACCESS.
ACCESS	The operation mode for the disassembler is based on the current mode of the CPU.
FLE	Use standard PowerPC instruction set disassembler mode (fixed length encoding) only.
VLE	Use VLE disassembler mode (variable length encoding) only.

SYStem.Option.DUALPORT

Implicitly use run-time memory access

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

Forces all list, dump and view windows to use the access class **E**: (e.g. **Data.dump E:0x100**) or to use the format option **%E** (e.g. **Var.View %E var1**) without being specified. Use this option if you want all windows to be updated while the processor is executing code. This setting has no effect if **SYStem.Option.MemAccess** is disabled or real-time memory access not available for used CPU.

Please note that while the CPU is running, MMU address translation can not be accesses by the debugger. Only physical addresses accesses are possible. Use the access class modifier "A:" to declare the access physical addressed, or declare the address translation in the debugger-based MMU manually using **TRANSlation.Create**.

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

FASTACCESS is a special operation mode that allows fast run control and response times, but with very limited features. The allowed debug actions are limited to below list:

- Go
- Break
- Set/Clear breakpoints and watchpoints (Break.Set / Break.Delete)
- Write to memory using physical addresses (access class "A:")
- Write debug registers DBCR*, DBCNT, IAC*, DAC*, DVC*, DEVENT, DDAM via SPR:
- Read and Write access class DBG:

This feature is not available for all processors.

SYStem.Option.FREEZE

Freeze system timers on debug events

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

Enabling this option will lead the debugger to set the FT bit in the DBCR0 register. This bit will lead the CPU to stop the system timers (TBU/TBL and DEC) upon all debug events, that can be defined in DBCR0. The system timers will not be frozen on events like EVTI or the brkpt instruction. The timers/clocks or watchdogs of the on-chip peripherals are not affected by this option, but often can be configured to stop in debug mode by a FREEZE bit. For details please see the processor reference manual.

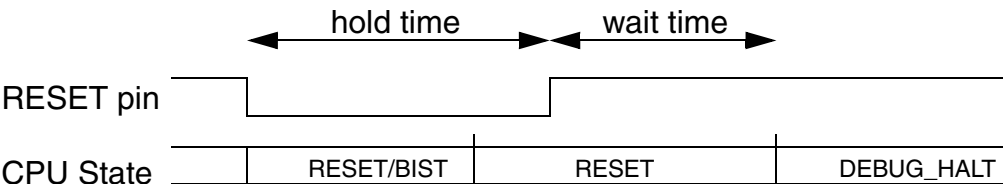
Format:

SYSystem.Option.HoldReset [*<time>*]

<time>:

1us ... 10s

Set the time that the debugger will drive the reset pin LOW, e.g. at **SYSystem.Up**. The time must be longer than the BIST takes to complete. If called without parameter, the default reset hold time is used. The default reset hold time is 100ms for processors that require a BIST delay, else 100us.



See also **SYSystem.Option.WaitReset** and **SYSystem.Option.SLOWRESET**.

SYSystem.Option.ICFLUSH

Invalidate instruction cache before go and step

Format:

SYSystem.Option.ICFLUSH [ON | OFF]

Only for cores with dedicated instruction cache (not for unified cache of e200z6 cores).

Default: ON.

Invalidates the instruction cache before starting the target program (Step or Go). If this option is disabled, the debugger will update Memory and instruction cache for program memory downloads, modifications and breakpoints. Disabling this option might cause performance decrease on memory accesses.

SYSystem.Option.ICREAD

Read from instruction cache

Format:

SYSystem.Option.ICREAD [ON | OFF]

Default: OFF:

If enabled, **List.auto** window and **Data.dump** window for access class P: (program memory) display the memory values from the instruction/unified cache if valid. If the data is not available in cache, the physical memory will be displayed.

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.

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.

NOTE:

Do not enable this option for code that disables MSR_EE. The debugger will disable MSR_EE while the CPU is running and restore it after the CPU stopped. If a part of the application is executed that disables MSE_EE, the debugger cannot detect this change and will restore MSE_EE.

SYStem.Option.KEYCODE

Inhibit censorship protection

Format:

SYStem.Option.KEYCODE [<password>]

<password>:

<64_bit_serial_password> | <p0> <p1> <p2> <p3> <p4> <p5> <p6> <p7>

Use this command to inhibit the censorship protection. The processor will then be unlocked during the next start of the debug session (**SYStem.Up**, **SYStem.Mode.Attach**, **SYStem.Mode.StandBy** etc.).

(no password)

Calling **SYStem.Option.KEYCODE** without parameters disables the censorship inhibit feature.

<64_bit_serial_
password>

For 64-bit password.

<p0> ... <p7>

MPC57XX/SPC57X/SPC58X: 8x32bit for 256-bit password.

If the processor implements a 256-bit password, the password has to be provided in **8x 32-bit** chunks. The first chunk is the most significant, i.e. it is the first chunk to be shifted into the inhibit register. The order of the 32-bit values matched the address order in FLASH.

NOTE:

- MPC55XX processors do not support censorship inhibit via JTAG.
- MPC57XX, SPC57X/SPC58X processors do not support the censorship unlock while reset is asserted. This affects power-on reset and, depending on the life cycle, also functional reset. See [Censorship Unlock](#) for available workarounds.
- Devices with C90LC Flash require the upper and lower DWORD exchanged when specified as parameter of **SYStem.Option.KEYCODE**.
- MPC577xK (RaceRunner) only:
If no password is set, the debugger automatically unlocks the processor using the public password (0xDEADDEEDFADEBADE)

Format:	SYStem.Option.LPMDebug <i><method></i>
<i><method></i> :	OFF HANDSHAKE PASSIVE ACTIVE

OFF	<p>Low power mode debugging not supported.</p> <p>At LPM entry the error message “emulation debug port fail” is generated and the communication between the debugger and the processor is lost.</p>
PASSIVE	<p>TRACE32 tries to detect low power mode entries and exits without using the LPM handshake provided by the processor.</p> <p>At LPM entry the communication between the debugger and the processor is lost. The state of the debugger changes to “running(lpm-stop/lpm-sleep)”. At LPM exit the communication between the debugger and the processor is re-established (attach). The breakpoints and the NEXUS settings are lost.</p>
ACTIVE	<p>TRACE32 tries to detect low power mode entries and exits without using the LPM handshake provided by the processor.</p> <p>At LPM entry the communication between the debugger and the processor is lost. The state of the debugger changes to “running(lpm-stop/lpm-sleep)”. At LPM exit the communication between the debugger and the processor is re-established (attach) and the program execution is stopped. The breakpoints and the NEXUS settings are re-established.</p>
HANDSHAKE	<p>TRACE32 uses LPM handshake for low power mode debugging. The processor signals LPM entries and exits to the debugger.</p> <p>At LPM exit breakpoints and NEXUS settings are re-established.</p>

NOTE:

- Enable this option if the target application makes use of low power modes and a debug port fail occurs when LPM is entered.
- If a handshaked LPM occurs, the debugger will display this state by printing “running (lpm-stop/lpm-sleep)” to the status line. If the debugger prints “running (sleep/stop/wait)”, the LPM handshake was not performed (either because not enabled or not initiated by processor).
- The LPM debug handshake is a request - acknowledge system - it has impact on the real-time behavior.
- Please check the processor’s reference manual about how the LPM debug handshake works for your specific device.
- If the processor tristates TDO during LPM, TDO must be pulled HIGH on the target. Recommended: 10k pull-up on TDO. (MPC5510: Use pull-down instead of pull-up)
- During a handshaked low-power mode, it is not possible to break (no JTAG communication possible during LPM)
- In some cases it is not possible to attach to a processor in LPM. The debugger has to assert reset in order to connect. In this case, the debugger will print a message to the message AREA window.
- Please check the processor’s device errata if there are any issues with LPM. On some devices, debug and trace settings are lost after LPM exit. On some devices, the LPM debug handshake only works if the NEXUS port is enabled. Some devices can hang at reset when LPM debug handshake is enabled.
- MPC5748G, MPC5746C (Calypso): PASS_LCSTAT[CNS] must be zero in order to use the LPM handshake.

SYStem.Option.LockStepDebug

Enable lock-step core register access

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

Enables read and write access to the core registers of the lock-step core. Only available for MPC564xL, SPC56EL, MPC567xK and SPC56HK. This feature is not available for processors that implement delayed lock-step (MPC57XX/SPC57X/SPC58X).

SYStem.Option.MMUSPACES

Separate address spaces by space IDs

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

Default: OFF.

Enables the use of [space IDs](#) for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of [address spaces](#) ([zone spaces](#), [MMU spaces](#), and [machine spaces](#)), see **“TRACE32 Concepts”** ([trace32_concepts.pdf](#)).

NOTE:

SYStem.Option.MMUSPACES should not be set to **ON** if only one translation table is used on the target.

If a debug session requires space IDs, you must observe the following sequence of steps:

1. Activate **SYStem.Option.MMUSPACES**.
2. Load the symbols with [Data.LOAD](#).

Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

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

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

SYStem.Option.NexusMemoryCoherency

Coherent NEXUS mem-access

MPC5676R and MPC5777C only

Format:

SYStem.Option.NexusMemoryCoherency [ON | OFF]

If this option is set to ON, the debugger configures the NEXUS run-time memory access to assert the signal `p_d_gbl` for each access. This signal will cause the Cache Coherency Unit to perform a cache snoop for the run-time memory access, which allows the debugger to update SRAM while maintaining cache coherency.

It is essential to set this option to ON *only* if the data cache is configured to write-through mode (`L1CSR0[DCWM]==1`). If the cache is operated in copy-back mode, setting this option to ON can cause undefined behavior.

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

Default: OFF.

On-chip debug events that cause a debug interrupt can be configured to cause one of two actions. If a JTAG debugger is used, the CPU is configured to stop for JTAG upon these debug events.

If this option is set to ON, the CPU will be configured to not stop for JTAG, but to enter the debug interrupt, like it does when no JTAG debugger is used.

Enable this option if the CPU should not stop for JTAG on debug events, in order to allow a target application to use the debug interrupt. Typical usages for this option are run-mode debugging (e.g. with t32server/gdbserver) or setting up the system for a branch trace via LOGGER (trace data in target RAM) or INTEGRATOR.

SYStem.Option.NoJtagRdy**Do not evaluate JTAG_RDY signal**

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

The JTAG_RDY pin is an output of the CPU to signal the debugger when memory accesses, done via the NEXUS block, are finished. Memory accesses via the NEXUS block are possible with the NEXUS debugger and trace, but also with the JTAG-only debugger. The existence of the JTAG_RDY signal depends on processor type and package size.

If this option is OFF (default), the debugger will use the JTAG_RDY signal for memory accesses via the NEXUS block. If the option is ON, the debugger will ignore JTAG_RDY signal.

If the used processor type does not provide JTAG_RDY in any package size (e.g. MPX551X), the debugger will automatically set this option. If availability of the signal depends on the package size, or if the signal is available but not connected, use this option to configure the debugger manually.

In Software since October 2007, the debugger will automatically probe the JTAG_RDY pin. On targets without JTAG_RDY signal, the JTAG_RDY pin of the debug connected should be connected to GND. If the JTAG_RDY pin is left unconnected, use **SYStem.Option.NoJtagRdy** ON to prevent problems with probing this signal.

Format:

SYStem.Option.NOTRAP [ON | OFF]

This system option configures if the TRAP instruction is used for external (JTAG/NEXUS) debug events. By default (OFF) TRAP is enabled as debug event in the (E)DBCR0 register.

e200z0	If ON, the BRKPT instruction is used instead of TRAP. In this case, the freeze timer option is ineffective.
e200z0H	
e200z0Hn2P	
e200z1	
e200z3	
e200z335	
e200z336	
e200z6	
e200z750	
all other cores	If ON, TRAP instruction is not treated as debug event. The Debugger always uses the DNH instruction for software breakpoints regardless of this setting.

Format:	SYSystem.Option.OVERLAY [ON OFF WithOVS]
---------	---

Default: OFF.

ON	Activates the overlay extension and extends the address scheme of the debugger with a 16 bit virtual overlay ID. Addresses therefore have the format <i><overlay_id>:<address></i> . This enables the debugger to handle overlaid program memory.
OFF	Disables support for code overlays.
WithOVS	Like option ON , but also enables support for software breakpoints. This means that TRACE32 writes software breakpoint opcodes to both, the <i>execution area</i> (for active overlays) and the <i>storage area</i> . This way, it is possible to set breakpoints into inactive overlays. Upon activation of the overlay, the target's runtime mechanisms copies the breakpoint opcodes to the execution area. For using this option, the storage area must be readable and writable for the debugger.

Example:

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

Format:	SYSystem.Option.PC <address> AUTO
---------	---

The debugger has to set a fetch address while accessing core resources. Per default (parameter AUTO), the debugger places the fetch address into the BAM space. If the BAM is disabled or access protected by the SOC-MPU (core MMU and MPU have no effect), use this command to define an address which the core is allowed to fetch.

When setting a fetch address, make sure that the address does not cause an instruction storage exception, e.g. because of unimplemented memory or ECC errors (e.g. in FLASH or uninitialized SRAM).

Format:	SYStem.Option.RESetBehavior <mode>
<mode>:	Disabled AsyncHalt AsyncStart ResetHalt ResetStart RESYNC

Defines the debugger's action when a reset is detected. Default setting is **ResetHalt**. If and how a reset can be detected is set using **SYStem.Option.ResetDetection**. This option is usually used for MPC55XX and some 56XX to restore breakpoints after a reset. Usually not required for MPC57XX/SPC57X and SPC58X.

Disabled	No actions to the processor take place when a reset is detected. Information about the reset will be printed to the message AREA .
AsyncHalt	Halt core as soon as possible after reset was detected. The core will halt shortly after the reset event.
AsyncStart	Halt core as soon as possible after reset was detected. The debugger sets debug and trace configuration registers and afterwards starts the core(s) again.
ResetHalt	When a reset is detected, the debugger keeps reset asserted and then halts the core at the reset address.
ResetStart	When a reset is detected, the debugger keeps reset asserted and then halts the core at the reset address. The debugger sets debug and trace configuration registers and afterwards starts the core(s) again.
RESYNC	When a reset is detected, the debugger waits until reset is released. Once the core is out of reset, the debugger sets debug and trace configuration registers on-the-fly.

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

Default: ON.

If **SYStem.Up** is called using the default setting **SYStem.Option.ResBreak** ON, the debugger will assert reset and send a halt command to the core while reset is asserted. If the processor is censored and a password is supplied, the debugger will also unlock the processor while reset is asserted. This method ensures that the debugger can halt the core directly at the reset address.

Some processors of the MPC57XX/SPC57X/SPC58X series have a bug causing the censorship unlock to fail if it is done while reset is asserted. This bug collides with the debugger's default SYStem.Up sequence. Processors known to have this bug are: MPC574xB/C/D/G (Calypso), MPC5777C (Cobra55), MPC5777M (Matterhorn), SPC58NE (Eiger).

In order to connect to above censored processors, set **SYStem.Option.ResBreak** to OFF. With this setting, the debugger will assert and release RESET. After a delay defined using **SYStem.Option.WaitReset**, the debugger will unlock the processor and halt the core. The longer the defined WaitReset time, the more program code is executed until the core can be halted. If the WaitReset time is too short, SYStem.Up will fail.

SYStem.Option.ResetDetection

Configure reset detection method

Format:	SYStem.Option.ResetDetection <method> SYStem.Option.RSTOUT [ON OFF] (deprecated)
<method>:	OFF RESETPIN RSTINOUT

Default: OFF. This option configures if the debugger's reset detection is enabled and if enabled, which signals are used to detect reset.

If reset detection is enabled and a reset is detected, the debugger will perform the action selected with **SYStem.Option.RESetBehavior**.

This feature is important for processors of the MPC55XX/MPC560X/SPC560X series, which clear debug and trace registers upon reset. For newer processors, which don't clear debug and trace register upon reset, this option can be set to off, unless any other tool connected to the target asserts JCOMP when it detects a target reset. See chapter **Debugging and Tracing Through Reset** for details.

Processors which require the PCRs to be configured by the debugger for tracing, reset detection has to be enabled in order to enable tracing through reset.

<method>	Function
OFF	Reset detection is disabled.

RESETPIN	Debugger observes only RSTIN for reset detection.
RSTINOUT	<p>Debugger observes RSTIN and RSTOUT for reset detection. Use only:</p> <ul style="list-style-type: none"> • if Processor has RSTOUT pin • if RSTOUT pin is configured to signal core resets • if RSTOUT pin is connected to debug/trace connector • with following debug/trace modules: LA-2708, LA-3736 (JTAG Debugger for MPC5xxx Automotive) LA-7630 (NEXUS Debug/Trace for Qorivva MPC5xxx/SPC5xxx) LA-7610 (NEXUS Debugger and Trace for MPC5500)

SYStem.Option.ResetMode Select reset mode for SYStem.Up

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

<mode>: **PIN | DESTructive | FUNCtional**

Default: PIN. Selects the method the debugger uses to reset the processor. Only available for MPC57XX/SPC57X processors with *Debug and Calibration Interface* (DCI)..

<i><mode></i>	Effect at SYStem.Up
PIN	The reset pin is asserted to reset the processor. This can result in either a functional or destructive reset, depending on which reset pin of the processor is connected to the debug / trace connector. This is the only method available for processors without DCI module.
DESTructive	The debugger performs a destructive processor reset using the DCI module. The reset pin of the debug/trace connector is asserted as well.
FUNCtional	The debugger performs a functional reset using the DCI module. The reset pin of the debug/trace connector is not asserted.

SYStem.Option.SLOWRESET Relaxed reset timing

Format: **SYStem.Option.SLOWRESET** [ON | OFF] (deprecated)

Default: OFF. Set to ON to use a relaxed reset timing for processors with BIST enabled, or when debugger is used with processor emulation systems. Deprecated, use **SYStem.Option.HoldReset** and **SYStem.Option.WaitReset** instead.

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

This method uses software breakpoints to perform an assembler single step instead of the processor's built-in single step feature. Works only for software in RAM. Do not turn ON unless advised by Lauterbach.

SYSystem.Option.TDOSElect

Select TDO source of lock step core pair

Format:	SYSystem.Option.TDOSElect [A B]
---------	--

If the processor consists of a lock-step core pair, this option defines, which core's TDO signal is routed to the TDO pin of the processor. Can be useful for debugging lock-step related application issues. Not available for cores in delayed lock-step. This setting should only be changed before starting the debug session, or at least while the core is running.

SYSystem.Option.VECTORS

Specify interrupt vector table address

Format:	SYSystem.Option.VECTORS <range> [<range> ...]
---------	---

Only required for MPC5553 and MPC5554. Not required for other processors.

On MPC5553/4, indirect branch messages do not indicate if the reason was an indirect branch or an interrupt. If the address range of the interrupt vectors are specified by this command, the TRACE32 NEXUS debugger marks all indirect branches to these addresses / the address range as interrupt. This information is needed for correct trace display and run-time statistic analysis.

Valid parameters for this command are addresses, address ranges, debug symbols and also ranges of debug symbols. Examples:

single addresses	<code>SYSystem.Option.VECTORS 0x40002000 0x40002100</code>
address range	<code>SYSystem.Option.VECTORS 0x40002000--0x40002FFF</code>
(2 methods)	<code>SYSystem.Option.VECTORS 0x40002000++0x00002FFF</code>

use IVPR/IVOR registers	&startaddr=R(IVPR) &range=DATA.LONG(SPR:415.) SYStem.Option.VECTORS &startaddr++&range ;IVPR/IVOR must already be initialized ;&range = MAX(IVOR0..IVOR34)
use debug symbols	SYStem.Option.VECTORS IVOR0_func IVOR1_func ...
use debug symbol range	SYStem.Option.VECTORS IVOR0_fnc--(IVOR15_fnc+3)

SYStem.Option.WaitBootRom

Wait for BootROM completion

[build 144848 - DVD 09/2022]

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

Default: ON.

If the debug session starts (e.g. **SYStem.Up**), by default, the debugger waits until the BootROM execution has completed.

If the option is set to OFF, the debugger will not wait for the BootROM completion. Set this option to off in order to recover a password protected processor with a bad FLASH image. After the processor has been recovered, set this option to ON again. While OFF, the BootROM will not boot from the flash image.

SYStem.Option.WaitReset

Set reset wait time

Format: **SYStem.Option.WaitReset [<time> [<reference>]]**

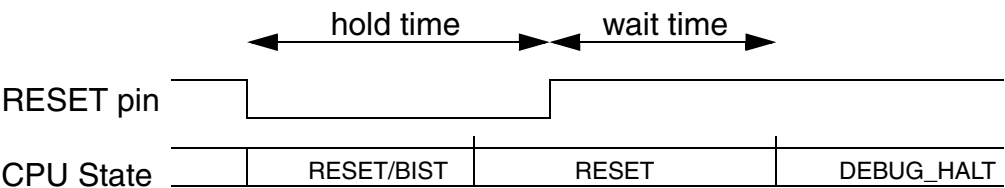
<time>: **1us...10s**

<reference>: **default
RESET
RSTOUT**

Set the time that the debugger will wait after releasing the reset pin, e.g. at **SYStem.Up**. If called without parameter, the default reset wait time is used (10us).

If the reference is set to **default**, the wait time starts when the debugger releases reset. If the reference is set to **RESET** or **RSTOUT**, the wait time starts when the debugger detects that reset is released on the corresponding pin.

Use this command when **SYSystem.Up** fails, and the message **AREA** shows the message “Target reset detected during system.up sequence”. A wait time of several ms should be sufficient. If a wait time > 10ms is required, the target might require a stronger RESET pull-up resistor.



For related commands, see also **SYSystem.Option.HoldReset** and **SYSystem.Option.SLOWRESET**. See chapter **Censorship Unlock** for typical use cases of this command.

Format:	SYStem.Option.WATCHDOG <i><method></i>
<i><method></i> :	[ON OFF PASSIVE]

Defined how the debugger is handling the on-chip software watchdog timer. Default: OFF.

- OFF

Default setting. The watchdog timer of the processor will be disabled during **SYStem.Up** and **SYStem.Mode.StandBy**. For MPC551x, the debugger will also set the SWCTR_RO (read_only) bit in order to prevent that the watchdog timer is enabled later by the application. For MPC563X and SPC563, the debugger will only clear WEN.
If the debugger is connected using **SYStem.Mode.Go** or **SYStem.Mode.Attach**, the debugger will try to disable and the watchdog timer as soon as the processor is stopped. If the watchdog is enabled and SWCTR_RO bit is set after **SYStem.Mode.Go** or **SYStem.Mode.Attach**, **SYStem.Option.WATCHDOG OFF** cannot be used.
- ON

The state of the SWT (enabled or disabled) is not changed by the debugger. If possible the debugger will try to configure the SWT so that it does not time out while the processor is halted. See tables below for details.
- PASSIVE

The debugger does not access (read or write) the SWT registers. The target application must ensure that the SWT does not time out while the core is halted for debugging.

The table below describes how the debugger is configuring the SWT when **SYStem.Option.WATCHDOG ON** is used. The configuration takes place any time the CPU stops for the debugger according to the tables below:

MPC56XX, SPC56X, MPC57XX, SPC57X:

CR[WEN]	CR[FRZ]	CR[HLK] CR[SLK]	Debugger Action
off	don't care	don't care	none
on	on	don't care	none
on	off	SLK on HLK off	set FRZ
on	off	HLK on	service watchdog (see note)

SWE	SWRWH	RO	Debugger Action
off	don't care	don't care	none
on	off	don't care	none
on	on	off	disable SWRWH
on	on	on	service watchdog (see note)

NOTE:**All CPUs: servicing watchdog**

If the debugger is servicing the watchdog, conditions might occur, where the watchdog times out before the debugger is able to service it. Unintended resets or interrupts can occur.

Further, SWT window mode is not supported by the debugger.

MPC5516, revision 0

If the system option is ON, the debugger will configure the watchdog to the longest timeout period on SYStem.Up and SYStem.Mode.StandBy. During debugging, the watchdog timer will be serviced if SWE is on.

MMU.DUMP

Page wise display of MMU translation table

Format:

MMU.DUMP <table> [<range> | <address> | <range> <root> | <address> <root>]

<table>:

PageTable
KernelPageTable
TaskPageTable <task_magic> | <task_id> | <task_name> | <space_id>:0x0
<cpu_specific_tables>

Displays the contents of the CPU specific MMU translation table.

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

<root>	The <root> argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> <address>	<p>Limit the address range displayed to either an address range or to addresses larger or equal to <address>.</p> <p>For most table types, the arguments <range> or <address> can also be used to select the translation table of a specific process if a space ID is given.</p>
PageTable	<p>Displays the entries of an MMU translation table.</p> <ul style="list-style-type: none">• if <range> or <address> have a space ID: displays the translation table of the specified process• else, this command displays the table the CPU currently uses for MMU translation.

KernelPageTable	<p>Displays the MMU translation table of the kernel.</p> <p>If specified with the MMU.FORMAT command, this command reads the MMU translation table of the kernel and displays its table entries.</p>
TaskPageTable <code><task_magic> </code> <code><task_id> </code> <code><task_name> </code> <code><space_id>:0x0</code>	<p>Displays the MMU translation table entries of the given process. Specify one of the TaskPageTable arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries.</p> <ul style="list-style-type: none"> • For information about the first three parameters, see “What to know about the Task Parameters” (general_ref_t.pdf). • See also the appropriate OS Awareness Manuals.

TLB1	Displays the contents of TLB1.
TLB2	Displays the contents of TLB2 (MPU).

MMU.List

Compact display of MMU translation table

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

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

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

<root>	The <root> argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> <address>	Limit the address range displayed to either an address range or to addresses larger or equal to <address>. For most table types, the arguments <range> or <address> can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Lists the entries of an MMU translation table. <ul style="list-style-type: none">• if <range> or <address> have a space ID: list the translation table of the specified process• else, this command lists the table the CPU currently uses for MMU translation.

KernelPageTable	<p>Lists the MMU translation table of the kernel.</p> <p>If specified with the MMU.FORMAT command, this command reads the MMU translation table of the kernel and lists its address translation.</p>
TaskPageTable <i><task_magic></i> <i><task_id></i> <i><task_name></i> <i><space_id>:0x0</i>	<p>Lists the MMU translation of the given process. Specify one of the TaskPageTable arguments to choose the process you want.</p> <p>In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and lists its address translation.</p> <ul style="list-style-type: none"> • For information about the first three parameters, see “What to know about the Task Parameters” (general_ref_t.pdf). • See also the appropriate OS Awareness Manuals.

Format:	MMU.SCAN <table> [<range> <address>]
<table>:	PageTable KernelPageTable TaskPageTable <task_magic> <task_id> <task_name> <space_id>:0x0 ALL [Clear] <cpu_specific_tables>

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

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

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

PageTable	<div>Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table.</div> <ul style="list-style-type: none">• if <range> or <address> have a space ID: loads the translation table of the specified process• else, this command loads the table the CPU currently uses for MMU translation.
------------------	---

KernelPageTable	<p>Loads the MMU translation table of the kernel.</p> <p>If specified with the MMU.FORMAT command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.</p>
TaskPageTable <task_magic> <task_id> <task_name> <space_id>:0x0	<p>Loads the MMU address translation of the given process. Specify one of the TaskPageTable arguments to choose the process you want.</p> <p>In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table.</p> <ul style="list-style-type: none"> For information about the first three parameters, see “What to know about the Task Parameters” (general_ref_t.pdf). See also the appropriate OS Awareness Manual.
ALL [Clear]	<p>Loads all known MMU address translations.</p> <p>This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table.</p> <p>See also the appropriate OS Awareness Manual.</p> <p>Clear: This option allows to clear the static translations list before reading it from all page translation tables.</p>

CPU specific tables in MMU.SCAN <table>

TLB1	Loads the TLB1 from the CPU to the debugger-internal translation table.
TLB2	Loads the TLB2 (MPU) from the CPU to the debugger-internal translation table. Usually not necessary.

Use this command to make the translation information available for the debugger even when the program execution is running and the debugger has no access to the TLBs. This is required for the real-time memory access (See also [SYStem.MemAccess](#) and [SYStem.Option.DUALPORT](#)). Use the command [TRANSlation.ON](#) to enable the debugger-internal MMU table.

Formats:

MMU.Set TLB1 <index> <mas1> <mas2> <mas3>

MMU.Set TLB2 <index> <mas0> <mas1> <mas2>

MMU.<table>.SET (deprecated)

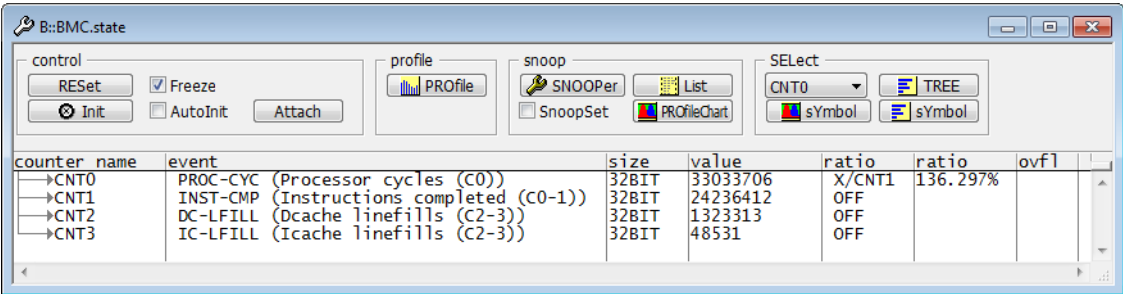
Sets the specified MMU TLB table entry in the CPU. The parameter *<tlb>* is not available for CPUs with only one TLB table.

<index>	TLB entry index. From 0 to (number of TLB entries)-1 of the specified TLB table
<mas0>	Values corresponding to the values that would be written to the MAS registers in order to set a TLB (or MPU) entry. See the processor's reference manual for details on MAS registers. For processors with a core MPU (MPC57XX/SPC57X series), use TLB2 to generate an MPU entry).
<mas1>	
<mas2>	
<mas3>	

Command Reference: BenchMarkCounter

The BenchMarkCounter features are based on the core’s performance monitor, accessed through the performance monitor registers (PMR).

TRACE32 displays the benchmark counter results in the **BMC.state** window:



- NOTE:
- These cores do **not** implement PMRs:
e200z0, e200z1, e200z3
e200z4d (MPC5643L, SPC56EL, MPC5645S, MPC564xC, SPC56xC)
e200z448 (MPC5644A, SPC564A)
e200z6 and e200z750.
 - These cores only provide PMR access while the core is halted:
e200z759, e200z760.
 - For a list and description of events that can be assigned to **BMC.<counter>.EVENT <event>**, please see the Freescale e200z core reference manuals.
 - In addition to the core defined events, TRACE32 provides events ALPHA...ECHO to count watchpoints set with **Break.Set**.

For information about *architecture-independent* **BMC** commands, refer to “**BMC**” (general_ref_b.pdf).

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

BMC.<counter>.ATOB

Enable event triggered counter start and stop

Format:

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

Enables event triggered counter start/stop. The events are defines using ALPHA and BETA breakpoints set with **Break.Set**. Every time the Alpha breakpoint condition triggers, the counter is started. The counter stops when the Beta breakpoint condition is triggered.

Example 1: Measure average processor cycles it takes from function *sieve* entry to exit. This measurement includes all interrupts, sub-function calls etc.

```
;Measure average cycles to execute unction sieve
  BMC.RESet
  Break.Delete

;set up counter start / stop events
  Break.Set sYmbol.BEGIN(sieve) /Program /Onchip /Alpha
  Break.Set sYmbol.EXIT(sieve) /Program /Onchip /Beta

;set up CNT0 to count processor cycles (using start/stop event)
  BMC.CNT0.EVENT PROC-CYC
  BMC.CNT0.ATOB ON

;set up CNT2 to count function entries
  BMC.CNT2.EVENT ALPHA

;run measurement (for 10 seconds)
  BMC.Init
  Go
  Wait 10s
  Break

PRINT FORMAT.DECIMAL(1.,BMC.COUNTER(0)/BMC.COUNTER(2))+ " cycles"
```

Example 2: Measure average processor cycles the core is inside function *sieve*. This method excludes interrupts and sub-function calls from the measurement:

```
;Measure net processor cycles to execute function sieve
  BMC.RESet
  Break.Delete

;set up counter start / stop events
  Var.Break.Set sieve /Program /Onchip /Alpha
  Var.Break.Set sieve /Program /Onchip /Beta /EXCLUDE

;set up additional BMC event to count function entries
  Break.Set sYmbol.BEGIN(sieve) /Program /Onchip /Charly

;set up CNT0 to count processor cycles (using start/stop event)
  BMC.CNT0.EVENT PROC-CYC
  BMC.CNT0.ATOB ON

;set up CNT2 to count function entries
  BMC.CNT2.EVENT CHARLY

;run measurement (for 10 seconds)
  BMC.Init
  Go
  Wait 10s
  Break

PRINT FORMAT.DECIMAL(1.,BMC.COUNTER(0)/BMC.COUNTER(2))+" cycles"
```

Example 3: Measure instructions per processor cycle for a specific task on an RTOS:

```
;Measure instructions per clock for task my_task
BMC.RESet
Break.Delete

;set up counter start / stop events
&magic=TASK.MAGIC("my_task") ;get magic value for the task of interest
Break.Set task.config(magic) /ReadWrite /Onchip /Data &magic /Alpha
Break.Set task.config(magic) /ReadWrite /Onchip /Data !&magic /Beta

;set up CNT0 to count processor cycles (using start/stop event)
BMC.CNT0.EVENT PROC-CYC
BMC.CNT0.ATOB ON

;set up CNT1 to count instructions executed (using start/stop event)
BMC.CNT1.EVENT INST-CMP
BMC.CNT1.ATOB ON

;set up ratio for BMC.state window display
BMC.CNT1.RATIO X/CNT0
BMC.state

;run measurement (for 10 second)
BMC.Init
Go
Wait 10s
Break

PRINT FORMAT.FLOAT(5.,2.,1.0*BMC.COUNTER(1)/BMC.COUNTER(0))+ " IPC"
```

BMC.<counter>.FREEZE

Freeze counter in certain core states

Format:	BMC.<counter>.FREEZE <state> [ON OFF]
<state>:	USER SUPERVISOR MASKSET MASKCLEAR

Halts the selected performance counter if one or more of the enabled states (i.e. states set to ON) match the current state of the core. If contradicting states are enabled (e.g. SUPERVISOR and USER), the counter will be permanently frozen. The table below explains the meaning of the individual states.

<state>	Dependency in core
USER	Counter frozen if MSR[PR]==1
SUPERVISOR	Counter frozen if MSR[PR]==0

MASKSET Counter frozen if MSR[PMM]==1

MASKCLEAR Counter frozen if MSR[PMM]==0

BMC.FREEZE

Freeze counters while core halted

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

On MPC5XXX, the core performance counters automatically stop when a core enters debug mode. Therefore this command has no effect.

BMC.Trace

Trace performance monitor events

Format: **BMC.Trace [ON | OFF] <periodicity>**

<periodicity>: **2^0 | 2^1 | 2^4 | 2^8 | 2^14 | 2^20 | 2^31**

This feature configured the processor to generate watchpoint hit messages upon performance monitor events. The frequency of the watchpoint messages can be controlled with the <periodicity> parameter. If <periodicity> is e.g. set to 2^8, the processor will generate a watchpoint hit message every 256 events.

This feature can help to improve time resolution on processors that do not support on-chip time-stamp generation.

TrOnchip.CONVert

Adjust range breakpoint in on-chip resource

Format:

TrOnchip.CONVert [ON | OFF]

There are 2 data address breakpoints. These breakpoints can be used to mark two single data addresses or one data address range.

- ON (default)

After a data address breakpoint is set to an address range all on-chip breakpoints are spent. As soon as a new data address breakpoint is set the data address breakpoint to the address range is converted to a single data address breakpoint. Please be aware, that the breakpoint is still listed as a range breakpoint in the [Break.List](#) window. Use the [Data.View](#) command to verify the set data address breakpoints.
- OFF

An error message is displayed when the user wants to set a new data address breakpoint after all on-chip breakpoints are spent by a data address breakpoint to an address range.

```
TrOnchip.CONVert ON
Break.Set 0x6020++0x1f
Break.Set 0x7400++0x3f
Data.View 0x6020
Data.View 0x7400
```


Format:

TrOnchip.EDBRAC0

<edbrac0>

<dirconfig>

TrOnchip.DBERC0

<edbrac0>

Use this command to assign which debug resources are reserved for debugger use and which resources should be used by the by the target application. On MPC56XX/SPC56XX, the assignment is done using the DBERC0 register, but on MPC57XX/SPC57XX the functionality was moved to EDBRAC0.

- <edbrac0>

Value written to EDBRAC0 or DBERC0 register. Default: 0x00000008

The selected events are excluded from debugger use (via [Break.Set](#)) and assigned for use by the target application.
- <dirconfig>

Default: 0x00000000

The selected events are excluded from debugger use (via [Break.Set](#)) and are available for direct register configuration through debugger writes, e.g. [Data.Set](#).

See core reference manual for the EDBRAC0/DBERC0 bit definitions.

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

When enabled, the processor is configured to enable the EVTI/EVTO pins. If disabled, that pins can be used for GPIO. (Default: ON)

NOTE:	<ul style="list-style-type: none">• This option sets the EVT_EN bit in the NPC_PCR register. It is not available on all processor. Please check the processor reference manual for availability.• On MPC551X, set this option to OFF when the EVTx pins are used for the EBI (External Bus Interface).• If the EVTx pins are not used for EVTI/EVTO, they should be disconnected from the debug/trace connector to avoid additional load, signal reflections etc.• LA-7610 and LA-7612 only: If the EVTx pins are not used for EVTI/EVTO, they must not be connected to the debug/trace connector.• LA-7630: EVTI pin is tristated when TrOnchip.EVTEN is OFF.
-------	--

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

Resets the on-chip trigger system to the default state.

TrOnchip.Set

Enable special on-chip breakpoints

Format:	TrOnchip.Set <i><event></i> [ON OFF]
<i><event></i> :	BRT IRPT RET CIRPT CRET BKPT

Enables the specified on-chip trigger facility to stop the core on below events. Default is OFF unless specified otherwise.

<i><event></i>	Break events, see below.
BRT	Branch taken.
IRPT	Interrupt entry.
RET	Return from interrupt.
CIRPT	Critical interrupt entry.
CRET	Critical interrupt return.
BKPT (Default: ON)	Execution of the BKPT pseudo-opcode. Please note that this opcode represents the software breakpoint for e200z750, e200z6, e200z3, e200z1 and e200z0Hn2 cores when operated in VLE mode.

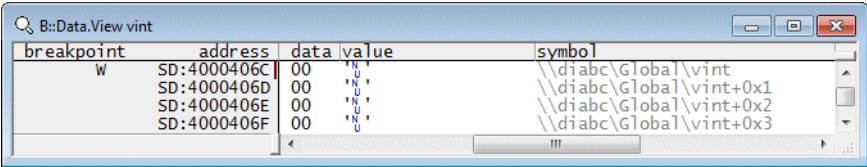
Format:

TrOnchip.VarCONVert [ON | OFF]

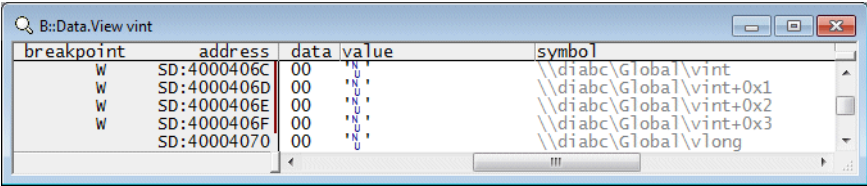
Default: ON.

OFF	<p>If a breakpoint is set to a scalar variable (int, float, double) breakpoints are set to all memory addresses that store the variable value.</p> <p>+ The program execution stops also on any unintentional accesses to the variable's address space.</p> <p>- Requires two onchip breakpoints since a range breakpoint is used.</p>
ON	<p>If a breakpoint is set to a scalar variable (int, float, double) the breakpoint is set to the start address of the variable.</p> <p>+ Requires only one single address breakpoint.</p> <p>- Program will not stop on unintentional accesses to the variable's address space.</p>

```
TrOnchip.VarCONVert ON
Var.Break.Set vint /Write
Data.View vint
```



```
TrOnchip.VarCONVert OFF
Var.Break.Set vint /Write
Data.View vint
```



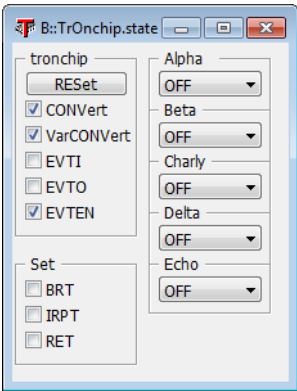
Format:

TrOnchip.state

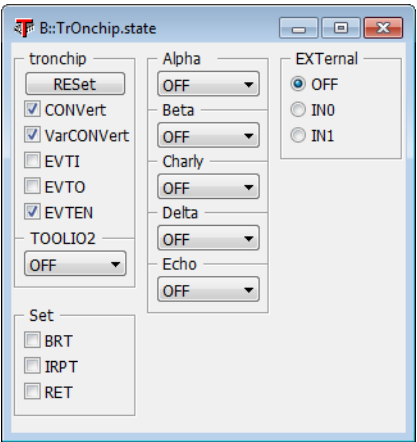
Displays the **TrOnchip.state** window for on-chip trigger setup.

Different commands are available in the **TrOnchip.state** window, depending on the Lauterbach hardware used:

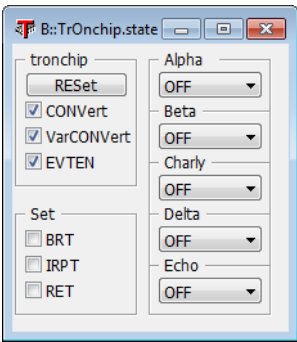
Debug cable LA-2708, LA-3736 (AUTO26) pinout)



NEXUS adapter LA-7610/12/30



Debug cable LA-7753 (JTAG/OnCE pinout)



Onchip.TBARange

Set on-chip trace buffer address range

Format:	Onchip.TBARange <access>:<range>
<access>:	A EEC

Sets the address space and size of the on-chip NEXUS trace buffer. This setting depends on the target processor in use. See [MPC57XX/SPC57X on-chip trace](#) for details about possible address ranges.

- A** Default address space, e.g. production device.
- EEC** Address space of emulation device trace buffer.
- <range> Address range of trace buffer, e.g. 0xD000000--0xD003FFF

Example for MPC5777M: automatic trace setup depending on connected debug tool and processor:

```
SYStem.DEECT CPU
IF CHIP.EmulationDevice()
(
  IF POWERTRACE()&&!POWERNEXUS()
  (
    ;use AURORA Nexus trace
    &all NEXUS.PortSize 4Lane
    &all NEXUS.PortMode 1250Mbps
    &all NEXUS.RefClock ON
    &all Trace.Method Analyzer
  )
  ELSE
  (
    ;use on-chip trace buffer of buddy die
    Onchip.TBARange EEC:0x0C000000--0x0C1FFFFF
    &all Trace.Method Onchip
  )
)
ELSE
(
  ;use on-chip trace buffer of production device
  Onchip.TBARange A:0x0D000000--0x0D003FFF
  &all Trace.Method Onchip
)
```

NEXUS.BTM

Enable program trace messaging

Format:

NEXUS.BTM [ON | OFF]
SYStem.Option.BTM [ON | OFF] (deprecated)

Control for NEXUS program trace messaging.

- ON (default)Program trace messaging enabled.
- OFFProgram trace messaging disabled.

NEXUS.CLIENT<x>.BUSSEL

Set NXMC target RAM

Format:

NEXUS.CLIENT1.BUSSEL [PRAM0 | PRAM1 | PRAM2]

MPC574xG only. Select the target RAM for which the NXMC should generate messages.

NEXUS.CLIENT<x>.MODE

Set data trace mode of nexus client

Format:

NEXUS.CLIENT1.MODE [Read | Write | ReadWrite | OFF]
NEXUS.CLIENT2.MODE [Read | Write | ReadWrite | OFF]
NEXUS.CLIENT3.MODE [Read | Write | ReadWrite | OFF]

Sets the data trace mode of the selected trace client. Select the trace client using **NEXUS.CLIENT<x>.SELECT** before setting the trace mode.

Format:

NEXUS.CLIENT1.SELECT <client>
NEXUS.CLIENT2.SELECT <client>
NEXUS.CLIENT3.SELECT <client>

<client>:

(dedicated trace clients, e.g. MPC5554, MPC5674F)
DMA_0 | DMA_1 | FLEXRAY

(source selector for SRAM port sniffers, e.g. MPC564xL, MPC567xK)
ALL | CORE | NEXCORE | DMA_0 | DMA_1

Select the trace client for data tracing. For processors with dedicated trace clients (e.g. MPC5554), any trace client can be assigned to any CLIENT field.

For processors with SRAM port sniffers (e.g. MPC564xL), the port sniffers are fix assigned to CLIENT fields (NXSS0 -> CLIENT1, NXSS1 -> CLIENT2) and <client> configures the source selector of the SRAM port sniffer.

NEXUS.CLIENT3.SPTACQMASTER

Trace individual SPT masters

Format:

NEXUS.CLIENT3.SPTACQMASTER OFF | <master_id>

MPC577xK only. If set to OFF (default), the processor sends messages if all SPT acquisition accesses. If set to a specific master ID, only the ID of this master is traced.

NEXUS.CoreEnable

Enable core tracing for dedicated cores in SMP

Format:

NEXUS.CoreEnable {<logical_core>}

Core tracing is enabled for all core of an SMP system by default. The command **NEXUS.CoreEnable** allows to enable core tracing for only the logical cores specified.

The **Core pulldown** of the TRACE32 state line shows you the list of logical cores that form the SMP system.

NEXUS.CoreEnable 1. ; Enable core tracing only for
; the logical core 1. of the SMP
; system

Format:	NEXUS.DDR [ON OFF] SYStem.Option.DDR [ON OFF] (deprecated)
---------	---

Default: OFF. Set trace port and NEXUS adapter to operate in DDR (double data rate) mode.

NOTE:	<ul style="list-style-type: none">• Only supported by NEXUS AutoFocus adapter (LA-7630).• Check processor reference manual if the processor supports DDR.• Usually DDR mode is not allowed with NEXUS.PortMode 1/1 and 1/3.• On many processors (esp. MPC55XX/56XX), MCKO does not change when data is valid, but together with data. In this case it is required to move the sample point using Analyzer.SAMPLE.• If DDR mode is used with high trace port frequencies, sometimes sample point fine tuning is required (Analyzer.SAMPLE)
--------------	---

Format:	NEXUS.DMADTM [Read Write ReadWrite DTM OFF] (deprecated) SYStem.Option.DMADTM [ON OFF] (deprecated)
---------	--

Deprecated. Use [NEXUS.CLIENT<x>](#) commands.

Format:	NEXUS.DTM <i><mode></i> SYStem.Option.DTM [Read Write ReadWrite OFF] (deprecated)
<i><mode></i> :	OFF Read Write ReadWrite IFETCH ReadLimited WriteLimited ReadWriteLimited

Controls the Data Trace Messaging method.

OFF	Data trace messaging disabled (default)
Read	Data trace messages for read accesses (load instructions)
Write	Data trace messages for write accesses (store instructions)
ReadWrite	Data trace messages for read and write accesses (load and store instructions)
ReadLimited	Same as above, but excluding data accesses using GPR R1 in effective address computations. NOTE: Only supported by MPC57XX and newer processors.
WriteLimited	
ReadWriteLimited	
IFETCH	Data trace messages contain information about instruction fetches

NEXUS.DTMARK

Data trace mark

Format:	NEXUS.DTMARK [ON OFF]
---------	--------------------------------

Controls the influence of MSR[PMM] in data trace messaging. Only available on processors which implement IEEE-ISTO 5001-**2008** or later.

OFF	Ignore MSR[PMM] for masking data trace messages (default)
ON	Mask (disable) data trace messages when MSR[PMM] = 0, unmask (enable) data trace messages when MSR[PMM] = 1

Format:

NEXUS.DTMWhileHalted [ON | OFF]

In the default setting (OFF), the debugger disables data trace messaging while the core is halted in debug mode. Therefore the core will not generate data trace messages for debugger issues read and write accesses while core halted. Set to ON to record the debugger's write accesses while the core is halted. Please note that memory accesses through the NEXUS block (**SYStem.MemAccess NEXUS**) do not generate data trace messages.

NEXUS.DQM

Enable data acquisition messaging

Format:

NEXUS.DQM [ON | OFF]
SYStem.Option.DQM [ON | OFF] (deprecated)

Default: OFF.

Set to ON to enable data acquisition messaging. Only available on processors which implement IEEE-ISTO 5001-2008 or later.

NEXUS.FRAYDTM

Enable FlexRay data trace messaging

Format:

NEXUS.FRAYDTM [Read | Write | ReadWrite | DTM | OFF] (deprecated)
SYStem.Option.DMADTM [ON | OFF] (deprecated)

Deprecated. Use **NEXUS.CLIENT<x>** commands.

Format:	NEXUS.HTM [ON OFF] SYStem.Option.HTM [ON OFF] (deprecated)
---------	---

Control program trace messaging mode.

OFF (default)	The core generates a program trace message for every taken direct or indirect branch (i.e. branch trace messaging).
ON	The core generates only program trace messages for taken indirect branches. For all direct branches, only the information taken/not take in generated (branch history trace messaging).

HTM can reduce the amount of program trace messages to about 10% of the classical branch trace messaging. Use this option to increase trace recording time or to prevent overflows of the on-chip nexus message fifo.

NOTE:	The debugger can reconstruct the full program flow with BTM as well as when using HTM. The only drawback of HTM is that runtime statistic results will be less accurate because of the lower amount of messages (fewer messages and therefore fewer timestamps per instruction). To some degree, the longer recording time will compensate the loss of accuracy. When supported by the processor, enable program trace correlation messages at branch-and-link occurrence (NEXUS.PTCM.BL_HTM ON) together with HTM. Doing so will achieve the same accuracy as BTM for run-time measurements on function level.
--------------	---

NEXUS.OFF

Switch the NEXUS trace port off

Format:	NEXUS.OFF
---------	------------------

If the debugger is used stand-alone, the trace port is disabled by the debugger.

If the debugger is used together with a calibration tool (**SYStem.Option.ETK / SYStem.Option.GSI**), any writes to trace registers (by the debugger) are suppressed. Use this setting if the calibration tool makes use of the data trace (e.g. XETK-V2 in data trace configuration). The debugger will continue to record the trace as a slave (i.e. trace configuration is exclusively done by calibration tool).

Format: **NEXUS.ON**

The NEXUS trace port is switched on. All trace registers are configured by debugger. Do not use if calibration tool makes use of data trace (e.g. XETK-V2 in data trace configuration).

Format:	NEXUS.OTM [PID0 NPIDR OFF] SYStem.Option.OTM [ON OFF] (deprecated)
---------	---

Controls ownership trace messaging.

OFF	Ownership trace messaging disabled (default)
PID0	Enable ownership trace messaging. An OTM is generated if the application writes to the PID0 register.
NPIDR	Enable ownership trace messaging. An OTM is generated if the application writes to the NPIDR register.
ON	Deprecated, use PID0 .

NOTE:	<ul style="list-style-type: none">Enable ownership trace messaging in order to get trace information about task switches. Some operating systems use a set of OTMs to transfer task switch information to the trace tool. In this case periodic ownership trace must be disabled using NEXUS.POTD ON.If program trace messaging is enabled (NEXUS.BTM ON) and NEXUS.PTCM.PID_MSR is ON, the core sends program trace correlation messages instead of ownership trace messages. The ownership data source (PID0/NIPDR) is nevertheless determined by the NEXUS.OTM setting.
--------------	--

Format:	NEXUS.PCRCONFIG [ON OFF]
---------	-----------------------------------

When enabled, the debugger configures the pads of MCKO, MDO and MSEO to NEXUS function. This command is only implemented for MPC560xS / SPC560S (Spectrum) and SPC56AP60.

Format:	NEXUS.PINCR NEXUS.PINCR <value>
---------	--

When this command is called with a value, the specified value is written to the DCI_PINCR register every time the debugger connects to the processor (SYStem.Up, Attach, Go, StandBy). The meaning of the individual bits, and the existence of this register in general, depends on the target processor.

Example for SPC570S:

```
SYStem.CPU SC570S50                ; select CPU

NEXUS.PortSize MDO4                 ; set trace port size to 4 MDOs
NEXUS.PINCR 0x01492492              ; map all trace pins to port "B"

SYStem.Up                           ; reset processor and halt core
```

Format:	NEXUS.PortMode <mode> SYStem.Option.MCKO <mode> (deprecated)
<mode>:	Parallel NEXUS: 1/1 1/2 1/3 1/4 1/8 Aurora NEXUS: 625MBPS 750MBPS 850MBPS 1000MBPS 1250MBPS 1500MBPS 1700MBPS 2000MBPS 2500MBPS 3000MBPS 3125MBPS

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

NOTES:	Parallel NEXUS: The settings 1/1 and 1/3 is not supported by all processors. Check the processor reference manual if this is a valid mode for your processor. Parallel NEXUS: Please check in the processor's data sheet if the NEXUS trace port of your processor is functional at the selected system frequency and MCKO divider. Current silicon versions allow trace port frequencies up to 60~80 MHz.
--------	---

Aurora NEXUS: Set the bit clock according to the processor's data sheet.

Aurora NEXUS: Automotive processors usually need an external reference clock for Aurora operation. The Aurora preprocessor can provide that clock signal. It is enabled using [NEXUS.RefClock ON](#).

NEXUS.PortSize

Set trace port width

Format:

NEXUS.PortSize *<port_size>*
SYStem.Option.NEXUS *<port_size>* (deprecated)

<port_size>:

Parallel NEXUS:
MDO16 | MDO12 | MDO8 | MDO 4 | MDO2

Aurora NEXUS:
2Lane | 4Lane

Sets the nexus port width to the number of used MDO pins or Aurora lanes. The setting can only be changed if no debug session is active ([SYStem.Down](#)).

NEXUS.POTD

Periodic ownership trace disable

Format:

NEXUS.POTD [ON | OFF]

Default: OFF. When enabled, the core is configured to suppress periodic ownership trace messages. A periodic ownership trace message is an OTM, which is generated without a write access to the PID register. Enable this option, when the OTM is used to generate trace information about task switches. OTMs are usually used for task switch tracing on processors with NEXUS 2+, because data trace is unavailable.

Format:	NEXUS.PTCM.<event> [ON OFF]
<event>:	PID_MSR BL_HTM TLBNEW TLBINV

Enables a program trace correlation message (PTCM) for the specified event. This program trace correlation messages are not needed to reconstruct the program flow, but give additional information which can increase precision of statistic measurements.

PID_MSR	Core generates PTCM when PID or MSR[IS] changes (EVCODE 0x5).
BL_HTM	Core generates PTCM on Branch and Link occurrence (EVCODE 0xA). Enable this PTCM to improve function profiling in branch history tracing mode .
TLBNEW	Core generates PTCM on new address translation - (EVCODE 0xB).
TLBINV	Core generates PTCM on address translation invalidated - (EVCODE 0xC).

Format:	NEXUS.PTMARK [ON OFF]
---------	-------------------------

Controls the influence of MSR[PMM] in program trace messaging. Only available on processors which implement IEEE-ISTO 5001-2008 or later.

OFF	Ignore MSR[PMM] for masking program trace messages (default)
ON	Mask (disable) program trace messages when MSR[PMM] = 0, unmask (enable) program trace messages when MSR[PMM] = 1

Format: NEXUS.RefClock [ON | OFF]

Aurora NEXUS only. When set to ON, the preprocessor provides the reference clock for the Aurora NEXUS block on the processor. Only enable when the processor requires this reference clock and when no module provides the Aurora clock source for the processor.

NEXUS.Register

Display NEXUS trace control registers

Format: NEXUS.Register

This command opens a window which shows the NEXUS configuration and status registers of NPC, core and other trace clients.

NEXUS.RESet

Reset NEXUS trace port settings

Format: NEXUS.RESet

Resets NEXUS trace port settings to default settings.

NEXUS.RFMHISTBUGFIX

Double RFM workaround

Format: NEXUS.RFMHISTBUGFIX [<time>]

Enable workaround for doubly issued register full messages. The program flow decoder will ignore the duplicate message when the processor sends the message twice within the specified time.

Affected processors are: MPC564xB/C, SPC564B/SPC56EC, MPC5674F, MPC567xK, SPC57HK, MPC564xL, SPC56EL, MPC564xA, SPC564A.

Format: **NEXUS.SmartTrace** [ON | OFF] (deprecated)

NEXUS.Spen<messagetype>

Enable message suppression

Format: **NEXUS.SpenDTM** [ON | OFF]
NEXUS.SpenDQM [ON | OFF]
NEXUS.SpenOTM [ON | OFF]
NEXUS.SpenPTM [ON | OFF]
NEXUS.SpenWTM [ON | OFF]

Configures the core to suppress one or more message types (WTM, PTM, DTM and OTM) when the on-chip NEXUS message FIFO reaches a certain fill level. Enabling one of these options will in most cases cause problems in trace analysis, because the trace message stream contains no information about if and when messages have been suppressed. The fill level at which message suppression occurs can be configured via the command [NEXUS.SupprTHReshold](#).

NOTE: Only available for processors which implement the **IEEE-ISTO 5001-2008** or later.

NEXUS.STALL

Stall the program execution when FIFO full

Format: Processors which implement IEEE-ISTO 5001-2003 standard:
NEXUS.STALL [ON | OFF]

Processors which implement IEEE-ISTO 5001-2008 or later:
NEXUS.STALL [1/4 | 1/2 | 3/4 | OFF]

SYStem.Option.STALL [ON | OFF] (deprecated)

Stall the program execution whenever the on-chip NEXUS-FIFO threatens to overflow. If this option is enabled, the NEXUS port controller will stop the core's execution pipeline until all messaged in the on-chip NEXUS FIFO are sent. Enabling this command will affect (delay) the instruction execution timing of the CPU. This system option, which is a representation of a feature of the processor, will remarkably reduce the amount FIFO OVERFLOW errors, but can not avoid them completely.

For processors which implement the **IEEE-ISTO 5001-2008** or later, STALL can be configured to occur at several fill levels, while processors which implement an older standard have a fixed level.

Format:

NEXUS.state

Displays the NEXUS trace configuration window.

NEXUS.SupprTHReshold

Set fill level for message suppression

Format:

NEXUS.SupprTHReshold [1/4 | 1/2 | 3/4]

Sets the NEXUS message FIFO fill level, at which messages will be suppressed by the core. The message types which will be suppressed are configured via the command **NEXUS.Suppr<message>**

NOTE:

Only available for processors which implement the **IEEE-ISTO 5001-2008** or later.

NEXUS.TimeStamps

Enable on-chip timestamp generation

Format:

NEXUS.TimeStamps [ON | OFF]

MPC57XX/SPC57X only. When enabled, the processor is configured to add timestamps to the NEXUS messages. If the chip-external trace is used (tracing to PowerTrace unit), on-chip timestamps are usually not needed, because the PowerTrace unit will add it's own timestamp. When using the on-chip trace (trace-to-memory), enable NEXUS.TimeStamps for run-time measurements.

NOTE:

- Check the processor reference manual if the used processor supports NEXUS timestamps.
- Not all trace clients of a processor may support NEXUS timestamps. (e.g. MPC5746M and SPC57EM80 do not support timestamps for NEXUS messages of the cores)
- Timestamps will consume ~20% of the trace bandwidth/trace memory

Format:

NEXUS.WTM [ON | OFF]

SYStem.Option.WTM [ON | OFF] (deprecated)

- ON

NEXUS outputs watchpoint messages.
- OFF

No watchpoint messages are output by NEXUS.

NOTE:

When a watchpoint is set with a [Break.Set](#) command, the NEXUS.WTM setting will be internally overridden to **ON**.

TrOnchip.Alpha

Set special breakpoint function

Format:

TrOnchip.Alpha <function>

<function>:

OFF

ProgramTraceON

ProgramTraceOFF

DataTraceON

DataTraceOFF

TraceEnableClient<x>

TraceDataClient<x>

TraceONClient<x>

TraceOFFClient<x>

TraceTriggerClient<x>

BusTriggerClient<x>

BusCountClient<x>

WATCHClient<x>

deprecated:

TraceEnableDMA | TraceDataDMA | TraceONDMA | TraceOFFDMA

TraceTriggerDMA | BusTriggerDMA | BusCountDMA | WATCHDMA

TraceEnableFRAY | TraceDataFRAY | TraceONFRAY | TraceOFFFRAY

TraceTriggerFRAY | BusTriggerFRAY | BusCountFRAY | WATCHFRAY

<x>:

1 | 2 | 3 | 4 | 5

Configures the functionality of the Alpha breakpoint. This breakpoint can be used to configure the on-chip NEXUS trace for special core features and for the trace clients configured via [NEXUS.CLIENT<x>SELECT](#).

For a description of the functionality and examples, see [Trace Filtering and Triggering with Debug Events](#) and [Tracing Peripheral Modules / Bus Masters](#).

TrOnchip.Beta

Set special breakpoint function

Format:

TrOnchip.Beta <function>

See [TrOnchip.Alpha](#).

Format: **TrOnchip.Charly** *<function>*

See [TrOnchip.Alpha](#).

Format: **TrOnchip.Delta** *<function>*

See [TrOnchip.Alpha](#).

Format: **TrOnchip.DISable**

Disables NEXUS register control by the debugger. By executing this command, the debugger will not write or modify any registers of the NEXUS block. This option can be used to manually set up the NEXUS trace registers. The NEXUS memory access is not affected by this command. To re-enable NEXUS register control, use command [TrOnchip.Enable](#). Per default, NEXUS register control is enabled.

Format: **TrOnchip.Echo** *<function>*

See [TrOnchip.Alpha](#).

Format:	TrOnchip.ENABLE
---------	-----------------

Enables NEXUS register control by the debugger. By default, NEXUS register control is enabled. This command is only needed after disabling NEXUS register control using [TrOnchip.Disable](#).

TrOnchip.EVTI

Allow the EVTI signal to stop the program execution

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

Default: OFF. If enabled, the debugger will use the EVTI signal to break program execution instead of sending a JTAG command. This will speed up reaction time. If the complex trigger unit is used to stop program execution, it is recommended to enable this option to achieve a shorter delay. If this option is disabled, the debugger will drive EVTI permanently high.

- NOTES:
- Only enable this option if the EVTI pin of the processor is connected to the NEXUS connector.
 - This option has no effect if [TrOnchip.EVTEN](#) is disabled.

TrOnchip.EVTO

Use EVTO signal for runtime measurement

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

Default: OFF. If enabled, the debugger will use the EVTO for [Run-time](#) measurement and external watchdog control. This will improve the precision of run-time measurement and reduce external watchdog control delays.

NOTE:	<ul style="list-style-type: none">• Only enable this option if the EVTO pin of the processor is connected to the NEXUS connector.• This option has no effect if TrOnchip.EVTEN is disabled.
-------	--

Format:	TrOnchip.EXternal <source>
<source>:	OFF IN0

The NEXUS adapter provides an additional (active-high) trigger input to stop the trace recording. The input is labeled “IN” or “IN0” on LA-7610 and “IX0” on LA-7630 adapters. The input channel recognizes signals with a minimum pulse length of 20 ns.

The recorded value of the input channel can be observed in the Trigger.0 row of the [Trace.List](#) window.

```
;Show program flow and input channel
Trace.List DEFault Trigger.0
```

The Complex Trigger Unit (CTU) supports the input channel level as condition *IN*.

TrOnchip.Out0

Select OUT0 pin signal source

Format:	TrOnchip.Out0 <source>
<source>:	Trigger Evto WDTC

Selects the signal source for the OUT0 pin of the NEXUS connector. On LA-7630 adapters, the signal is labeled “Ox0”.

Trigger	Trigger output of CTU (OUT.A)
Evto	High-active EVTO signal (inverted from processor’s low-active EVTO signal)
WDTC	Watchdog Timer Control

WDTC source can be used to activate the OUT0 output in parallel to the **TD/WDTE** pin. It can control an external watchdog. It is a second output, controlled by [TrOnchip.TOOLIO2](#).

There are the following features and restrictions:

- Only valid for Nexus AutoFocus preprocessor (LA-7630)
- Settings before the correct CPU is recognized or selected, will be removed.
- If **TriggerOnchip.Out0 WDTC** is selected, an external WDT can be controlled by the OUT0 pin of the Nexus probe.
- OUT0 pin can control a WDTC before a SYStem.Up command. (TD/WDTE pin cannot offer that).
- OUT0 can not be tristated and is always driving HIGH or LOW.
- OUT0 is not 5V tolerant, it can drive only 3.3V circuitry.

TrOnchip.Out1

Select OUT1 pin signal source

Format:

TrOnchip.Out1 <source>

<source>:

Trigger | Low | High | Run

Selects the signal source for the OUT1 pin of the NEXUS connector. Only available on LA-7610.

Trigger	Trigger output of CTU (OUT.B)
Low	Permanently low (GND)
High	Permanently high (VCC)
Run	Low while CPU running, high while stopped

Format:	TrOnchip.TOOLIO2 <source>
<source>:	OFF Trigger Low High Run Stop

Selects the signal source for the TOOLIO2 pin of the NEXUS connector. The signal source of TOOLIO2 is pin OUT1 on the NEXUS preprocessor, so changing this setting will also change OUT1. Refer also to [TrOnchip.Out0](#) command as a second way to control an output pin with slightly different features.

OFF	Tristated
Trigger	Trigger output of CTU (OUT.A)
Low	Permanently low (GND)
High	Permanently high (VCC)
Run	Low while CPU running, high while stopped; can be used to disable on-board watchdogs.
Stop	High while CPU running, low while stopped

Format:	TrOnchip.TRaceControl <i><event></i> <i><action></i>
<i><event></i> :	IRPT RET CIRPT CRET
<i><action></i> :	OFF TraceON TraceOFF ProgramTraceON ProgramTraceOFF DataTraceON DataTraceOFF TraceTrigger BusTrigger WATCH

Use this command to enable above trace actions for the specified debug event. Only available on processors which implement IEEE-ISTO 5001-2008 or later. See [Trace Filtering and Triggering with Debug Events](#) for an explanation of the available actions.

IRPT	Interrupt taken debug event
RET	Interrupt return debug event
CIRPT	Critical interrupt taken debug event
CRET	Critical interrupt return debug event

The example below shows how to disable the program trace for interrupts. On account of the on-chip implementation, the program trace will start after the first interrupt return (RFI instruction) is executed.

```
;Disable program trace for interrupt handler

TrOnchip.TRaceControl IRPT TraceOFF
TrOnchip.TRaceControl RET TraceON
```

14-pin JTAG/OnCE Connector (JTAG)

Signal	Pin	Pin	Signal
TDI	1	2	GND
TDO	3	4	GND
TCK	5	6	GND
(EVTI-)	7	8	N/C
RESET-	9	10	TMS
JTAG-VTREF	11	12	GND
(RDY-)	13	14	JCOMP

This connector is compatible to the JTAG connector used on the NXP/Freescale and STM evaluation boards.

AUTO26 Connector (JTAG)

Signal	Pin	Pin	Signal
VTREF	1	2	TMS
GND	3	4	TCK
GND	5	6	TDO
KEY(GND)	-	8	TDI
GND(PRESENCE)	9	10	RESET-
GND	11	12	RESETOUT-
GND	13	14	WDTDIS
GND	15	16	JCOMP
GND	17	18	EVTI-
GND	19	20	EVTO-
GND	21	22	BREQ-
GND	23	24	BGRNT-
GND	25	26	EXTIO

10-pin ECU14 Connector (with converter LA-3843)

Signal	Pin	Pin	Signal
GND	1	2	TCK/DAP0
TRST-/JCOMP	3	4	TDO/DAP2
TMS/DAP1	5	6	TDI/DAPEN
USERIO	7	8	VTREF
RESETOUT-	9	10	RESET-

38-pin Mictor Connector (NEXUS parallel)

Signal	Pin	Pin	Signal
MDO12	1	2	MDO13
MDO14	3	4	MDO15
MDO09	5	6	(CLKOUT)
N/C	7	8	MDO08
RSTIN-	9	10	EVTI-
TDO	11	12	VTREF
MDO10	13	14	RDY-
TCK	15	16	MDO07
TMS	17	18	MDO06
TDI	19	20	MDO05
JCOMP	21	22	MDO04
MDO11	23	24	MDO03
RESETOUT	25	26	MDO02
TDET/WDTDIS	27	28	MDO01
BGRNT	29	30	MDO00
N/C	31	32	EVTO-
N/C	33	34	MCKO
BREQ	35	36	MSEO1-
N/C	37	38	MSEO0-

50-pin SAMTEC ERF8 Connector (NEXUS parallel)

Signal	Pin	Pin	Signal
MSEO0-	1	2	VREF
MSEO1-	3	4	TCK
GND	5	6	TMS
MDO00	7	8	TDI
MDO01	9	10	TDO
GND	11	12	TRST- (JCOMP)
MDO02	13	14	DBGACK- (RDY)
MDO03	15	16	EVTI-
GND	17	18	EVTO-
MCKO	19	20	RSTIN-
MDO04	21	22	RSTOUT
GND	23	24	GND
MDO05	25	26	CLKOUT
MDO06	27	28	TD/WDTE
GND	29	30	GND
MDO07	31	32	DAI1
MDO08	33	34	DAI2
GND	35	36	GND
MDO09	37	38	ARBREQ
MDO10	39	40	ARBGRT
GND	41	42	GND
MDO11	43	44	MDO13
MDO12	45	46	MDO14
GND	47	48	GND
MDO15	49	50	N/C

51-pin GlenAir / ROBUST Connector (NEXUS parallel)

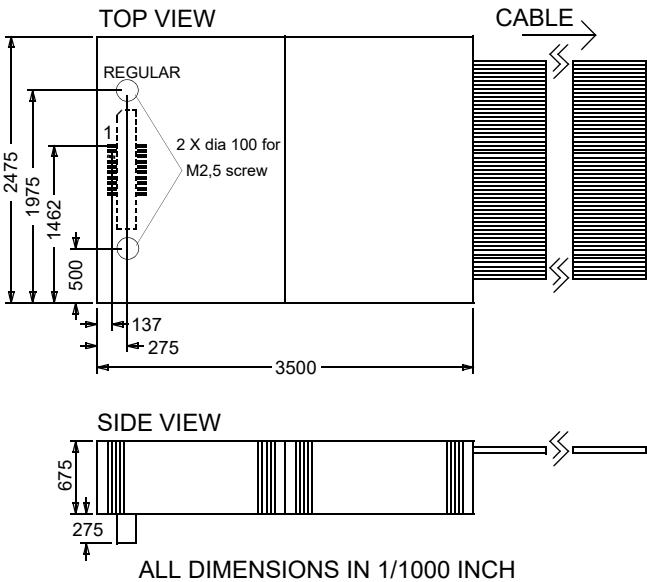
Pin	Signal
1	N/C
2	N/C
3	N/C
4	ARBREQ(TOOLIO0)
5	TDO
6	RDY-
7	RSTIN-
8	VREF
9	EVTI-
10	GND
11	TRST-
12	GND
13	TMS
14	GND
15	TDI
16	GND
17	TCK
18	GND
19	MDO0
20	GND
21	MCKO
22	GND
23	EVTO-
24	GND
25	MSEO0-
26	MDO9
27	MDO1
28	GND
29	MDO2
30	GND
31	MDO3
32	GND
33	ARBGRT(TOOLIO1)
34	GND
35	MSEO1-
36	GND
37	MDO4
38	GND
39	MDO5
40	GND
41	MDO6
42	GND
43	MDO7
44	GND
45	MDO8
46	GND
47	MDO10
48	GND
49	MDO11
50	GND (TDET)
51	RSTOUT(VENIO2)

34-pin SAMTEC ERF8 Connector (Aurora NEXUS)

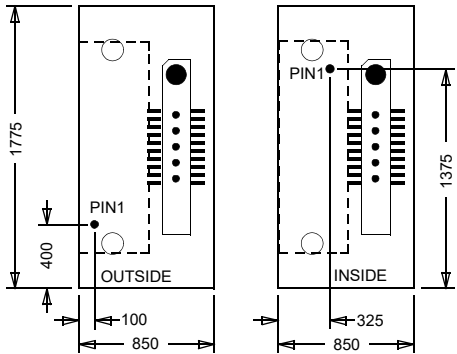
Signal	Pin	Pin	Signal
TXP0	1	2	JTAG-VTREF
TXN0	3	4	TCK
GND	5	6	TMS
TXP1	7	8	TDI
TXN1	9	10	TDO
GND	11	12	JCOMP-
TXP2	13	14	N/C
TXN2	15	16	EVTI-
GND	17	18	EVTO-
TXP3	19	20	RSTOUT-
TXN3	21	22	RSTIN-
GND	23	24	GND
N/C	25	26	CLKP
N/C	27	28	CLKN
GND	29	30	GND
N/C	31	32	RDY-
N/C	33	34	WDIS

Dimension

LA-7610 NEXUS-MPC5500



CONVERTER MICTOR TO GLENAIR 51
TOP VIEW



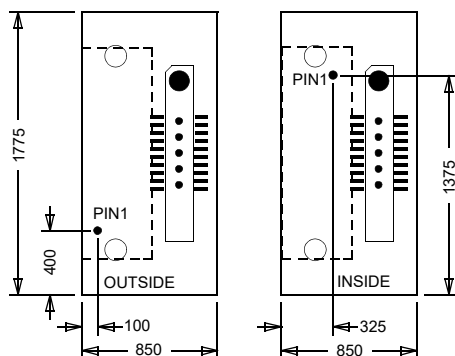
TWO ROTATE VERSIONS OF THE GLENAIR 51 PLUG AVAILABLE
STANDARD ORIENTATION IS OUTSIDE

Dimension

LA-7611 CONV-MIC38-GL51-5500

CONVERTER MICTOR TO GLENAIR 51

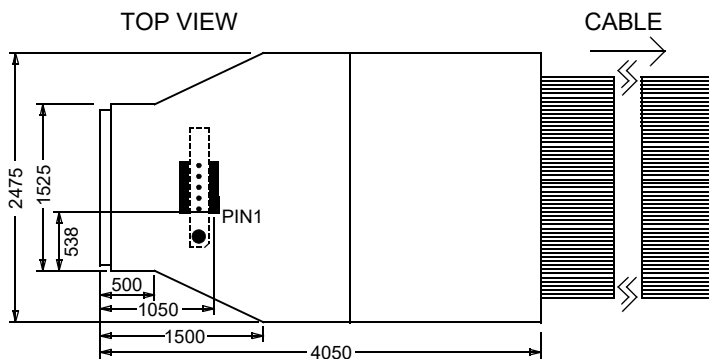
TOP VIEW



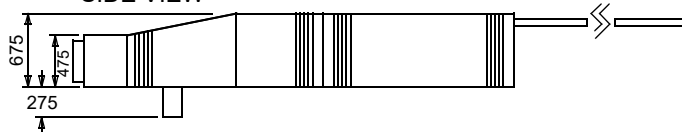
TWO ROTATE VERSIONS OF THE GLENAIR 51 PLUG AVAILABLE
STANDARD ORIENTATION IS OUTSIDE

LA-7612 NEXUS-MPC551X

TOP VIEW



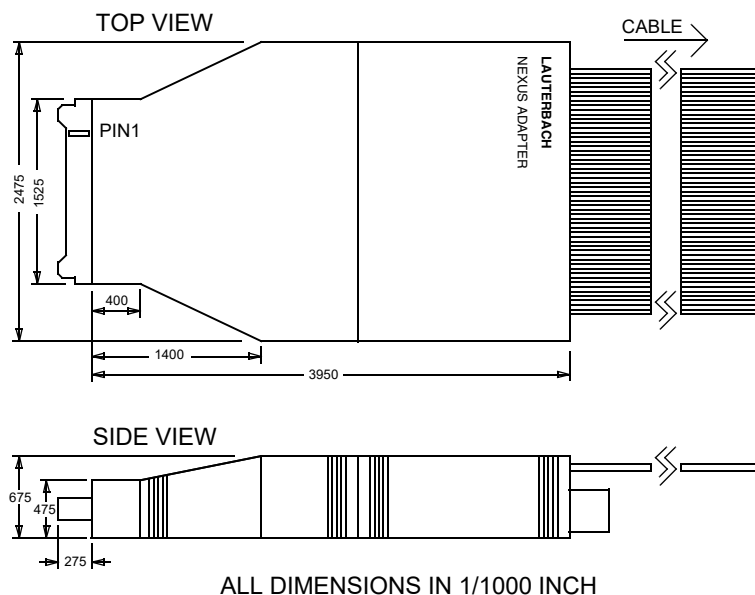
SIDE VIEW



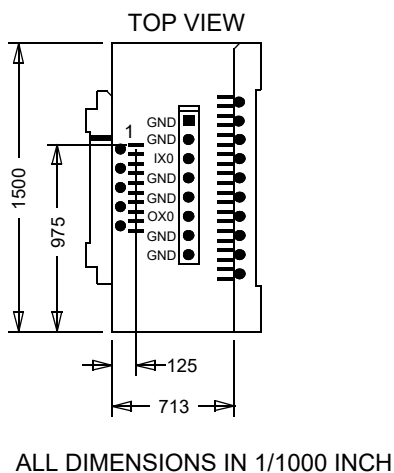
ALL DIMENSIONS IN 1/1000 INCH

Dimension

LA-7630 NEXUS-MPC5500-AF

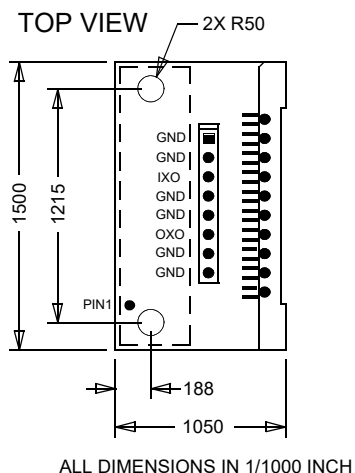


LA-7631 CONV-MIC38-GENERIC

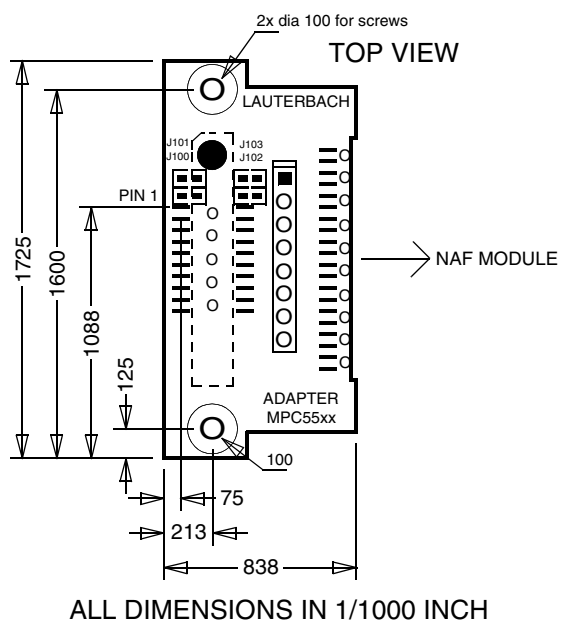


Dimension

LA-7632 CONV-GL51-MPC5500

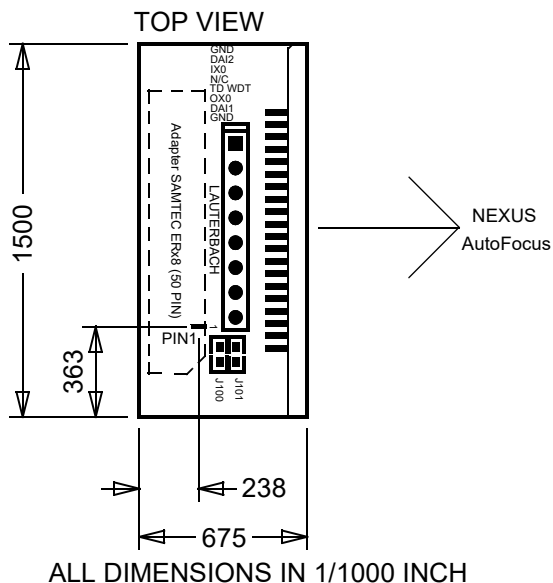


LA-7633 CONV-MIC38-MPC5500R

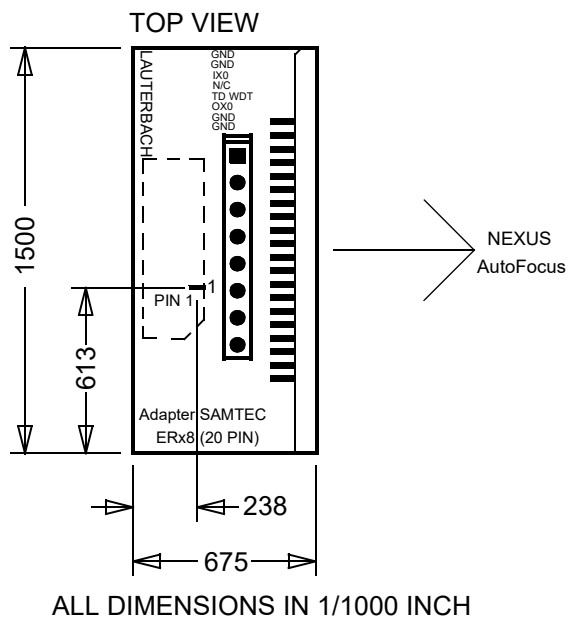


Dimension

LA-7636 CONV-MIC76-MPC5500-L

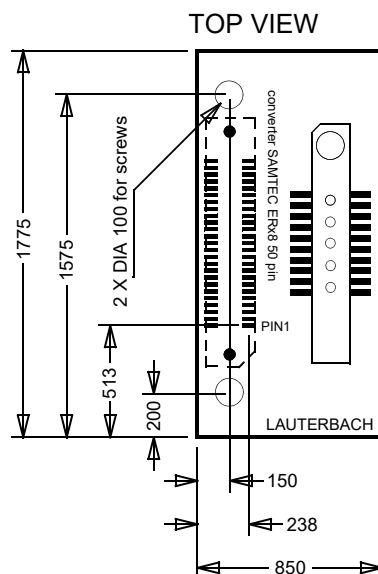


LA-7637 CONV-MIC76-MPC5500-S



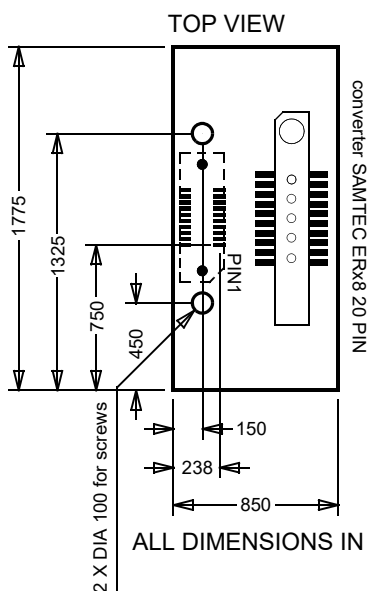
Dimension

LA-7638 CONV-MIC38-MPC5500-L



ALL DIMENSIONS IN 1/1000 INCH

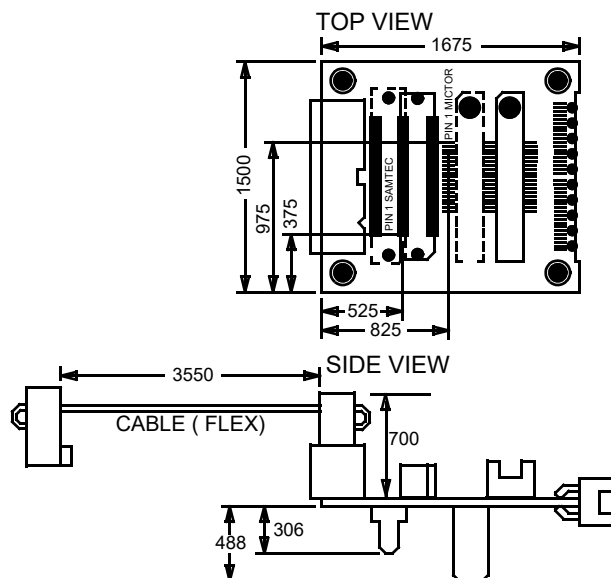
LA-7639 CONV-MIC38-MPC5500-S



ALL DIMENSIONS IN 1/1000 INCH

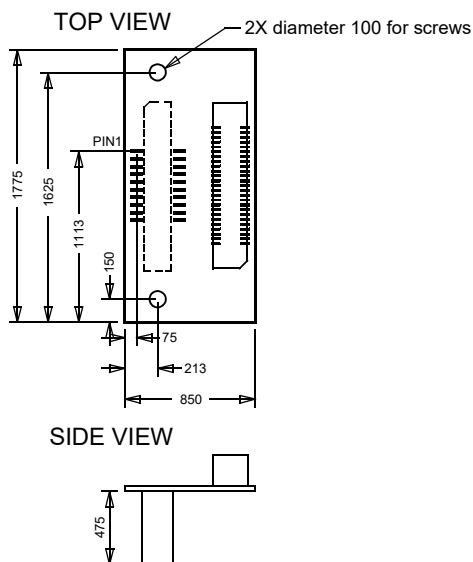
Dimension

LA-7640 CONV-MIC76-JTAG14



ALL DIMENSIONS IN 1/1000 INCH

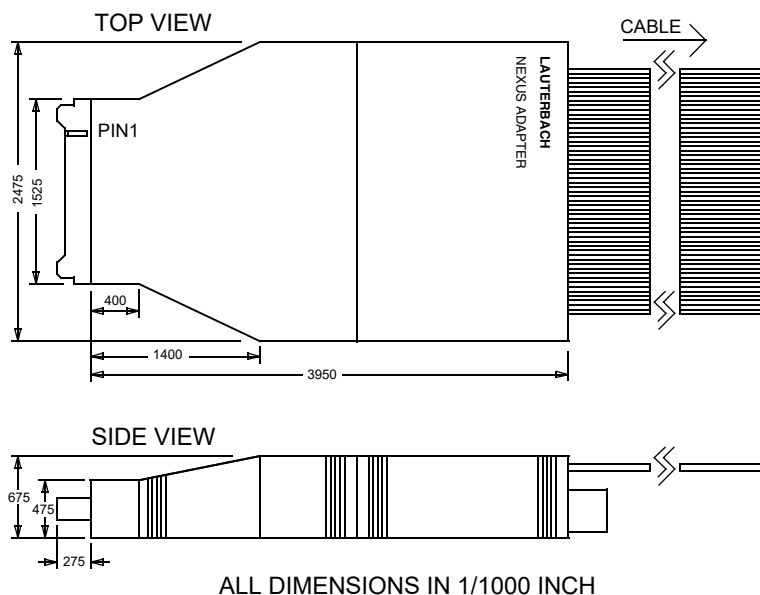
LA-7641 CONV-SAM50-MPC5500-L



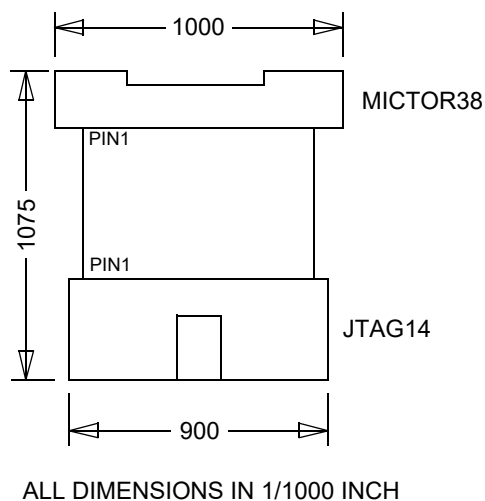
ALL DIMENSIONS IN 1/1000 INCH

Dimension

LA-7645 NEXUS-AVR32-AF

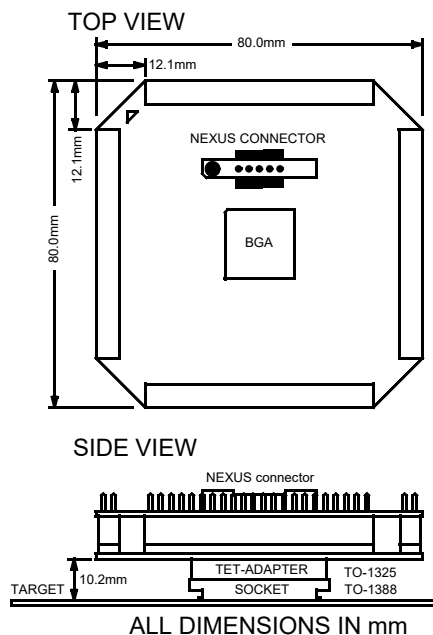


LA-3725 CONV-MIC38-J14-5500

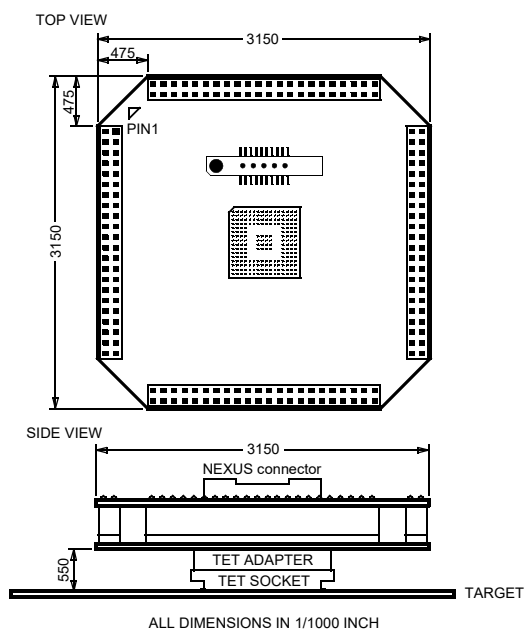


Dimension

LA-3854 ET176-SPC56ECXX



LA-3855 ET176-MPC5607BC



Operation Voltage

Adapter	OrderNo	Voltage Range
Debugger for MPC5xxx Automotive PRO	LA-2708	0.9 .. 5.5 V
Debug-Bundle MPC5xxx/RH850 Automotive PRO	LA-2712	1.6 .. 5.5 V
Debug-Bundle MPC5xxx/TriCore Automotive PRO	LA-2713	1.6 .. 5.5 V
JTAG Debugger for MPC5xxx Automotive	LA-3736	1.6 .. 5.5 V
Debugger-Bundle MPC5xxx/TriCore Automotive	LA-3738	1.6 .. 5.5 V
JTAG Debugger Qorivva MPC5xxx/SPC5xxx (ICD)	LA-7753	1.6 .. 5.5 V

Adapter	OrderNo	Voltage Range
Adap. SPC560B64-BGA208 to ET176 NEXUS	LA-3850	3.0 .. 3.5 V
Adap. SPC560C50-BGA208 to ET100 NEXUS	LA-3852	3.0 .. 3.5 V
Adap. MPC5646C-BGA256 to ET176 NEXUS	LA-3853	3.0 .. 3.5 V
Adap. SPC56ECxx-BGA256 to ET176 NEXUS	LA-3854	3.0 .. 3.5 V
Adap. MPC5607BC-BGA208 to ET176 NEXUS	LA-3855	3.0 .. 3.5 V
Adap. MPC5604BC-BGA208 to ET144 NEXUS	LA-3856	3.0 .. 3.5 V
Adap. MPC5604BC-BGA208 to ET100 NEXUS	LA-3857	3.0 .. 3.5 V
NEXUS Debug/Trace for Qorivva MPC5xxx/SPC5xxx	LA-7630	1.0 .. 5.2 V

Operation Frequency

Parallel NEXUS:

- 0 .. 100MHz

Aurora NEXUS:

- up to 3.125 gigabit/second with 4 lanes
- up to 6.250 gigabit/second with up to 3 lanes
- reference clock output up to 3.125 GHz
-