PQIII Debugger

PQIII Debugger

Version 06-Jun-2024

History

20-Jul-22	For the MMU.SCAN ALL command, CLEAR is now possible as an optional second parameter.

Introduction

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

- Freescale PowerQuicc III Series MPC85XX
- Freescale QorIQ P101x, P102x, P2010, P2020
- Freescale Qonverge PSC91XX, PSC92XX series

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

If some of the described functions, options, signals or connections in this Processor Architecture Manual are only valid for a single CPU or for specific families, the name(s) of the family(ies) is added in brackets.

Brief Overview of Documents for New Users

Architecture-independent information:

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

Architecture-specific information:

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

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

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

Demo and Start-up Scripts

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.

Warning

Signal Level

NOTE:

The debugger drives the output pins of the BDM/JTAG/COP connector with the same level as detected on the VCCS pin. If the debug/trace I/O pins of the processor are operating at e.g. 3.3 V, then VCCS should be connected to 3.3 V as well.
See also **System.up Errors**.

ESD Protection

WARNING:

To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF.

Recommendation for the software start:

1. Disconnect the Debug Cable from the target while the target power is off.
2. Connect the host system, the TRACE32 hardware and the Debug Cable.
3. Power ON the TRACE32 hardware.
4. Start the TRACE32 software to load the debugger firmware.
5. Connect the Debug Cable to the target.
6. Switch the target power ON.
7. Configure your debugger e.g. via a start-up script.

Power down:

1. Switch off the target power.
2. Disconnect the Debug Cable from the target.
3. Close the TRACE32 software.
4. Power OFF the TRACE32 hardware.

Target Design Requirement/Recommendations

General

- Locate the **BDM/JTAG/COP 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 $\overline{\text{HRESET}}$ is connected directly to the $\overline{\text{HRESET}}$ of the processor. This will provide the ability for the debugger to drive and sense the status of $\overline{\text{HRESET}}$. The target design should only drive $\overline{\text{HRESET}}$ 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 $\overline{\text{HRESET}}$.
- In order to start debugging right from reset, the debugger must be able to control CPU $\overline{\text{HRESET}}$ and CPU $\overline{\text{TRST}}$ independently. There are board design recommendations to tie CPU $\overline{\text{TRST}}$ to CPU $\overline{\text{HRESET}}$, 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.

Starting up the Debugger is done as follows:

5. Select the CPU to load the CPU specific settings. SYStem.DETECT CPU can set the appropriate CPU automatically.

```
SYStem.CPU MPC85XX
SYStem.DETECT CPU
```

6. Specify that on-chip breakpoints should be used by the debugger, e.g. for program in FLASH.

```
MAP.BOnchip 0xFF800000--0xFFFFFFFF
```

7. Reset processor and enter debug mode

```
SYStem.Up
```

   The core is now stopped at the reset address.

8. After SYStem.Up, only the boot page is visible for the CPU. Initialize MMU TLBs to configure which memory is visible to the CPU at which address. See **MMU.Set** for details.

```
; set up TLB entry starting from address 0 for SDRAM
MMU.Set TLB1 1. 0x80000600 0x00000000 0x0000003f
```

9. This step prepares the target memory for program loading. To configure the CPU for the access to all memories either run the initialization code on your target or configure the CPU by using the **Data.Set** command. For complete example scripts, see ~~/demo/powerpc/hardware.

```
; set CCSR base address to 0xE0000000
Data.Set ANC:iobase() %LONG 0x000E0000
; local access window
Data.Set ANC:iobase()+0x0C08 %long 0x00000000
Data.Set ANC:iobase()+0x0C10 %long 0x80F0001C
...
```

10. Load the program.

```
Data.LOAD.ELF demo.elf            (ELF specifies the format,
                                  demo.elf is the file name)
```

The option of the **Data.LOAD** command depends on the file format generated by the compiler. A detailed description of the **Data.LOAD** command is given in the **“General Commands Reference”**.

Troubleshooting

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command, there can be several reasons. The Following chapters list possible errors and explains how to fix them.

Target Power Fail

The target has no power, the debug cable is not connected or not connected properly. Check if the JTAG VCC pin is driven by the target. The voltage of the pin must be identical to the debug voltage of the JTAG signals. It is recommended to connect VCC directly to the pin, or via a resistor < 5 kOhm.

Debugger Configuration Error

The debugger was not able to identify the connected processor. There are two possible reasons for this error. In both cases, please check the AREA window for more information:

- The connected processor is not supported by the used software. Please check if the processor is supported by the debugger. Processors that appeared later than the debugger software version are usually not supported. Please download and install the latest software from our homepage, or contact technical support to get a newer software. Please also check if the processor or the software update is covered by your current licence.
- A JTAG communication error prevented correct determination of the connected processor. Please check if the debugger is properly connected to the target.

Target Reset Fail

On SYStem.Up, the debugger will assert HReset in order to stop the CPU at the reset address. A target reset fail means, that an unexpected reset behavior caused an error:

- The reset is asserted longer than 500ms and is not visible on the JTAG connector. Try **SYStem.Option.SLOWRESET**, and check signal level of the JTAG HRESET pin.
- The target reset is permanently asserted. Check target reset circuitry and reset pull-up
- A chip external watchdog caused a reset after the debugger asserted reset. Disable the watchdog and try again.

Emulation Debug Port Fail

An emulation debug port fail can have a variety of reasons. Please check the AREA window for a detailed error message. Here is a collection of frequently seen issues:

- JTAG communication error. Please check the signals on the debug connector
- Problems related with Reset can not always be detected as those. Please check **Target Reset Fail**
- AREA window error message “`Error reading BPTR`“: This error usually occurs if the CPU is permanently in reset or checkstop. Please check on your target:

  - reset and checkstop signals

  - power supply

  - system clocks and PLL

  - bootstrap configuration pins

In many cases it is possible to verify the bootstrap configuration even if SYStem.Up fails:

```
SYStem.CPU MPC85XX
SYStem.DETECT CPU
SYStem.MemAccess Enable
SYStem.Mode.Attach
PER , "Global Utilities,Power-On" /DualPort
```

If the above sequence fails to display the power-on reset configuration registers (displaying question marks, bus error. This can e.g. be the case when the PLL configuration is wrong), there is an alternative method to access the bootstrap configuration information. For instructions please contact bdmppcpq3-support@lauterbach.com.

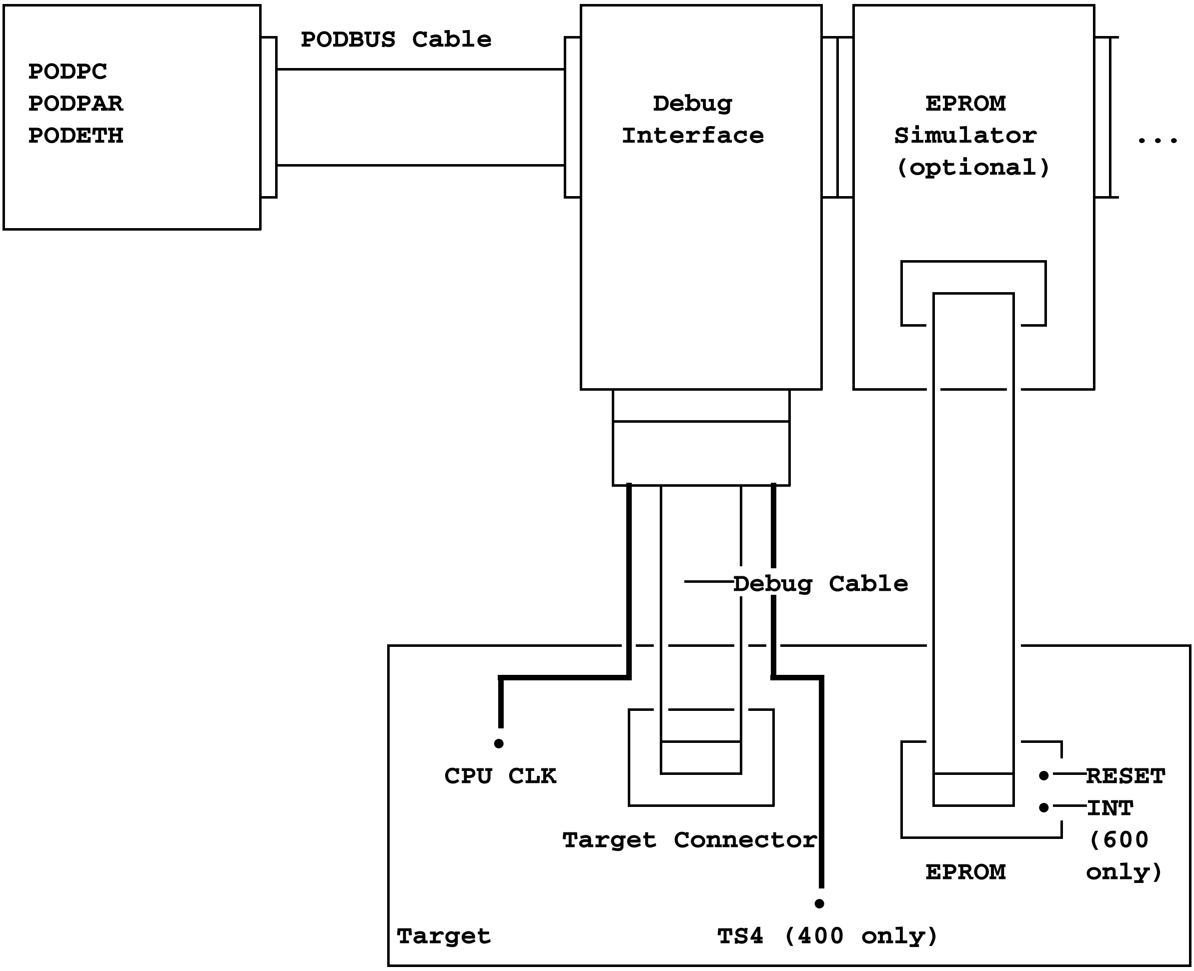
SYStem.Up will also fail if the processor is configured to boot from NAND, but the NAND flash contains invalid data. The processor enables NAND error checking upon reset. If the ECC in the spare area does not match data in main AREA, loading the NAND flash sector fails and the debugger can not connect. The workaround is to change the bootstrap configuration to ROM_LOG=GPCM.

If the bootstrap configuration was found to be wrong or needs to be changed temporarily (e.g. for NAND programming), it is possible to override the bootstrap configuration setting through JTAG. For instructions again please contact support using above email address.

FAQ

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

System Overview



Basic configuration for the BDM Interface

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.

Processor	On-chip Breakpoints	Instruction Address Breakpoints	Data Address Breakpoints	Data Value Breakpoints
MPC85XX P10xx P20xx P40xx	4 Instruction 2 Read/Write	2 single breakpoints -- or -- 1 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	none

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 0xFFFC0000--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.

```
;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

The e500 core does not support onchip breakpoints on data values, but TRACE32 supports them by software emulation. When a data value breakpoint is set, the debugger will use one of the data address breakpoint s. 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.

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.
```

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:**
Data.List **P:**0x1000	Opens a List window displaying **program** memory
Data.dump **D:**0xFF800000 /LONG	Opens a DUMP window at **data** address 0xFF800000
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)

In addition to the access classes, there are access class attributes: Examples:

Command:	**Effect:**
Data.List SP:0x1000	Opens a List window displaying **supervisor** program memory
ED:0x3330 0x4F	Write 0x4F to address 0x3330 using **real-time memory access**

The following access class attributes are available:

Access Class Attributes	Description
E	Use real-time memory access
A	Given address is physical (bypass MMU)
U	TS (translation space) == 1 (user memory)
S	TS (translation space) == 0 (supervisor memory)

If an Access class attributes is specified without an access class, TRACE32 PowerView will automatically add the default access class of the used command. For example, **Data.List** U:0x100 will be changed to **Data.List** 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

Cache

Memory Coherency

The following table describes which memory will be updated depending on the selected memory class:

Memory Class	D-Cache	I-Cache	L2 Cache	Memory (uncached)
DC:	updated	not updated	not updated	not updated
IC:	not updated	updated	not updated	not updated
L2:	not updated	not updated	updated	not updated
NC:	not updated	not updated	not updated	updated
(*) Depending on the debugger configuration, the coherency of the instruction cache will not be achieved by updating the instruction cache, but by invalidating the instruction cache. See **SYStem.Option.ICFLUSH** for details.				

Memory Class	D-Cache	I-Cache	L2 Cache	Memory (uncached)
D:	updated	not updated	updated	updated
P:	not updated	updated (*)	updated	updated

(*) Depending on the debugger configuration, the coherency of the instruction cache will not be achieved by updating the instruction cache, but by invalidating the instruction cache. See **SYStem.Option.ICFLUSH** for details.

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)

Viewing Cache Contents

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

Cache	Command
L1 instruction cache	CACHE.DUMP **IC**
L1 data dache	CACHE.DUMP **DC**
L2 (unified cache)	CACHE.DUMP **L2**

The meaning of the data fields in the **CACHE.DUMP** window os explained in the following table:

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, s	Status bits of the cache line v(alid), d(irty), s(hared)
#	MESI state
l	l(ocked).
00 04 08 ...	Address offsets within cache line corresponding to the cached data
address (right field)	Debug symbol assigned to address

Debugging Information

In order to properly use all debug features (breakpoints, single step etc) of the MPC85XX, the **Debug Interrupt Vector** (IVPR+IVOR15) must be set to an address which is

- properly mapped in the MMU (memory management unit) and
- points to an address which contains a valid instruction (NOP is recommended).

Please note that both IVOR/IVPR and memory contents can be changed by the application any time, especially during the boot process. When debugging is done after the boot process finished, the interrupt vector and memory is usually properly set up by the application. There are however operating systems that don’t use the debug interrupt and let it point to an illegal instruction.

For early CPU revisions (PVR=0x8020XXXX) it is recommended to place the instructions NOP followed by RFCI to the debug interrupt vector. These two instructions are needed for **SYStem.Option.FREEZE**.

```
; CORE 1 setup script:                  ; CORE 2 setup script:

SYStem.CPU 5516                         SYStem.CPU 5516

SYStem.CONFIG.CORE 1. 1.                SYStem.CONFIG.CORE 2. 1.

SYStem.UP                               SYStem.Mode.Attach

; do board initialization here          ; z0 is still in reset

Data.LOAD.Elf demo.elf                  Data.LOAD.Elf demo.elf /NoCODE

                                        Break          ; with this command
                                                       ; z0 will stop when
                                                       ; reset is released

Go     ; start z1                       WAIT !RUN()    ; wait until cpu stops
       ; application will start z0
       ; core

                                        Break.Set somez0function

                                        Go
```

Multicore Debugging e500 cores

SMP Debugging

For the dual-core processors MPC8572 and the dual-core variants of P10xx and P20xx, SMP debugging is selected by default. No further configuration is needed. As soon as the debugger is connected (SYStem.Up, SYStem.Mode.Attach etc.), it is possible to switch to any core using the **CORE <core_index>** command. The currently selected core is displayed in the status line. If the cores are running and one of the cores hits a breakpoint, the debugger’s view will automatically switch to this core.

AMP Debugging

For AMP debugging, a separate instance of TRACE32 has to be started for each core. It is recommended to use **TRACE32 Start** to start the TRACE32 instances. Optionally the second instance can also be started by PRACTICE script. Each TRACE32 instance has to be configured to address one of the cores. This is done using the commands **SYStem.CONFIG.CORE** and **CORE.NUMBER**. SYStem Options PERSTOP and DCFREEZE have to be turned OFF to maintain cache coherency for the times when one of the cores is running and the other stopped.

The following commands show the basic setup commands for both TRACE32 instances:

```
; CORE 0 setup script:               ; CORE 1 setup script:

SYStem.CPU P2020                     SYStem.CPU P2020

SYStem.CONFIG.CORE 1. 1.             SYStem.CONFIG.CORE 2. 1.

CORE.NUMBER 1                        CORE.NUMBER 1

SYStem.Option.PERSTOP OFF            SYStem.Option.PERSTOP OFF

SYStem.Option.DCFREEZE OFF           SYStem.Option.DCFREEZE OFF

SYStem.Up

                                     SYStem.Mode.ATTACH
```

In order to synchronously run and halt both cores, use the **SYNCH** commands.

There is a complete demo for debugging P10xx/20xx dual-core processors on AMP mode in demo\powerpc\hardware\qoriq_p1_p2\amp_debugging in the TRACE32 installation directory.

Synchronous stop of both e500 cores

MPC8572/P10xx/P20xx processors do not implement a break switch on silicon. If SYNCH is configured to synchronous break in AMP mode, or always if SMP mode is selected, the core that did not hit a breakpoint will be stopped by the debugger. The missing hardware implementation on the processor causes a delay between both cores typically in the 1..10 millisecond range.

Programming Flash on MPC85XX / QorIQ P10XX/P20XX, PSC93XX

There are many example scripts for NOR FLASH, NAND FLASH and EEPROMs available.

The example scripts are in the folders:

- `~~/demo/powerpc/hardware/mpc85xx/`
- `~~/demo/powerpc/hardware/qoriq_p1_p2/`
- `~~/demo/powerpc/hardware/bsc913x/`

For NOR FLASH on LBC/IFC CS0, there are ready-to-use flash scripts which can be used without change. These scripts can be found in the all_boards subfolder.

Scripts for NAND, EMMC, SPI and for the I2C boot sequencer EEPROM have to be modified in respect of the target board's characteristics and used FLASH devices. Therefore many reference scripts usable on evaluation boards are included in the corresponding subfolder.

There are also example script which can encode and program the data as requested by the processor when booting from SPI or using the boot sequencer, for example:

- `~~/demo/powerpc/hardware/mpc85xx/mpc8536ds/program_spibootflash.cmm`
- `~~/demo/powerpc/hardware/mpc85xx/mpc8569mds/program_bootsequencer.cmm`

On-chip Trace on MPC85XX/QorIQ

Processors of the MPC85XX series have a built-in trace buffer with 256 entries. It can be used to trace transactions that occur on the internal memory bus according to the selected major interface (local bus, DDR SDRAM and PCI). The trace buffer holds information about transaction address, transaction type, source, target ID and the byte count.

The interface can be selected with the command Onchip.Mode.IFSel. All other configurations can be done directly via the menu for CPU peripherals in the section “Debug Features and Watchpoint Facility”.

Here is an example of how to set up the on-chip trace buffer to trace the data accesses of the PowerPC core. Please note that only uncached accesses will be recorded in the trace buffer::

```
; select interface ECM
Onchip.Mode.IFSEL ECM

; configure onchip trace
; TBCR0 address match disable              0x40000000
;       transaction match disable          0x20000000
;       source ID enable                   0x04000000
;       method trace events                0x00020000
Data.Set iobase.address()+0x000E2040 %LONG 0x64020000

; TBCR1 src ID = d-fetch          0x00110000
Data.Set iobase.address()+0x000E2044 %LONG 0x00110000

; enable automatically when CPU is started
Onchip.AutoArm ON
```

```
; initialize trace buffer
Onchip.Init

; start program until some_func is reached
Go some_func

; display trace buffer
Onchip.List
```

Regarding instruction fetch traces, please note that the trace buffer is connected outside the caches, so instruction fetches on cached addresses will not appear in the trace. As the core will always fetch a full instruction cache way (32 bytes) at once, the program trace can not be reconstructed using this on-chip trace.

Also data trace is limited to uncached accesses. The data value of the load/store access is not contained in the trace data.

For more information about general trace commands see ’Trace’ in ’General Commands Reference Guide T’ and ’Onchip Trace Commands’ in ’General Commands Reference Guide O’.

SYStem.BdmClock

Set BDM clock frequency

Format:	**SYStem.BdmClock** <*rate*>
<*rate*>:	**5kHz … 50MHz**

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

NOTE:	**MPC85XX / QorIQ** The recommended maximum JTAG frequency is 1/10th of the core frequency. Multi-core processors are limited to max 30 MHz.

SYStem.CONFIG.state

Display target configuration

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

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.

SYStem.CONFIG — Configure debugger according to target topology

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

<parameter> (JTAG):

DRPRE
DRPOST
IRPRE
IRPOST

CHIPDRLENGTH *<bits>*
CHIPDRPATTERN [**Standard** | **Alternate** *<pattern>*]
CHIPDRPOST *<bits>*
CHIPDRPRE *<bits>*
CHIPIRLENGTH *<bits>*
CHIPIRPATTERN [**Standard** | **Alternate** *<pattern>*]
CHIPIRPOST*<bits>*
CHIPIRPRE *<bits>*

TAPState
TCKLevel
TriState
Slave

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.

… **DRPOST** *<bits>*

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

… **DRPRE** *<bits>*

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

… **IRPOST** *<bits>*

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

… **IRPRE** *<bits>*

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

CHIPDRLENGTH *<bits>*

Number of Data Register (DR) bits which needs to get a certain BYPASS pattern.

CHIPDRPATTERN [**Standard** | **Alternate** *<pattern>*]

Data Register (DR) pattern which shall be used for BYPASS instead of the standard (1...1) pattern.

CHIPIRLENGTH *<bits>*

Number of Instruction Register (IR) bits which needs to get a certain BYPASS pattern.

CHIPIRPATTERN [**Standard** | **Alternate** *<pattern>*]

Instruction Register (IR) pattern which shall be used for BYPASS instead of the standard pattern.

TAPState

(default: 7 = Select-DR-Scan) This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.

TCKLevel

(default: 0) Level of TCK signal when all debuggers are tristated.

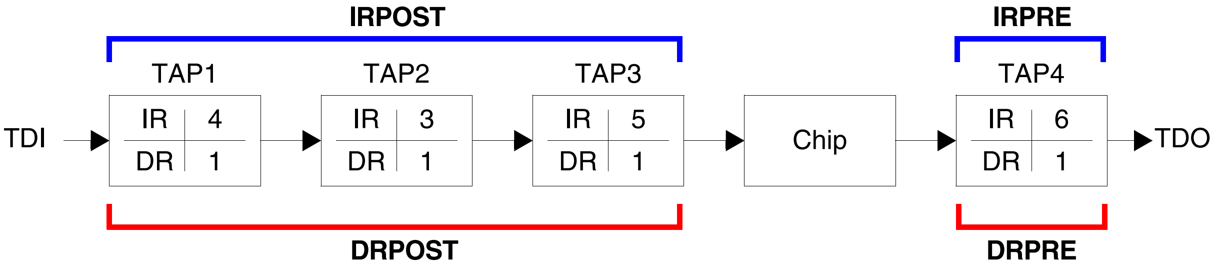
TriState

(default: OFF) If 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.

Slave

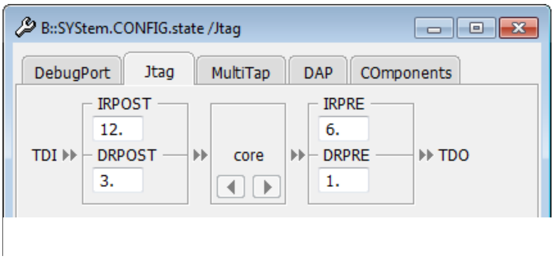
(default: OFF) If more than one debugger share the same JTAG port, all except one must have this option active. Only one debugger - the “master” - is allowed to control the signals nTRST and nSRST (nRESET).

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.


B::SYStem.CONFIG.state /Jtag

IRPOST 12.

DRPOST 3.

IRPRE 6.

DRPRE 1.

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.
```

TapStates

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`

SYStem.CONFIG.CHKSTPIN Control pin 8 of debug connector

Format: **SYStem.CONFIG.CHKSTPIN LOW | HIIGH**

Default: HIGH.

Controls the level of pin 8 (/CHKSTP_IN or /PRESENT) of the debug connector.

SYStem.CONFIG.DriverStrength — Configure driver strength of TCK pin

Format:	**SYStem.CONFIG DriverStrength** *<signal>* <**LOW** | **MID** | **HIGH**>
<signal>:	**TCK**

Default: HIGH.

Configures the driver strength of the TCK pin.

Available for debug cables with serial number C15040204231 and higher.

SYStem.CONFIG.QACK — Control QACK pin

Format:	**SYStem.CONFIG QACK TRISTATE** | **QREQ** | **LOW** | **HIGH**

Controls the level and function of pin 2 (/QACK) of the debug connector. Default: TRISTATE.

TRISTATE	Pin is disabled (tristate).
QREQ	Pin is driven to level of QREQ (pin 5).
LOW	Pin is driven to GND permanently.
HIGH	Pin is driven to JTAG_VREF permanently.

SYStem.CPU — Select the target processor

Format:	**SYStem.CPU** *<cpu_name>*
<cpu_name>:	**MPC85XX** | **MPC8540** | **MPC8560**…

Select the target processor or target core. If the target processor is not available in the CPU selection of the SYStem window, or if the command results in an error,

- check if the licence of the debug cable includes the desired processor. You will find the information in the **VERSION** window.
- check if the debugger software is sufficiently recent to support the target processor. The debugger software version can be looked up in the **VERSION** window. If the processor release occurred after the debugger software release, the processor is most likely not supported. Please check the Lauterbach download center (**www.lauterbach.com**) for updates. If the debugger software version from the download center also does not support the processor, please contact technical support and request a software update.

If you are unsure about the processor, try **SYStem.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. The command has no effect for the simulator.

SYStem.MemAccess Select run-time memory access method

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

<mode>: **Denied** | **Enable**

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 CPU (deprecated)	The debugger performs memory accesses via a dedicated CPU interface. This memory access will snoop data cache and L2 cache if a access class for data ("D:") is used.
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.

SYStem.Mode

Select operation mode

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** | **Go** | **Attach** | **StandBy** | **Up**

Select target reset mode.

Down	Disables the debugger. The state of the CPU remains unchanged.
NoDebug	Resets the target with debug mode disabled. In this mode no debugging is possible. The CPU state keeps in the state of NoDebug.
Go	Resets the target with debug mode enabled and prepares the CPU 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 and then stop the CPU at the first instruction at the reset address.
Up	Resets the target/processor and sets the CPU to debug mode. After execution of this command the CPU is stopped and prepared for debugging.

CPU specific SYStem.Option Commands

SYStem.Option.CINTDebug Enable debugging of critical interrupts

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

If the CPU enters a critical interrupt, MSR_DE will be cleared, which means that breakpoints are disabled. Enable this option in order to set a breakpoint in a critical interrupt handler. When enabled, the debugger will stop the CPU upon entering a critical interrupt, set MSR_DE and run the CPU again.

Please note that this option will have **influence on the run-time behavior** (i.e. performance loss) of the system, as the debugger needs to stop the CPU to set MSR_DE. As alternative to this option, patch the application to re-enable MSR_DE in the critical interrupt handlers. After MSR_DE has been restored, it is safe to use breakpoints.

SYStem.Option.CoreStandBy On-the-fly breakpoint 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 right from the first instruction executed. This option has impact on the real-time behavior. Releasing a secondary core from reset / disable state will be delayed for a few milliseconds.

SYStem.Option.DCFREEZE Prevent data cache line load/flush in debug mode

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

Default: OFF.

If OFF, the debugger will maintain D/L2 cache coherency by performing cache snoops for memory accesses. During the cache snoop, the processor will flush (clean and invalidate) dirty lines from data caches before the debugger's memory access takes place. This setting allows better data throughput and is recommended for normal application level debugging. In order to see changes to the cache state caused by debugging in the **CACHE.DUMP** window, use the command **CACHE.RELOAD**.

If ON, the debugger will maintain cache coherency by reading or writing directly to the cache arrays. This method guarantees that the D/L2 cache tags and status bits (valid, dirty) remain unaffected by the memory accesses of the debugger. This setting is recommended for low-level and cache debugging.

SYStem.Option.DCREAD — Read from data cache

Format: **SYStem.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 or L2 cache, if valid. If data is not available in cache, physical memory will be read.

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**.

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.

NOTE: **MPC85XX with PVR 0x8020XXXX**	For the MPC85XX CPU family, the debugger needs to execute a RFCI instruction out of memory to unfreeze the system timers on a resume (go, step). In order to use SYStem.Option.FREEZE, you have to patch two instructions to memory, a NOP followed by a RFCI, and let the IVPR/IVOR15 point to the NOP instruction. If **SYStem.Option.Freeze** is ON, the debugger will automatically check if the IVPR/IVOR15 vector is pointing to the NOP / RFCI instructions. If this condition does not match, the system timers will stay frozen and there will be an error output in the AREA window.

SYStem.Option.HOOK Compare PC to hook address

Format:	**SYStem.Option.HOOK** *<address>* | *<address_range>*

The command defines the hook address. After program break the hook address is compared against the program counter value.

If the values are equal, it is supposed that a hook function was executed. This information is used to determine the right break address by the debugger.

SYStem.Option.ICFLUSH Invalidate instruction cache before go and step

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

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.

SYStem.Option.ICREAD Read from instruction cache

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

Default: OFF:

If enabled, **Data.List** window and **Data.dump** window for access class P: (program memory) display the memory values from the instruction cache L2 cache if valid. If the data is not available in cache, the physical memory will be displayed.

SYStem.Option.IMASKASM Disable interrupts while single stepping

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

Default: OFF.

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

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.MMUSPACES Separate address spaces by space IDs

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

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **"TRACE32 Concepts"** (trace32_concepts.pdf).

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

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

1. Activate **SYStem.Option.MMUSPACES**.

2. Load the symbols with **Data.LOAD**.

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

Examples:

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

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

SYStem.Option.NoDebugStop Disable JTAG stop on debug events

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.NOTRAP   Use alternative software breakpoint instruction

Format:	**SYStem.Option.NOTRAP** *<type>*
<type>:	**OFF** | **FPU** | **ILL** **ON** (deprecated, same as **FPU**)

Defines which instruction is used as software breakpoint instruction.

OFF

Use TRAP instructions as software breakpoint (default setting). Software breakpoint will overwrite SRR0/1 registers.

FPU

Use an FPU instruction as software breakpoint.
Gives the ability to use the program interrupt in the application without halting for JTAG.
This setting only works if the application does not use floating point instructions (neither hardware nor software emulated). MSR[FP] must be set to 0 at all times.
Software breakpoint will overwrite SRR0/1 registers.

ILL

Use an illegal instruction as software breakpoint. This setting is recommended for MPC82XX, MPC5200, RHPPC (G2/G2_LE cores) and MPC830X, MPC831X, MPC832X and MPC512X (e300c2/3/4). Gives the ability to use the program interrupt in the application without halting for JTAG.
Illegal instructions as software breakpoints will preserve SRR0/1 registers.

SYStem.Option.OVERLAY   Enable overlay support

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

Default: OFF.

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

OFF Disables support for code overlays.

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

Example:

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

SYStem.Option.PERSTOP   Stop on-chip peripherals in debug mode

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

Default: ON. If enabled the debugger will halt the on-chip peripherals of the processor while in debug mode. memory accesses and cache snoops of e.g. TSEC, USB etc will not take place and memory spaces of some peripherals are inaccessible. If disabled, the on-chip peripherals will stay active also during debug mode. The data buffers of TSEC etc. can overflow, because the target application does not process the data when stopped.

NOTE: If **SYStem.Option.PERSTOP** is disabled, it is recommended to also disable **SYStem.Option.DCFREEZE**, in order to see the memory accesses performed by the peripherals.

SYStem.Option.RESetBehavior Set behavior when target reset detected

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

Defines the debugger's action when a reset is detected. Default setting is **Disabled**. The reset can only be detected and actions taken if it is visible to the debugger's reset pin.

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.

SYStem.Option.SLOWRESET Relaxed reset timing

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

This system option defines, how the debugger will test JTAG_RESET. For some system mode changes, the debugger will assert JTAG_RESET. By default (OFF), the debugger will release RESET and then read the RESET signal until the RESET pin is released. Reset circuits of some target boards prevent that the current level of RESET can be determined via JTAG_RESET. If this system option is enabled, the debugger will not read JTAG_RESET, but instead waits up to 4 s and then assumes that the boards RESET is released.

SYStem.Option.STEPSOFT Use alternative method for ASM single step

Format:	**SYStem.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.

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.

MMU.DUMP Page wise display of MMU translation table

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

Displays the contents of the CPU specific MMU translation table.

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

<root>	The *<root>* argument can be used to specify a page table base address deviating from the default page table base address. This allows to display a page table located anywhere in memory.
<range> *<address>*	Limit the address range displayed to either an address range or to addresses larger or equal to *<address>*. For most table types, the arguments *<range>* or *<address>* can also be used to select the translation table of a specific process if a space ID is given.
PageTable	Displays the entries of an MMU translation table. • if *<range>* or *<address>* have a space ID: displays the translation table of the specified process • else, this command displays the table the CPU currently uses for MMU translation.

KernelPageTable	Displays the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and displays its table entries.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Displays the MMU translation table entries of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and displays its table entries. • For information about the first three parameters, see **“What to know about the Task Parameters”** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

TLB0	Displays the contents of TLB0.
TLB1	Displays the contents of TLB1.

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. • if *<range>* or *<address>* have a space ID: list the translation table of the specified process • else, this command lists the table the CPU currently uses for MMU translation.

KernelPageTable	Lists the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the MMU translation table of the kernel and lists its address translation.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>*:**0x0**	Lists the MMU translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and lists its address translation. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manuals**.

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

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

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

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

PageTable	Loads the entries of an MMU translation table and copies the address translation into the debugger-internal static translation table. • if *<range>* or *<address>* have a space ID: loads the translation table of the specified process • else, this command loads the table the CPU currently uses for MMU translation.

KernelPageTable	Loads the MMU translation table of the kernel. If specified with the **MMU.FORMAT** command, this command reads the table of the kernel and copies its address translation into the debugger-internal static translation table.
TaskPageTable *<task_magic>* | *<task_id>* | *<task_name>* | *<space_id>***:0x0**	Loads the MMU address translation of the given process. Specify one of the **TaskPageTable** arguments to choose the process you want. In MMU-based operating systems, each process uses its own MMU translation table. This command reads the table of the specified process, and copies its address translation into the debugger-internal static translation table. • For information about the first three parameters, see **"What to know about the Task Parameters"** (general_ref_t.pdf). • See also the appropriate **OS Awareness Manual**.
ALL [**Clear**]	Loads all known MMU address translations. This command reads the OS kernel MMU table and the MMU tables of all processes and copies the complete address translation into the debugger-internal static translation table. See also the appropriate **OS Awareness Manual**. **Clear:** This option allows to clear the static translations list before reading it from all page translation tables.

CPU specific tables in MMU.SCAN <table>

TLB0	Loads the TLB0 from the CPU to the debugger-internal translation table.
TLB1	Loads the TLB1 from the CPU to the debugger-internal translation table.

Formats:	**MMU.Set** TLB0 <*index*> <*mas1*> <*mas2*> <*mas3*> <*mas7*> **MMU.Set** TLB1 <*index*> <*mas1*> <*mas2*> <*mas3*> <*mas7*> **MMU.**<*table*>**.SET** (deprecated)
<*index*>:	TLB entry index. From 0 to (number of TLB entries)-1 of the specified TLB table
<*mas1*>: <*mas2*>: <*mas3*>:	Values corresponding to the values that would be written to the MAS registers in order to set a TLB entry. See the processor's reference manual for details on MAS registers.
<*mas7*>:	MAS7 contains the most significant bits of the physical 36 bit address (e500v2 cores only).

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

CPU specific BenchMarkCounter Commands

The BenchMarkCounter features are based on the core's performance monitor, accessed through the performance monitor registers (PMR). PMC access is only possible while the core is halted.

Notes:

- BMC.PROfile and BMC.SnoopSet are not supported.
- 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.
- Events can be assigned to **BMC.<counter>.EVENT <event>**. For descriptions of available events, please check Freescale's core reference manual.

BMC.FREEZE — Freeze counters while core halted

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

Enable this setting to prevent that actions of the debugger have influence on the performance counter. As this feature software controlled (no on-chip feature), some events (especially clock cycle measurements) may be counted inaccurate even if this setting is set ON.

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.<counter>.SIZE

No function

Format: **BMC.***<counter>***.SIZE** *<size>*

Since only one counter size is possible, this command is only available for compatibility reasons.

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
```

TrOnchip.DISable Disable NEXUS trace register control

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.

TrOnchip.ENable Enable NEXUS trace register control

```
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.RESet Reset on-chip trigger settings

```
Format:      TrOnchip.RESet
```

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

Format:	**TrOnchip.Set** *<event>* [**ON** | **OFF**]

Enables the specified on-chip trigger facility to stop the CPU on below events:

<event>	Description
BRT	Break on branch taken event.
IRPT	Break on interrupt entry.
RET	Break on return from interrupt.
CIRPT	Break on critical interrupt entry.
CRET	Break on return from critical interrupt.
CI	Break on critical input interrupt.
MC	Break on machine check interrupt.
DS	Break on data storage interrupt.
IS	Break on instruction storage interrupt.
EI	Break on external input interrupt.
AL	Break on alignment interrupt.
PR	Break on program interrupt.
FP	Break on fpu unavailable interrupt.
SC	Break on system call.
AU	Break on auxiliary processor unavailable interrupt.
DEC	Break on decrementer interrupt.
FIT	Break on fixed interval timer interrupt.
WD	Break on watchdog interrupt.
DTLB	Break on data TLB error interrupt.
ITLB	Break on instruction TLB interrupt.
DBG	Break on debug interrupt - do not clear if breakpoints are used.
SPEU	Break on SPE APU unavailable interrupt.
SPED	Break on SPE floating-point data interrupt.
SPER	Break on SPE floating-point round interrupt.
PM	Break on performance monitor interrupt.

TrOnchip.VarCONVert   Adjust HLL breakpoint in on-chip resource

Format: **TrOnchip.VarCONVert** [**ON** | **OFF**]

ON (default)

After a data address breakpoint is set to an HLL variable all on-chip breakpoints are spent. As soon as a new data address breakpoint is set the data address breakpoint to the HLL variable 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

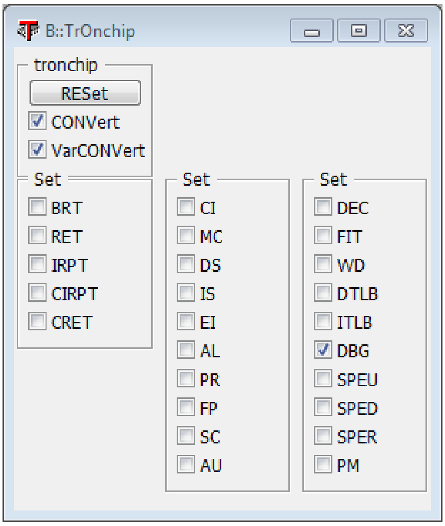
An error message is displayed when the user wants to set a new data address breakpoint after all on-chip breakpoints are spent by a data address breakpoint to an HLL variable.

```
TrOnchip.VarCONVert ON
Var.Break.Set flags
Var.Break.Set ast
Data.View flags
Data.View ast
```

TrOnchip.state

Format:	**TrOnchip.state**

Display the trigger setup dialog window.



☑ CONVert
☑ VarCONVert

□ BRT
□ RET
□ IRPT
□ CIRPT
□ CRET

□ CI
□ MC
□ DS
□ IS
□ EI
□ AL
□ PR
□ FP
□ SC
□ AU

□ DEC
□ FIT
□ WD
□ DTLB
□ ITLB
☑ DBG
□ SPEU
□ SPED
□ SPER
□ PM

Onchip.Mode.IFSel Select interface to be traced

Format:	**Onchip.Mode.IFSel** *<interface>*
<interface>:	**ECM** (processor core interface) **SDRAM** (SDRAM interface) **PCI, PCI2** (PCI controller interface) **RI0** (RapidI0 interface) **PCIEX, PCIEX2, PCIEX3** (PCI Express interface)

Interface selection. Specifies the interface that sources information for both comparison/buffer control and buffer data capture. The availability of certain <interface> options depends on the target processor. Please check the processor user’s manual for which interfaces are available.

JTAG Connector

Mechanical Description

JTAG Connector MPC85XX (COP)

Signal	Pin	Pin	Signal
TDO	1	2	N/C
TDI	3	4	TRST-
(RUNSTOP-)	5	6	JTAG-VREF
TCK	7	8	(CHKSTPIN-)
TMS	9	10	N/C
(SRESET-)	11	12	GND
HRESET-	13	14	N/C (KEY PIN)
(CKSTOPOUT-)	15	16	GND

This is a standard 16 pin double row (two rows of eight pins) connector (pin-to-pin spacing: 0.100 in.). (Signals in brackets are not strong necessary for basic debugging, but its recommended to take in consideration for future designs.)