LAUTERBACH
DEVELOPMENT TOOLS

Training Hexagon ETM Tracing

MANUAL

Training Hexagon ETM Tracing

Training Hexagon ETM Tracing

Version 06-Jun-2024

Introduction Hexagon ETM

The Hexagon ETM can export trace information

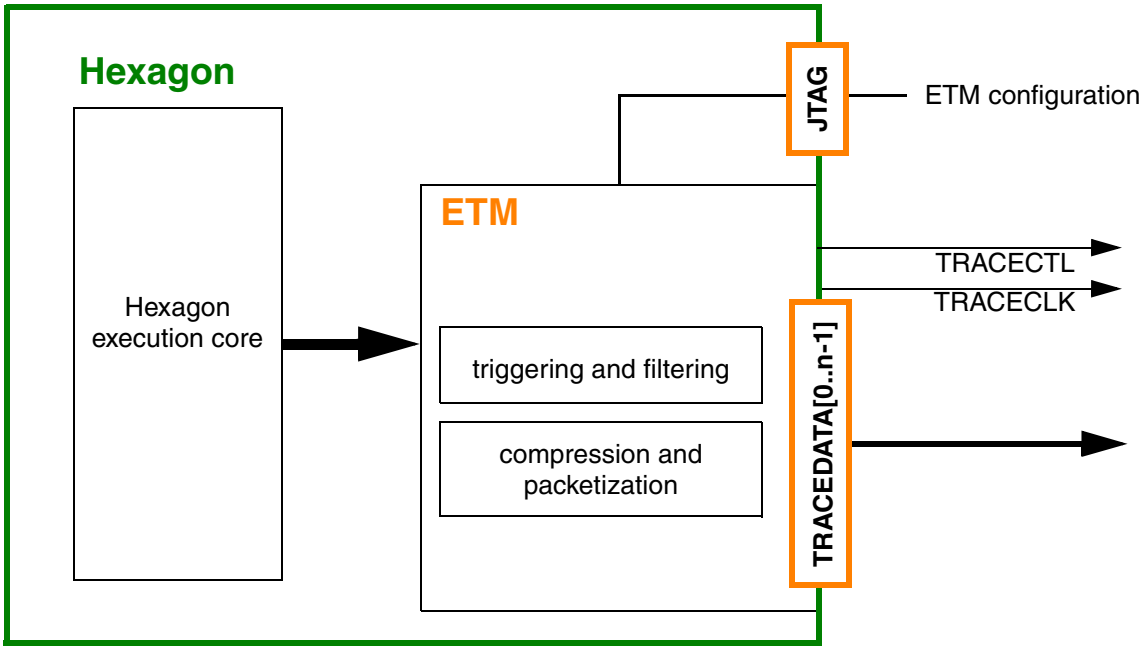
- Off-chip via dedicated pins for recording by TRACE32 PowerTrace.
- To the on-chip trace memory called ETB (Embedded Trace Buffer). The ETB has a size of 2 KB and can store 512 entries, each 32-bits wide.

The Hexagon is using the ETMv3 protocol.

Off-chip Trace Port

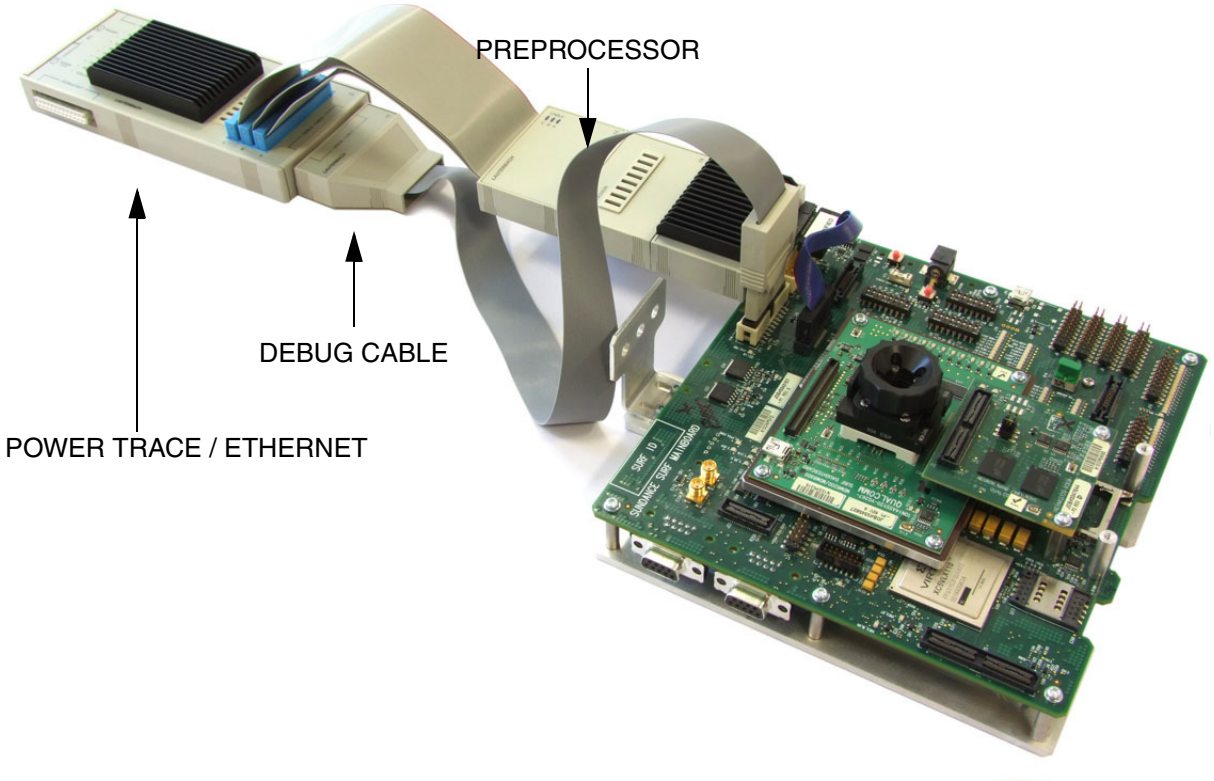
The trace information exported by the Hexagon ETM is captured by TRACE32 and recorded into the trace memory of the PowerTrace hardware.

The trace memory within the PowerTrace is maintained by the TRACE32 command group **Analyzer.<sub_cmd>**.

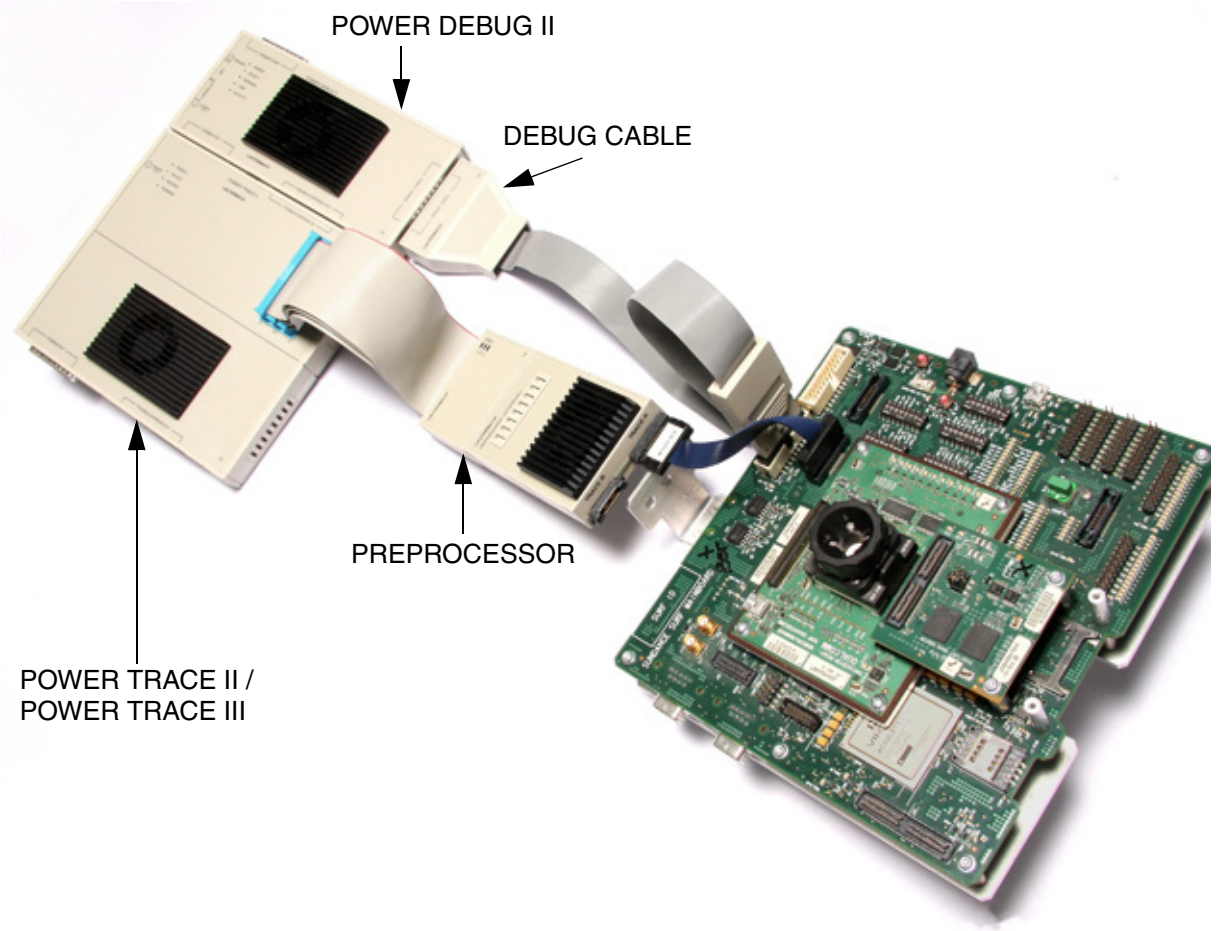


The following TRACE32 hardware is required to record and analyze trace information exported off-chip:

- POWER TRACE / ETHERNET
- DEBUG CABLE
- PREPROCESSOR



- POWER DEBUG II and POWER TRACE II / POWER TRACE III
- DEBUG CABLE
- PREPROCESSOR



The trace memory within the PowerTrace contains trace information for all hardware threads

Trace packet from hardware thread 1
Trace packet from hardware thread 1
Trace packet from hardware thread 2
Trace packet from hardware thread 3
Trace packet from hardware thread 5
Trace packet from hardware thread 5
Trace packet from hardware thread 4
Trace packet from hardware thread 2
Trace packet from hardware thread 2
Trace packet from hardware thread 0
Trace packet from hardware thread 0
Trace packet from hardware thread 0

The **Analyzer.List** command displays the trace information for all hardware threads.

```
Analyzer.List                           ; Display a trace listing for
                                        ; all hardware threads
```

Trace Display/Evaluation for a Single Hardware Thread

Alternatively TRACE32 provides the possibility to display/evaluate the trace information for a single hardware thread via the option **/CORE <number>**.

Analyzer.<*sub_cmd*> **/CORE 0**
Analyzer.<*sub_cmd*> **/CORE 1** etc.

```
Analyzer.List /CORE 0                  ; Display a trace listing for
                                       ; hardware thread 0
```

The aim of the following start-up sequence is:

- To set up the ETM to export a maximum of trace information (full trace port width, maximum trace speed)
- To configure the TRACE32 recording tool for an error-free recording

TRACE32 provides the following commands for enabling the ETM:

PER.Set.simple *<address>* [*<format>*] *<value>*

Data.Set *<address>* [*<format>*] *<value>*

Starting-up the ETM requires the following steps:

1. **Enable the ETM.**

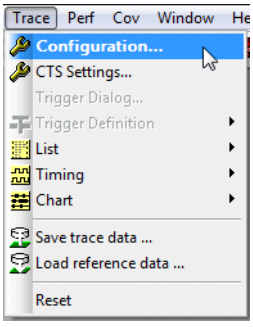
   Enabling the ETM is done by writing to memory-mapped configuration registers. For details, refer to your Hexagon manual.

```
; Write the 32-bit value 0x00000002 in little endian mode to the
; configuration register at address 0xA9000208
PER.Set.simple 0xA9000208 %LE %Long 0x2

; Write the 32-bit value 0x00000001 in little endian mode to the
; address 0xA8100000
Data.Set 0xA8100000 %LE %Long 0x1
```

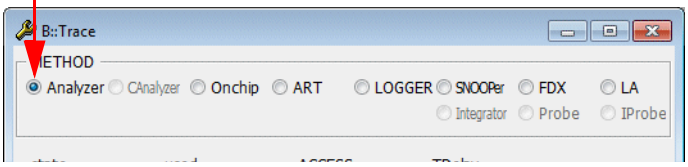
2. **Enable the trace port pins for your target hardware.**

   Enabling the trace port pins for the ETM is done by likewise writing to memory-mapped configuration registers. Refer to your Hexagon manual for details.

3. **Select Analyzer as TRACE32 trace method.**



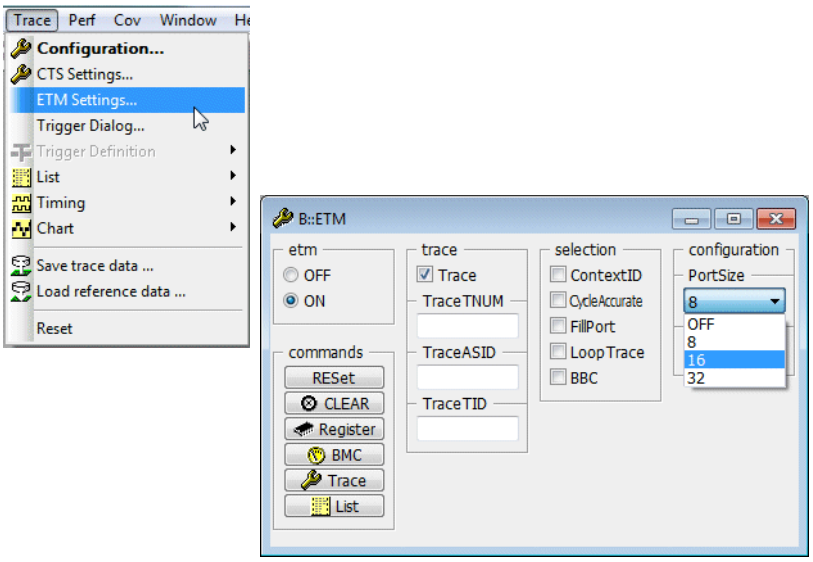
Select trace method Analyzer




```
Trace.METHOD Analyzer                   ; Default if a TRACE32 pre-
                                        ; processor hardware is
                                        ; connected (see page 5)
```

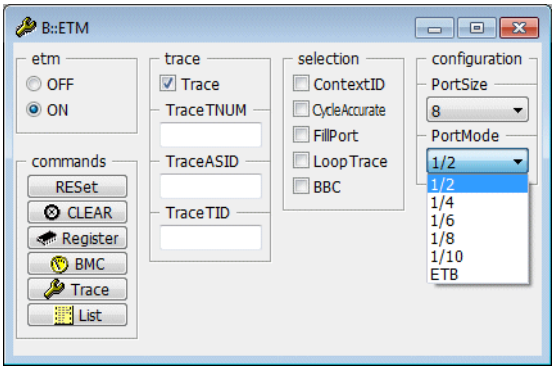
This setting informs TRACE32 that you want to use off-chip tracing.

4. **Define the ETM port size for the off-chip tracing.**



By defining the ETM port size you inform TRACE32 how many TRACEDATA pins are used on your target hardware to export the trace packets. Please refer to your target hardware's schematics to get the number of TRACEDATA pins.

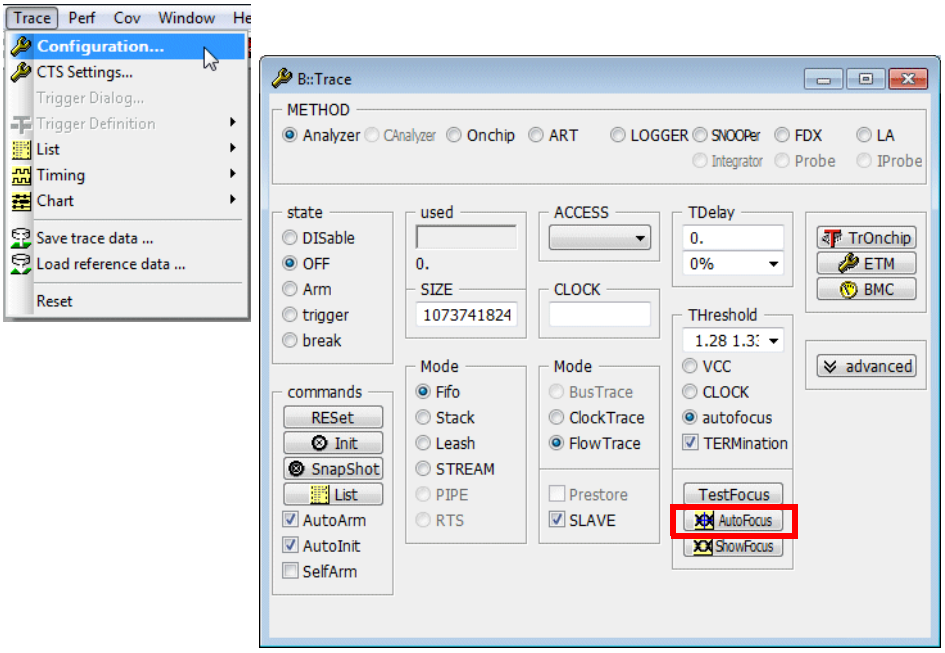
5. **Define the ETM port mode for the off-chip tracing.**



By defining the ETM port mode you inform TRACE32 about the TRACECLK (trace clock). Please refer to your target hardware description for the trace clock information.

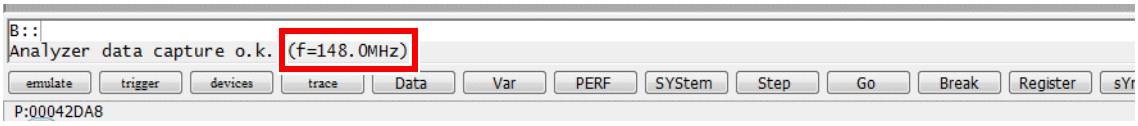
For the Hexagon ETM the trace clock is always a divided core clock.

6. Calibrate the TRACE32 recording hardware.



Push the **AutoFocus** button to set up the recording tool.

If the calibration is performed successfully, the following message will be displayed:



(f=148.MHz) displays the *<trace_port_frequency>*.

The *<core_clock>* can be calculated out of the *<trace_port_frequency>* as follows:

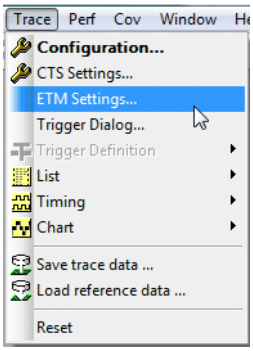
<core_clock>= 2 * *<trace_port_frequency>* * (1/*<port_mode>)*

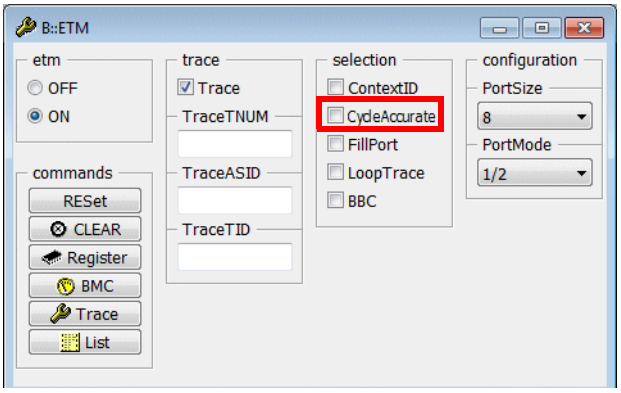
e.g. *<core_clock>* = 2 * 148MHz *(1 / 1/2) = 148MHz * 4 = 592MHz

For details on the calibration of the TRACE32 recording tool, refer to "**Appendix A**".

Example for a start-up script:

```
;…                                  ; Setup for the Hexagon debugger

PER.Set.simple …                    ; Enable the ETM and the trace port

Trace.METHOD Analyzer               ; Select "Analyzer" as trace method

Analyzer.RESet                      ; Reset the "Analyzer"

ETM.RESet                           ; Reset ETM

ETM.CLEAR                           ; Reset ETM registers

ETM.PortSize 16.                    ; Target system provides 16 pins
                                    ; for TRACEDATA

ETM.PortMode 1/2                    ; Target system is using
                                    ; 1/2 <core_clock> as trace clock

Analyzer.AutoFocus                  ; Calibrate the TRACE32 recording
                                    ; tool

;…
```





If **ETM.CycleAccurate** is **OFF**, trace recording and time stamping is done as follows:

ETM is exporting the **addresses of the executed instructions** in form of trace packets

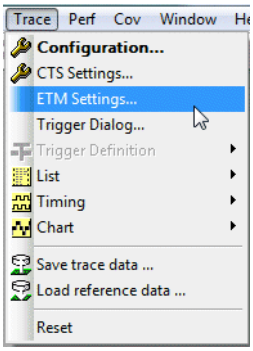
trace packets	timestamp
trace packets	timestamp
trace packets	timestamp
trace packets	timestamp
trace packets	timestamp
trace packets	timestamp
trace packets	timestamp
trace packets	timestamp

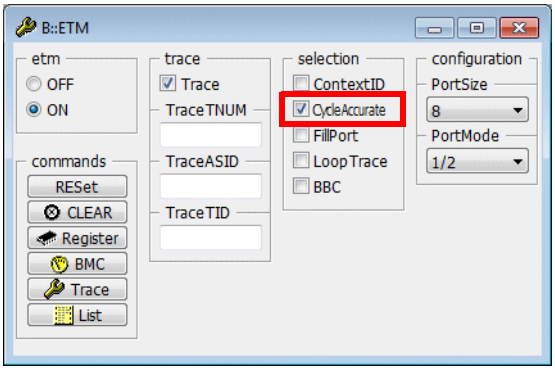
The **TRACE32** recording tool
- collects the trace packets
- stores the trace packets into the trace memory
- **timestamps the trace packets**

```
;…

ETM.CycleAccurate OFF

ETM.FillPort OFF                        ; Trace packets are organized in
                                        ; bytes

                                        ; As soon as a trace packet is
                                        ; available, it is exported

;…
```

The resolution of the timestamp is:

- 10 ns if a POWER TRACE / ETHERNET is used
- 5 ns if a POWER TRACE II / POWER TRACE III is used





If **ETM.CycleAccurate** is **ON** trace recording and time stamping is done as follows:

ETM is exporting the **addresses of the executed instructions** and the **number of stalls** between the instructions in form of trace packets

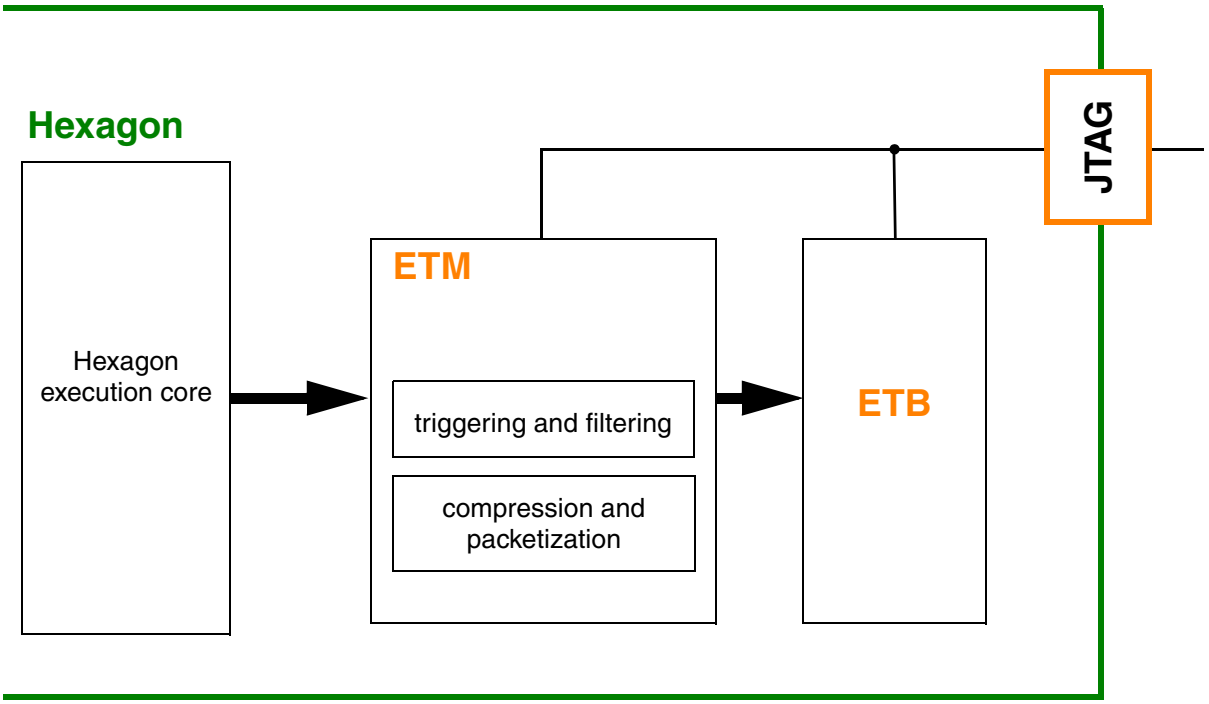
trace packets	
trace packets	
trace packets	
trace packets	
trace packets	
trace packets	
trace packets	
trace packets	

The TRACE32 recording tool
- collects the trace packets
- stores the trace packets

TRACE32 is generating the time information for the trace display out of the exported trace information and the *<core_clock>* provided by the command **Analyzer.CLOCK**.

Cycle accurate tracing provides a more detailed timing and allows a higher density of trace packets in the trace memory, but generates a higher load on the trace port.

```
Analyzer.CLOCK 600.MHz                  ; Inform TRACE32 about the
                                        ; core clock

ETM.CycleAccurate ON

(ETM.FillPort ON)                       ; Automatically switched to ON if
                                        ; cycle accurate tracing is ON

                                        ; The ETM collects the trace
                                        ; packets and exports them as
                                        ; soon as TRACEDATA/8 packets are
                                        ; available
```

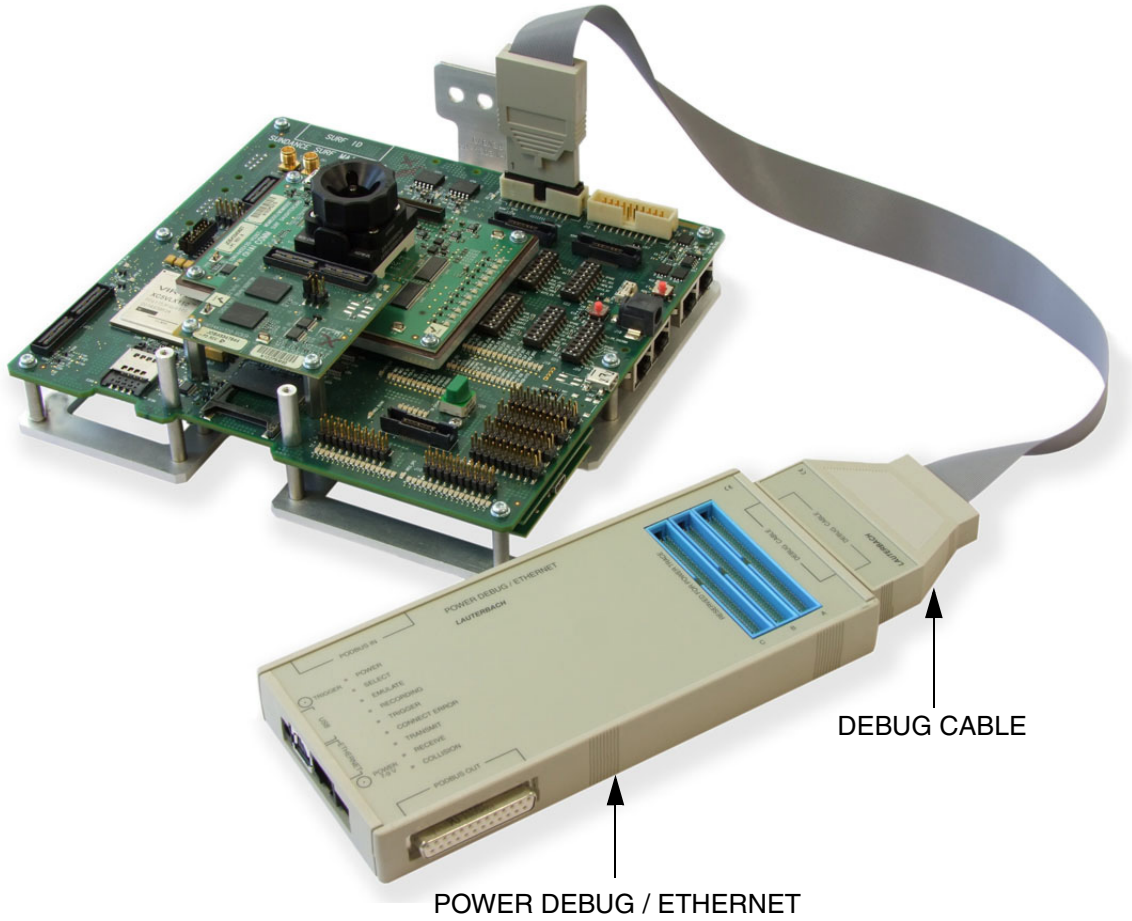


The trace information exported by the Hexagon ETM is stored in the on-chip trace memory (ETB).

The ETB is maintained by the TRACE32 command group **Onchip.<sub_cmd>**.

The following TRACE32 hardware is sufficient to analyze the trace information piped into the ETB:

- POWER DEBUG / ETHERNET
- DEBUG CABLE



The ETB contains
trace information for all hardware threads

Trace packet from hardware thread 1
Trace packet from hardware thread 1
Trace packet from hardware thread 2
Trace packet from hardware thread 3
Trace packet from hardware thread 5
Trace packet from hardware thread 5
Trace packet from hardware thread 4
Trace packet from hardware thread 2
Trace packet from hardware thread 2
Trace packet from hardware thread 0
Trace packet from hardware thread 0
Trace packet from hardware thread 0

The command **Onchip.List** displays the trace information for all hardware threads:

```
Onchip.List                             ; Display a trace listing for
                                        ; all hardware threads
```

Trace Display/Evaluation for a Single Hardware Thread

Alternatively TRACE32 provides the possibility to display/evaluate the trace information for a single hardware thread via the option **/CORE** *<number>*.

Onchip.*<sub_cmd>* **/CORE 0**
Onchip.*<sub_cmd>* **/CORE 1** etc.

```
Onchip.List /CORE 0                     ; Display a trace listing for
                                        ; hardware thread 0
```

TRACE32 provides the following commands for enabling the ETM:

PER.Set.simple *<address>* [*<format>*] *<value>*

Data.Set *<address>* [*<format>*] *<value>*

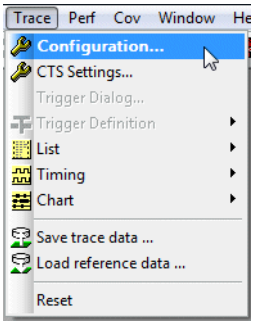
Starting-up the ETM requires the following steps:

1. **Enable the ETM.**

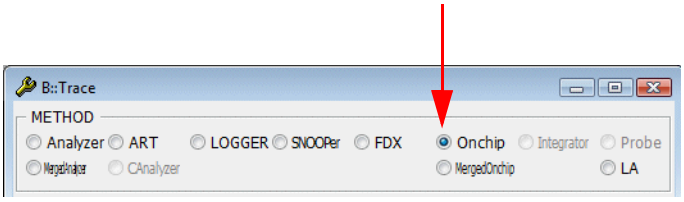
   Enabling the ETM is done by writing to memory-mapped configuration registers. Refer to your Hexagon manual for details.

```
; Write the 32-bit value 0x00000002 in little endian mode to the
; configuration register at address 0xA9000208
PER.Set.simple 0xA9000208 %LE %Long 0x2

; Write the 32-bit value 0x00000001 in little endian mode to the
; address 0xA8100000
Data.Set 0xA8100000 %LE %Long 0x1
```

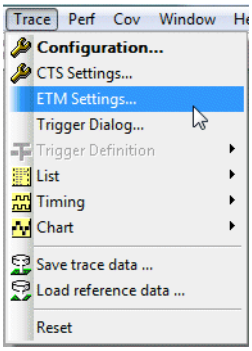
2. **As soon as the trace method Onchip is selected, all settings for the ETB are automatically done by TRACE32.**

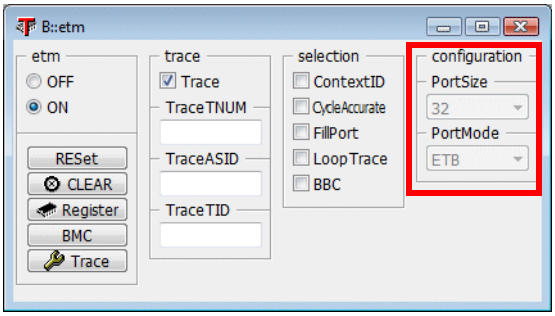


Select trace method Onchip




```
Trace.METHOD Onchip                    ; Default if no TRACE32 pre-
                                       ; processor hardware is
                                       ; connected (see page 17)
```





automated setup

Example for a start-up script:

```
;…                                  ; Setup for the Hexagon debugger

PER.Set.simple …                    ; Enable the ETM and the ETB

Trace.METHOD Onchip                 ; Select "Onchip" as trace method

Onchip.RESet                        ; Reset the Onchip trace

ETM.RESet                           ; Reset ETM

ETM.CLEAR                           ; Reset ETM registers

;…
```

ETM is exporting
trace packets

trace packet	
trace packet	
trace packet	
trace packet	
trace packet	
trace packet	
trace packet	
trace packet	

Trace information within the ETB is never time-stamped.

FillPort is automatically enabled for the ETB.

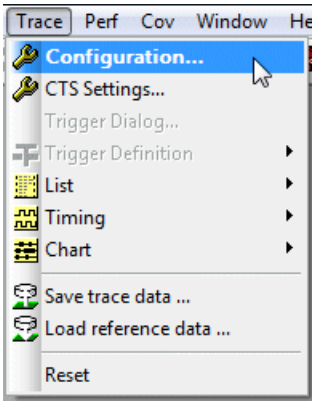
In order to get timing information, **CycleAccurate** tracing needs to be enabled (not fully supported yet).

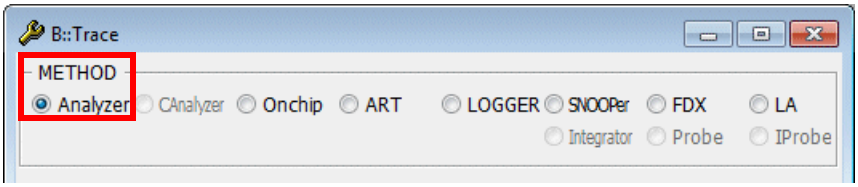
```
;…

Onchip.CLOCK 600.MHz                  ; Inform TRACE32 about the core
                                      ; clock

ETM.CycleAccurate ON

;…
```

Specifying the Trace Method

Specifying the trace method has three effect:

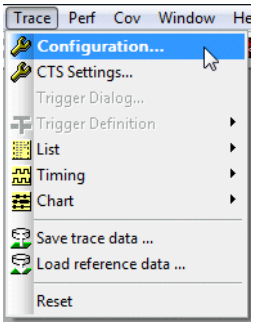
1. Selection of the trace repository.
2. Admit the command group **Trace.<sub_cmd>** as an alias.
3. Program TRACE32 to use the trace information from the specified trace repository as source for various trace evaluation commands.



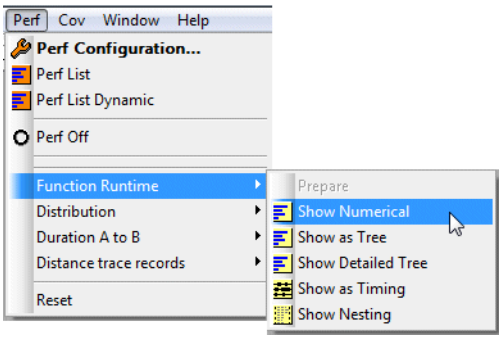



```
Trace.METHOD Analyzer                   ; Trace repository is the trace
                                        ; memory of the TRACE32 PowerTrace
```

```
                                        ; Trace is used as an alias for
                                        ; Analyzer

Trace.List                              ; Means Analyzer.List
```



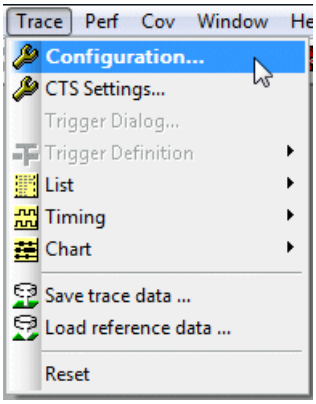
All commands in the **Trace** menu apply to Analyzer

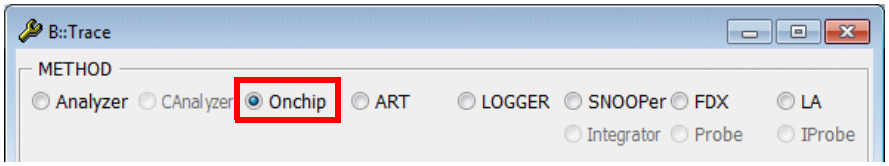


All **Function Runtime** commands apply to Analyzer

The following commands analyze trace information stored into the PowerTrace hardware:

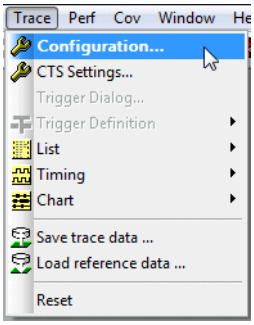
```
CTS.List                              ; Read the trace information from
                                      ; Analyzer and provide a high-level
                                      ; language trace display

COVerage.List                         ; Read the trace information from
                                      ; Analyzer and list which code
                                      ; ranges were executed.

ISTATistic.List                       ; Read the trace information from
                                      ; Analyzer and provide an detailed
                                      ; instruction statistic

MIPS.PROfileChart.sYmbol              ; Read the trace information from
                                      ; Analyzer and provide a MIPS
                                      ; analysis for all executed
                                      ; functions

BMC.List                              ; Read the trace information from
                                      ; Analyzer, display the instruction
                                      ; flow including the benchmark
                                      ; counters
```



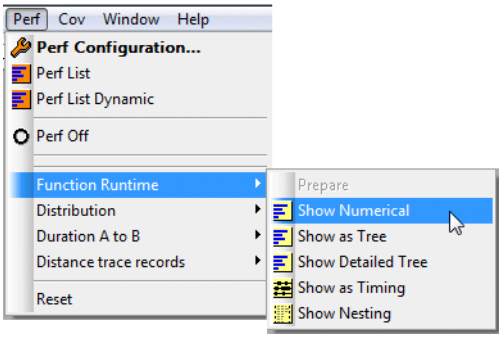



```
Trace.METHOD Onchip                    ; Trace repository is the ETB
```

```
                                       ; Trace is used as an alias for
                                       ; Onchip

Trace.List                             ; Means Onchip.List
```



All commands in the **Trace** menu apply to Onchip.



All **Function Runtime** commands apply to Onchip.

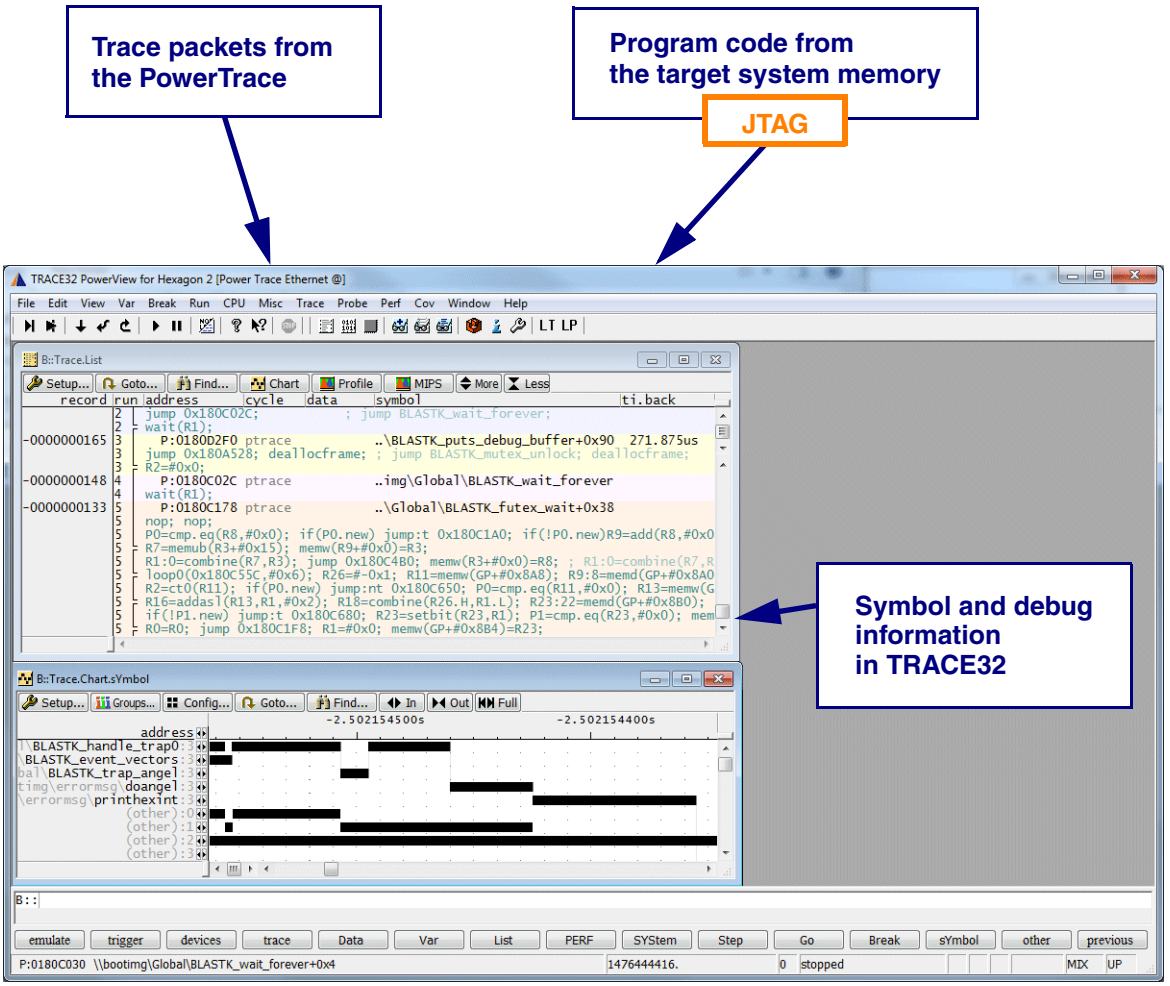
The following commands analyze trace information stored into the ETB:

```
CTS.List                              ; Read the trace information from
                                      ; Onchip and provide a high-level
                                      ; language trace display

COVerage.List                         ; Read the trace information from
                                      ; Onchip and list which code
                                      ; ranges were executed.

ISTATistic.List                       ; Read the trace information from
                                      ; Onchip and provide an detailed
                                      ; instruction statistic

MIPS.PROfileChart.sYmbol              ; Read the trace information from
                                      ; Onchip and provide a MIPS
                                      ; analysis for all executed
                                      ; functions

BMC.List                              ; Read the trace information from
                                      ; Onchip, display the instruction
                                      ; flow including the benchmark
                                      ; counters
```

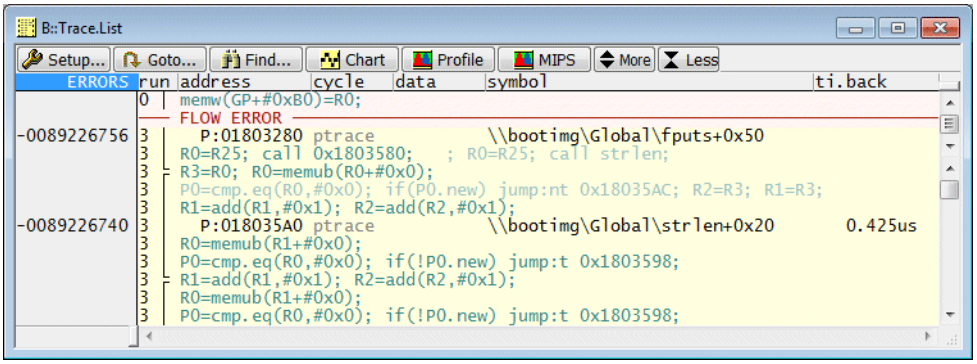
Description

In order to provide an intuitive trace display the following sources of information are merged:

- The trace packets stored in the trace memory of the PowerTrace or the ETB. The trace packets provide only the addresses of the executed instruction packets (instruction flow).
- The program code from the target memory read via JTAG.
- The symbol and debug information already loaded to TRACE32.



If the program code does not match the captured instruction flow, FLOW ERROR is displayed:



Such an error can have the following reasons:

- The program code in the target memory has changed (e.g. by a faulty pointer)
- The off-chip trace recording is not working correctly (e.g. a single trace pin is permanently 0)

FLOW ERROR indicates that the trace information is not reasonable. Please solve problems first and then continue to analyze/evaluate your trace information.

Diagnosis

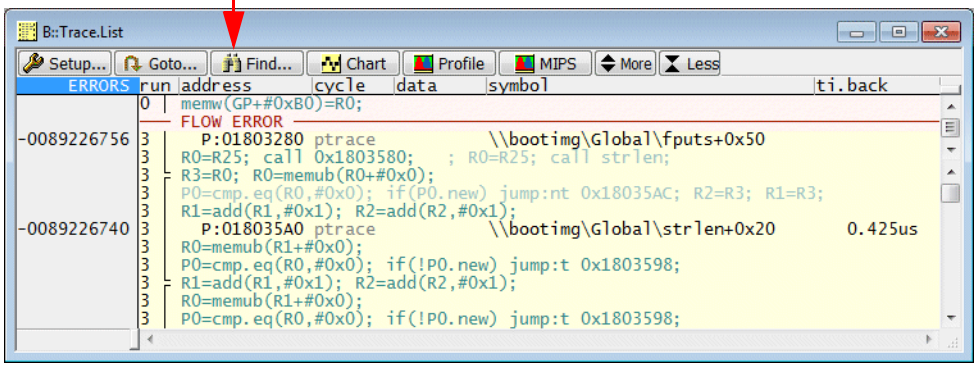
In order to provide the user information quickly, TRACE32 uploads only a specific number of trace records (currently 50 000). Thus FLOW ERRORs are not always detected immediately.

For a FLOW ERROR detection for off-chip tracing proceed as follows:

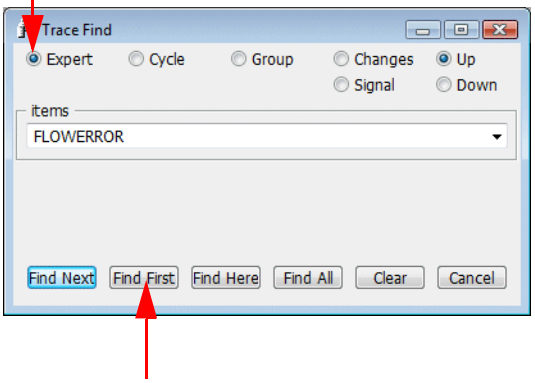
```
Analyzer.FLOWPROCESS                          ; Upload the complete trace
                                              ; contents from the PowerTrace
                                              ; to the host and merge it
                                              ; with the
                                              ; program code/debug
                                              ; information

PRINT %Decimal Analyzer.FLOW.ERRORS()         ; Print the number of FLOW
                                              ; ERRORs as a decimal number
```

To inspect single FLOW ERRORs proceed as follows:

Push the **Find...** button



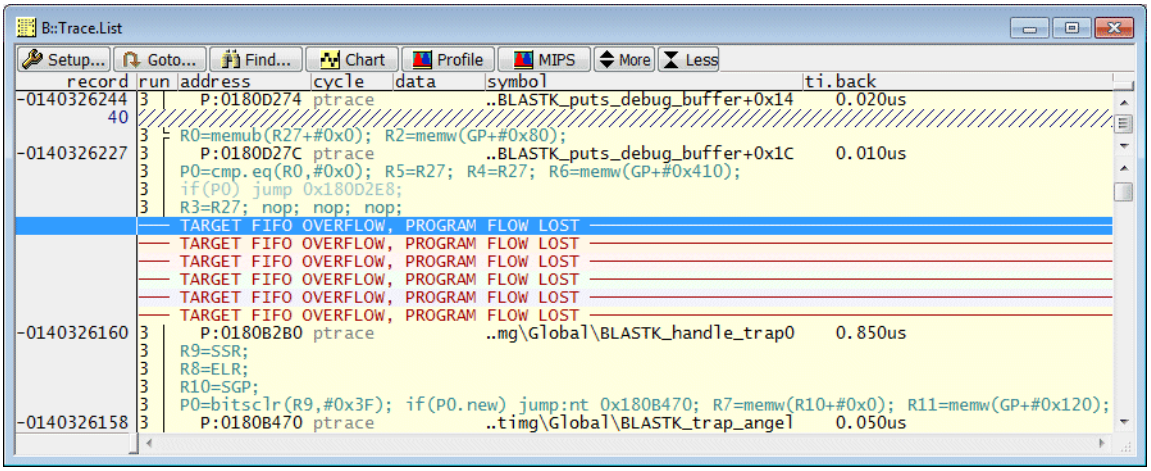
Type FLOWERROR into the Expert window and push the appropriate Find button



TARGET FIFO OVERFLOW

Description

If more trace packets are generated than the ETM can export, the FIFO buffer within the ETM can overflow and some trace packets can be lost. If this is the case TARGET FIFO OVERFLOW, PROGRAM FLOW LOST is displayed:



TARGET FIFO OVERFLOWs indicate that trace packets are lost. TARGET FIFO OVERFLOWs are likely to happen if cycle accurate tracing is used.

All commands that analyze the function nesting are sensitive with regards to TARGET FIFO OVERFLOWs!

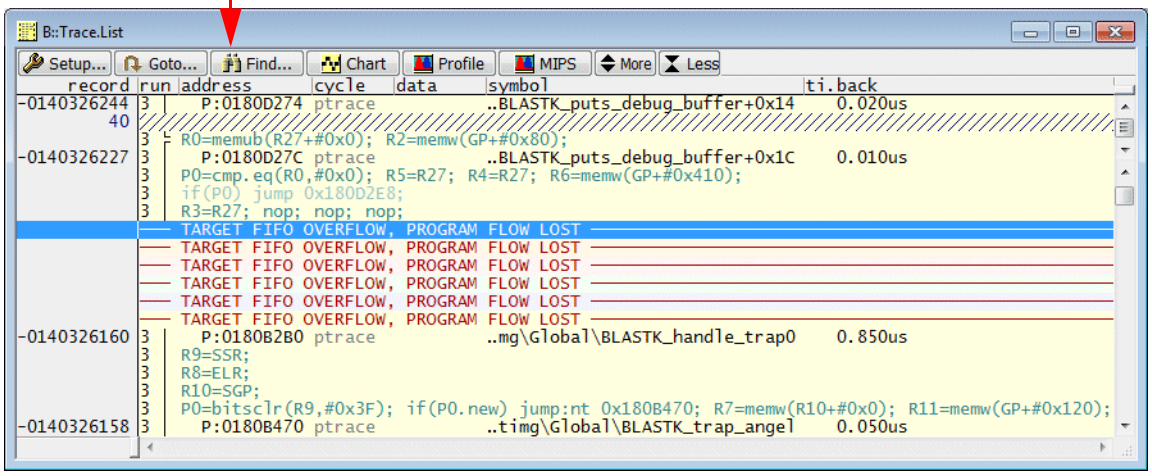
In order to provide the user information quickly, TRACE32 uploads only a specific number of trace records (currently 50 000). Thus TARGET FIFO OVERFLOWs are not always detected immediately.

For a TARGET FIFO OVERFLOW detection for off-chip tracing proceed as follows:

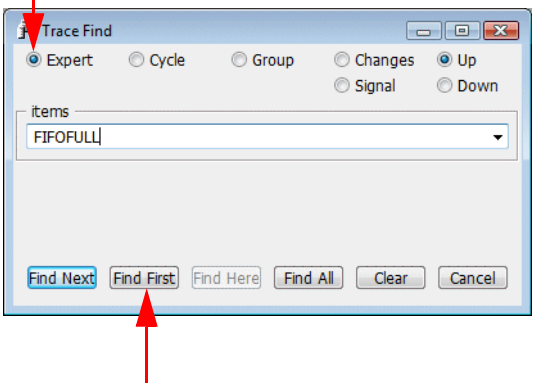
```
Analyzer.FLOWPROCESS                          ; Upload the complete trace
                                              ; contents from the PowerTrace
                                              ; to
                                              ; the host and merge it with
                                              ; the program code/debug
                                              ; information

PRINT %Decimal Analyzer.FLOW.FIFOFULL()       ; Print the number of TARGET
                                              ; FIFO
                                              ; OVERFLOWs as a decimal
                                              ; number
```

To inspect single TARGET FIFO OVERFLOWs proceed as follows:

Push the **Find...** button



Type FIFOFULL into the Expert window and push the appropriate Find button

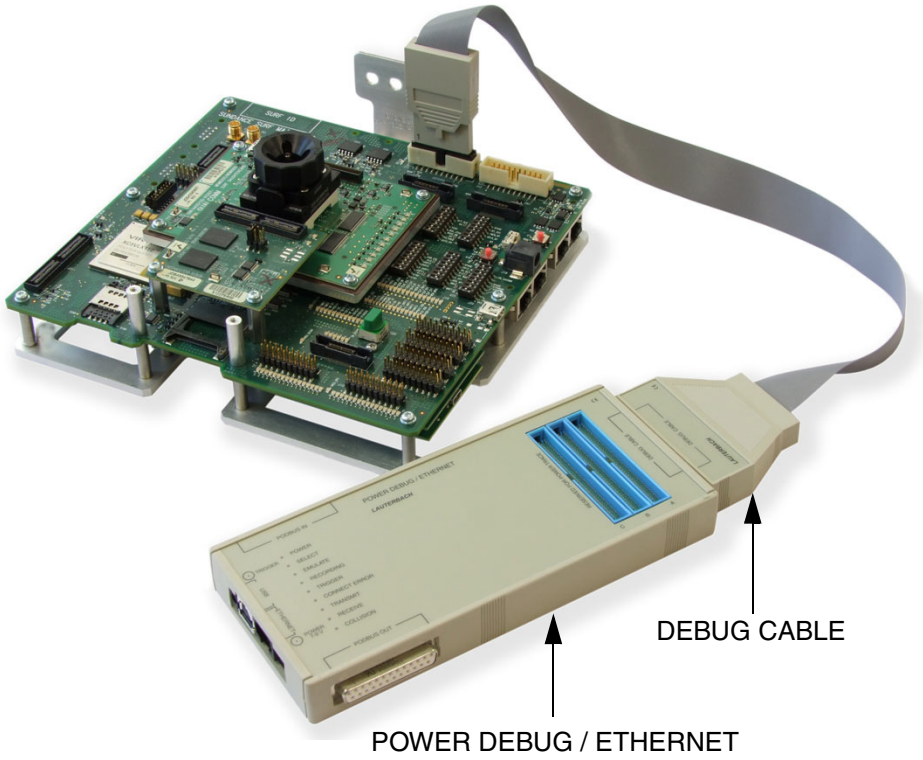


Introduction

TRACE32 Hardware Configuration

The following TRACE32 hardware is sufficient to use ETM based real-time breakpoints:

- POWER DEBUG / ETHERNET
- DEBUG CABLE



Requirements

In order to use ETM based real-time breakpoints, the ETM has to be enabled. For details refer to:

- **"Basic Start-Up Sequence"** (training_hexagon_etm.pdf) on **page 10** or
- **"Basic Start-up Sequence"** (training_hexagon_etm.pdf) on **page 21**.

The examples in this section are given on the assumption, that you are familiar with the breakpoint handling in TRACE32.

If your aren't, please refer to the chapters "**Breakpoints**" and "**Breakpoint Handling**" in **"Training Basic Debugging"** (training_debugger.pdf).

Hint

ETM based real-time breakpoints can be set while the program execution is running.

Complex Program Breakpoints

Complex breakpoint: Stop the program execution after n hits of a program breakpoint.

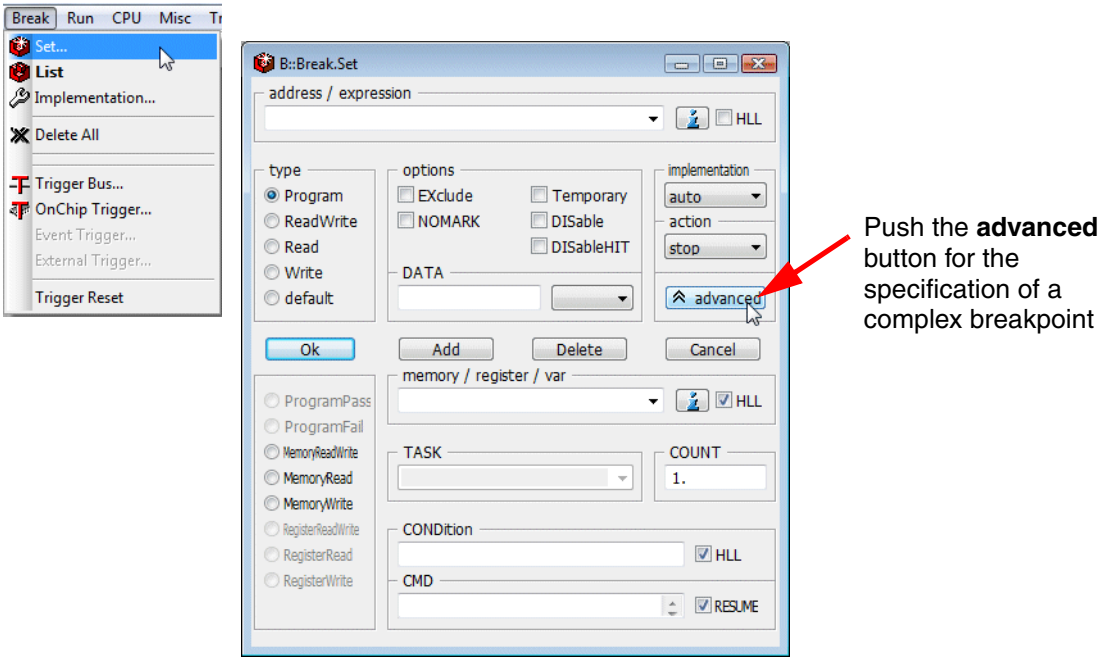
To illustrate the handling of complex program breakpoints, the following examples are provided:

- Example 1: Stop the program execution at the nth call of a particular function.
- Example 2: Stop the program execution at the nth call of a particular function in a particular hardware thread.

Example 1

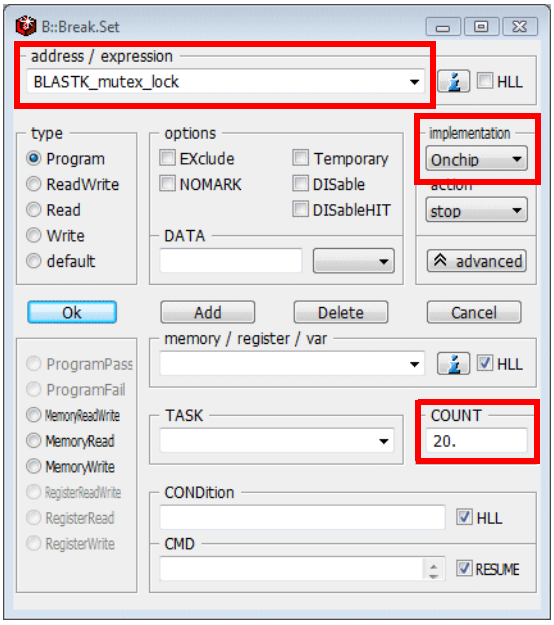
Stop the program execution at the 20th call of the function *BLASTK_mutex_lock* (etm_break1.cmm).

1. Choose **Break** menu > **Set**.



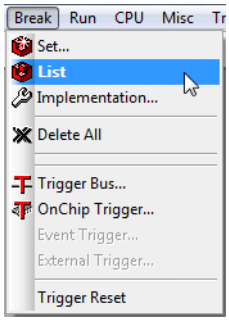
Push the **advanced** button for the specification of a complex breakpoint

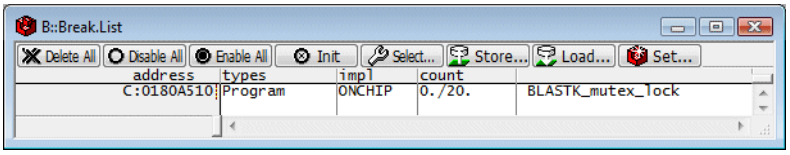
2. Specify the breakpoint.



- Specify the program address in the **address / expression** field.
- Specify the **implementation** Onchip.
- Specify the **COUNT**er value.

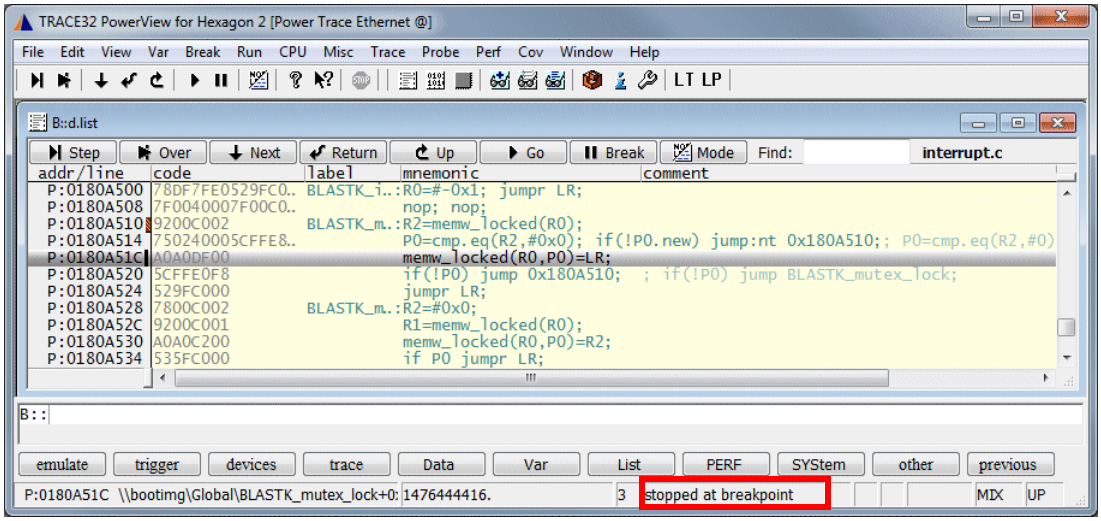
3. Display a breakpoint listing.



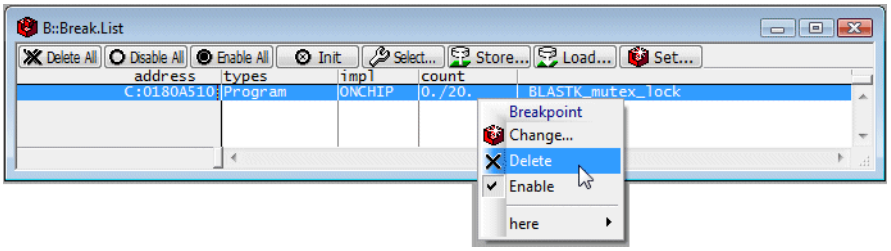


4. Start the program execution.

5. ETM-based breakpoints are not cycle-exact, some logic needs to be passed in order to stop the program execution. As a result the program execution stops shortly after the specified event.



6. Delete the breakpoint when you are done with your test.

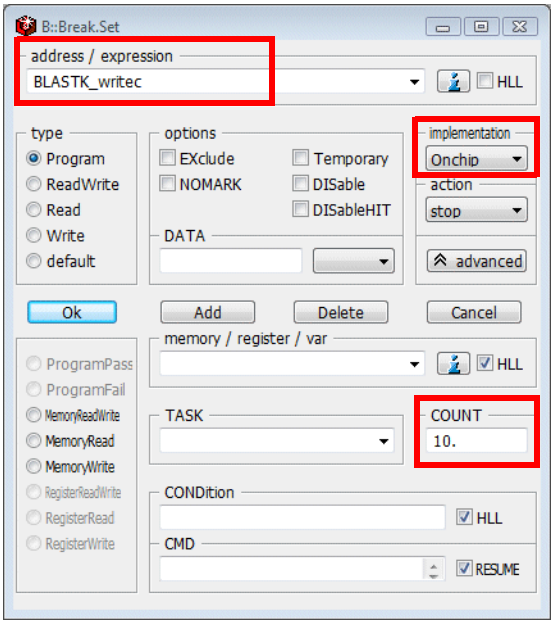



```
; Display a source listing
List

; Display a break listing
Break.List

; Set breakpoint, select symbol via symbol browser
; Break.Set * /Program /Onchip /COUNT 20.

; Set the breakpoint
Break.Set BLASTK_mutex_lock /Program /Onchip /COUNT 20.

; Start the program execution
Go

; …

; Delete breakpoint
Break.Delete BLASTK_mutex_lock
```

Example 2

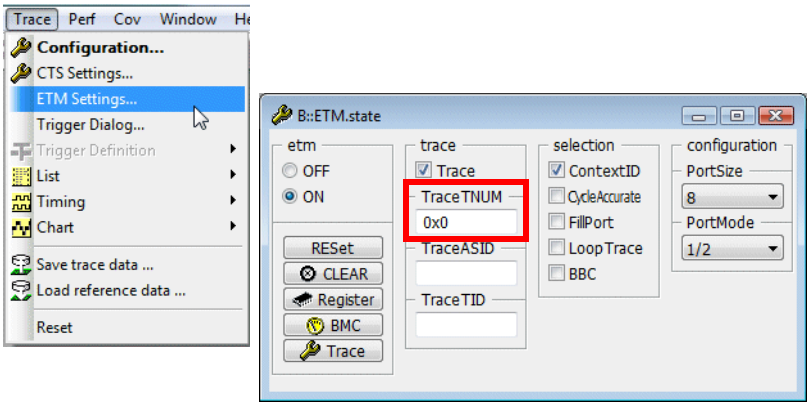
Stop the program execution at the 10th call of the function *BLASTK_writec* in hardware thread 0x0 (etm_break2.cmm).

1. Specify the breakpoint.

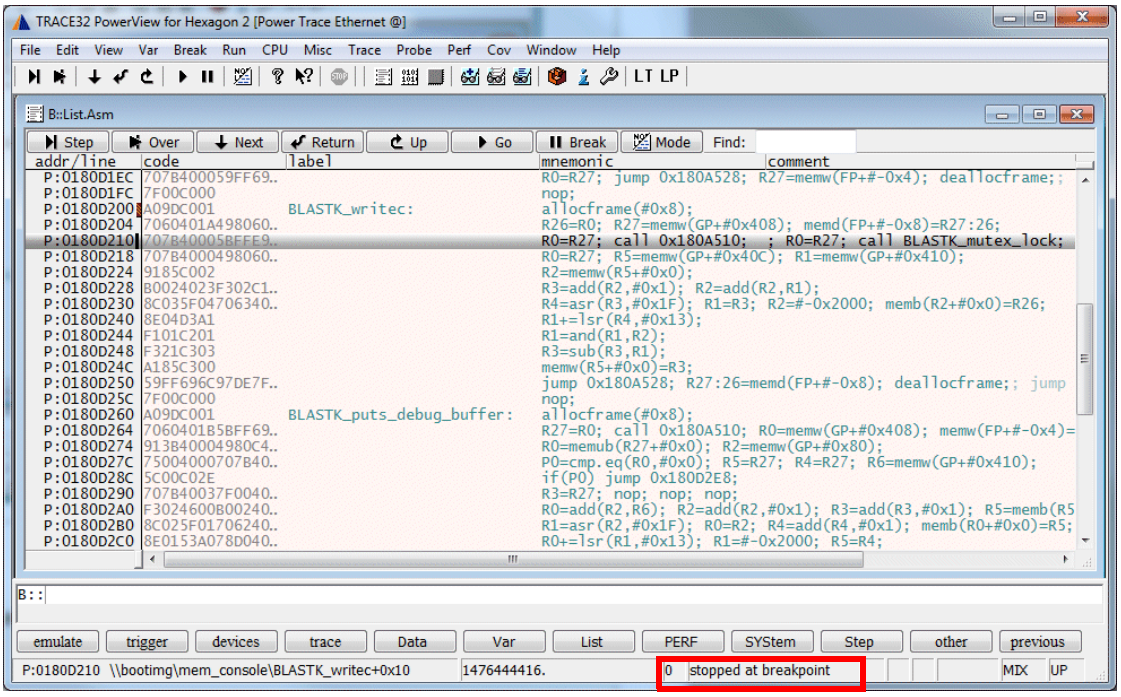


- Specify the program address in the **address / expression** field.
- Specify the **implementation** Onchip
- Specify the **COUNT**er value.

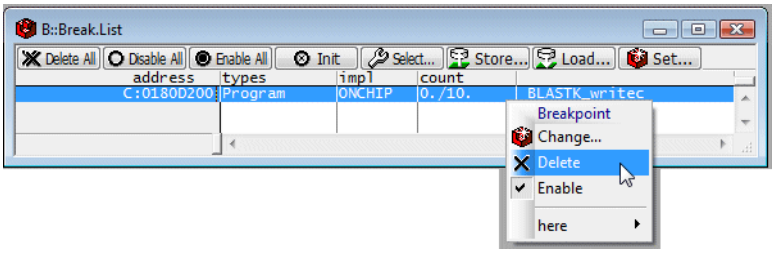
2. Specify the hardware thread in the **ETM.state** window.

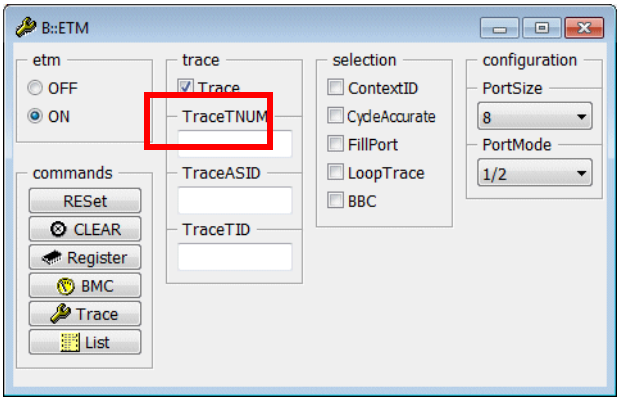


3. Start the program execution.



4. Delete the breakpoint and remove the hardware thread selection when you are done with your test.





```
; Set the breakpoint
Break.Set BLASTK_writec /Program /Onchip /COUNT 10.

; Display the ETM settings
ETM.state

; Specify hardware thread 0x0 for the breakpoint and the trace
; exporting
ETM.TraceTNUM 0x0

Go

; …

; Delete breakpoint
Break.Delete BLASTK_writec

; Remove hardware thread setting
ETM.TraceTNUM
```

Summary

Use the following command to stop the program execution after the specified instruction was executed a specified number of times. You can specify up to 4 to single instruction addresses and up to 4 instruction address ranges.

Break.Set *<address>* | *<range>* **/Program /Onchip /COUNT** *<number>*

Complex data breakpoint: Stop the program execution after the specified address was read/written, specification of data value possible.

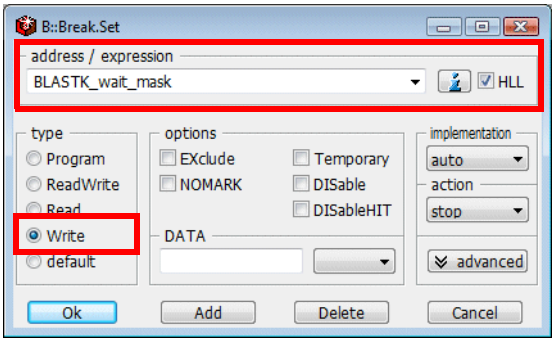
To illustrate the handling of complex data breakpoints, the following examples are provided:

- Example 1: Stop the program execution after a write access to a specific integer variable.
- Example 2: Stop the program execution after a specific value was written to a specific integer variable.
- Example 3: Stop the program execution after a specific data value was written to a specified address n-times.

Example 1 - Complex Data Breakpoints

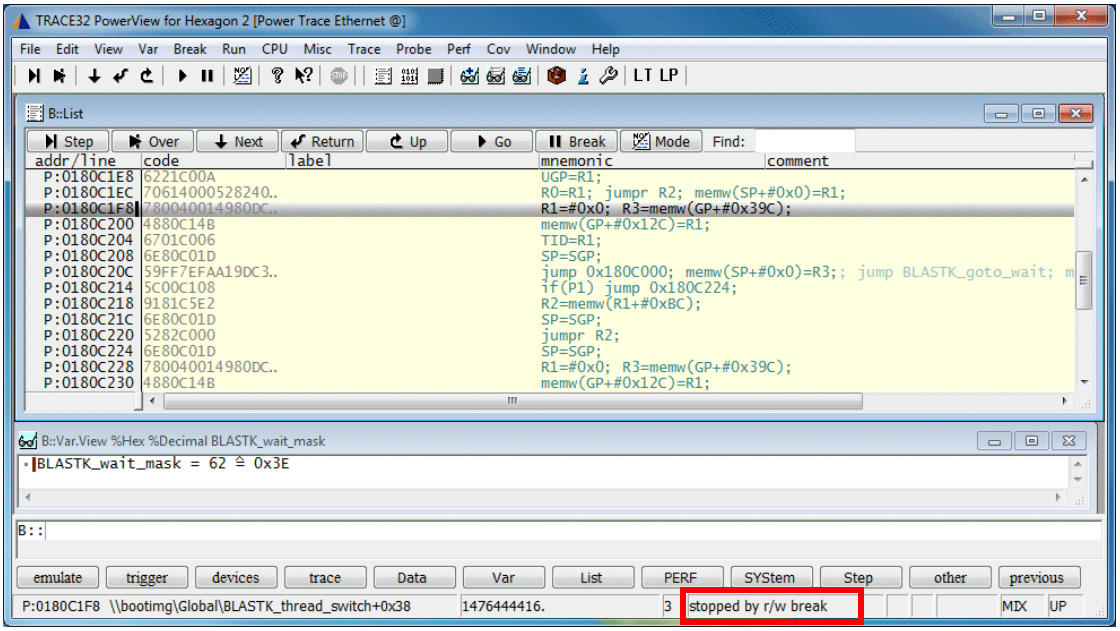
Stop the program execution after a write access to the integer variable *BLASTK_wait_mask* (etm_break3.cmm).

1. Specify the breakpoint.



   - Specify the variable in the **address / expression** field and enable the **HLL** check box.
   - Specify **Write** as breakpoint type.

2. Start the program execution.



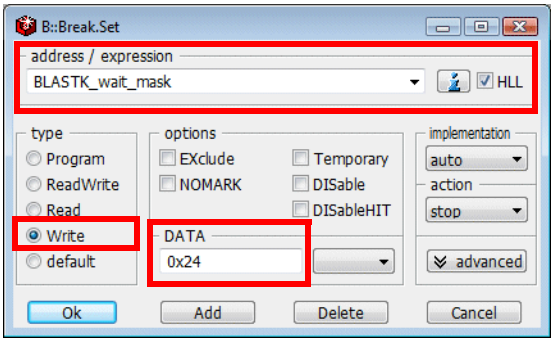
NOTE:	The instruction that performed the write access and so caused the program stop, cannot be detected automatically since • ETM-based breakpoints are not cycle-exact • register indirect addressing is used

```
Var.View %Hex %Decimal BLASTK_wait_mask     ; Display contents of variable
                                            ; BLASTK_wait_mask

Var.Break.Set BLASTK_wait_mask /Write       ; Set the breakpoint

Go                                          ; Start the program execution
```

Example 2 - Complex Data Breakpoints

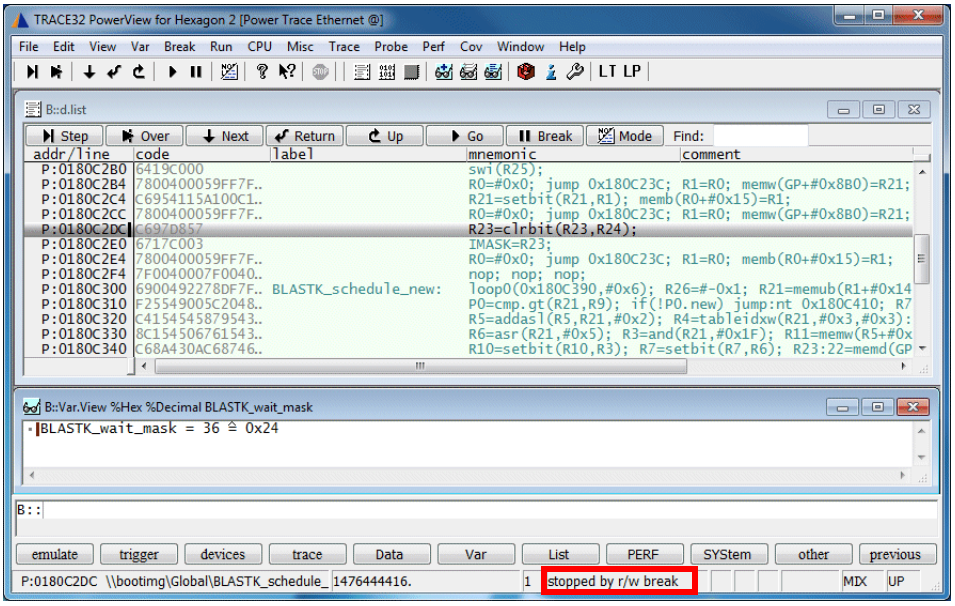
Stop the program execution after the value 0x24 was written to the integer variable *BLASTK_wait_mask* (etm_break4.cmm).

1. Specify the breakpoint.



- Specify the variable in the **address / expression** field and enable the **HLL** check box.
- Specify **Write** as breakpoint type.
- Specify the **DATA** value.

2. Start the program execution.




```
Var.Break.Set BLASTK_wait_mask /Write /DATA.auto 0x24

Go
```

; Set memory access breakpoint, data value possible
; (up to 4 accesses to single addresses, up to 2 accesses to address ranges)

Break.Set *<address>* | *<range>* **/ReadWrite** | **/Read** | **/Write**
Var.Break.Set *<hll_expression>* **/ReadWrite** | **/Read** | **/Write**

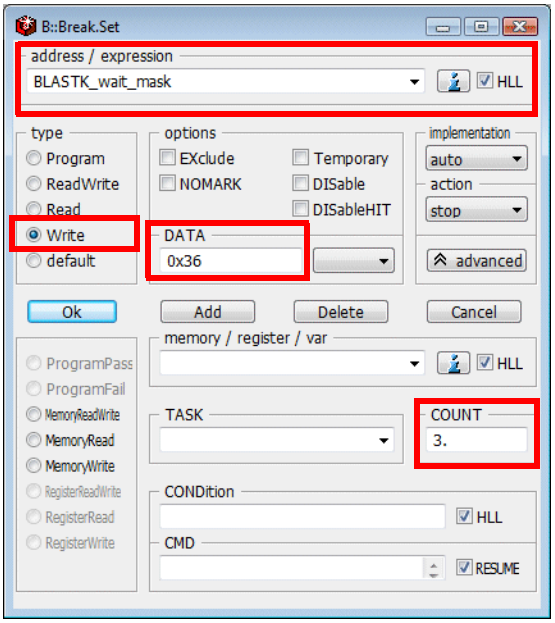
Break.Set *<address>* | *<range>* **/***<access>* **/DATA.auto** *<data>* | **/DATA.Byte** *<data>*
Break.Set *<address>* | *<range>* **/***<access>* **/DATA.Word** *<data>* | **/DATA.Long** *<data>*

Var.Break.Set *<hll_expression>* **/***<access>* **/DATA.auto** *<data>*

Complex data breakpoint: Stop the program execution after a specific data value was read/written from/to a specified address n-times.

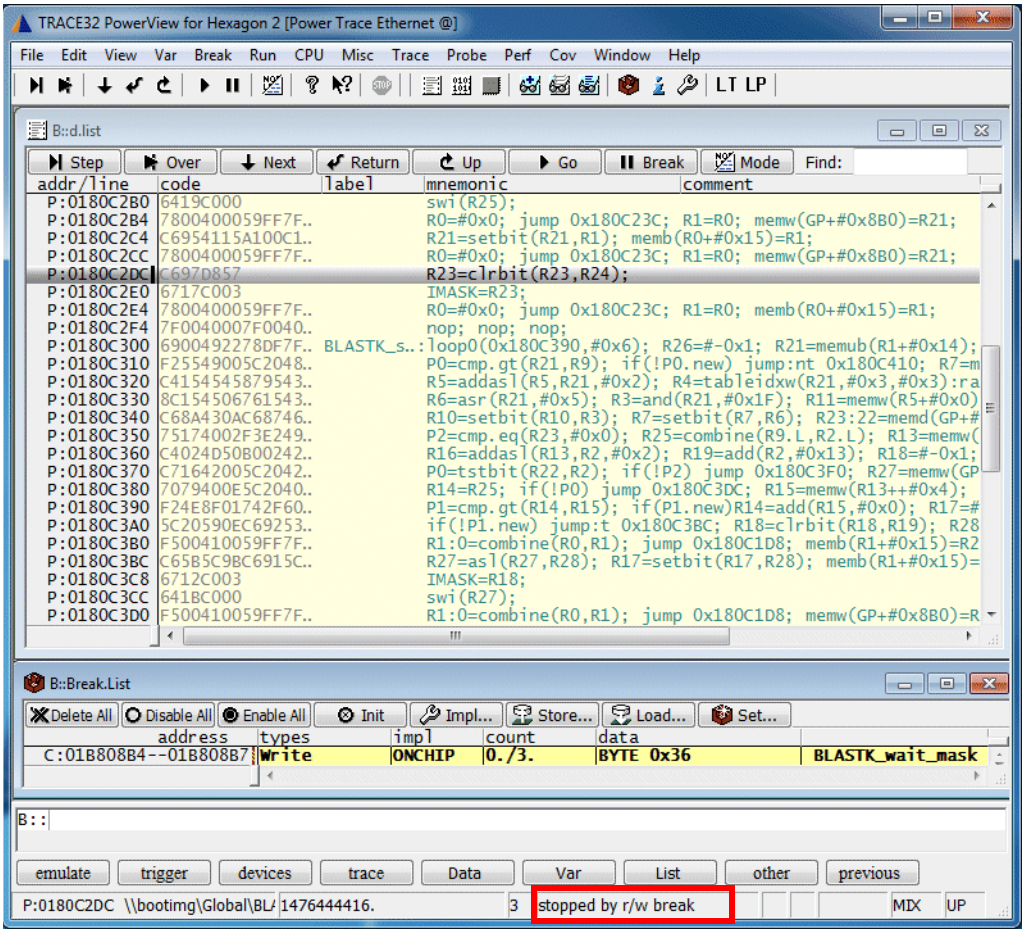
Stop the program execution after the value 0x36 was written 3. times to the integer variable *BLASTK_wait_mask* (etm_break5.cmm).

1. Specify the breakpoint.



   - Specify the variable in the **address / expression** field and enable the **HLL** check box.
   - Specify **Write** as breakpoint type.
   - Specify **DATA** value.
   - Specify the **COUNT**er value.

2. Start the program execution.




```
Var.View %Hex %Decimal BLASTK_wait_mask

Var.Break.Set BLASTK_wait_mask /Write /DATA.auto 0x36 /COUNT 3.

Go
```

Summary

; Set memory access breakpoint, data value possible, one counter
(up to 1)

Break.Set *<address>* | *<range>* **/***<access>* *<data_def>* **/COUNT** *<number>*
Var.Break.Set *<hll_expression>* **/***<access>* *<data_def>* **/COUNT** *<number>*

Complex breakpoint: Stop the program execution after the specified instruction has read/written the specified data value from/to the specified address (negation of the instruction address possible).

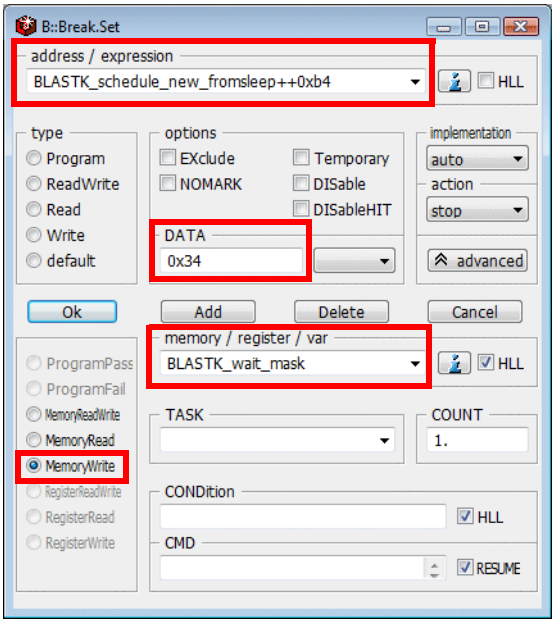
To illustrate the combination of program and data breakpoints, the following examples are provided:

- Example 1: Stop the program execution after an instruction from a <function> has written a <value> to an <integer variable>.
- Example 2: Stop the program execution if any <function>, but not <function X>, writes to the <variable Y>.

Example 1

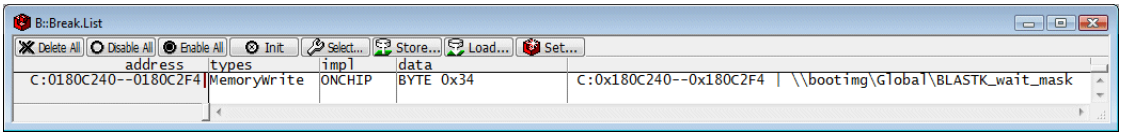
Stop the program execution after an instruction from the function *BLASTK_schedule_new_fromsleep* has written the value 0x34 to the integer variable *BLASTK_wait_mask* (etm_break6.cmm).

1. Specify the breakpoint.

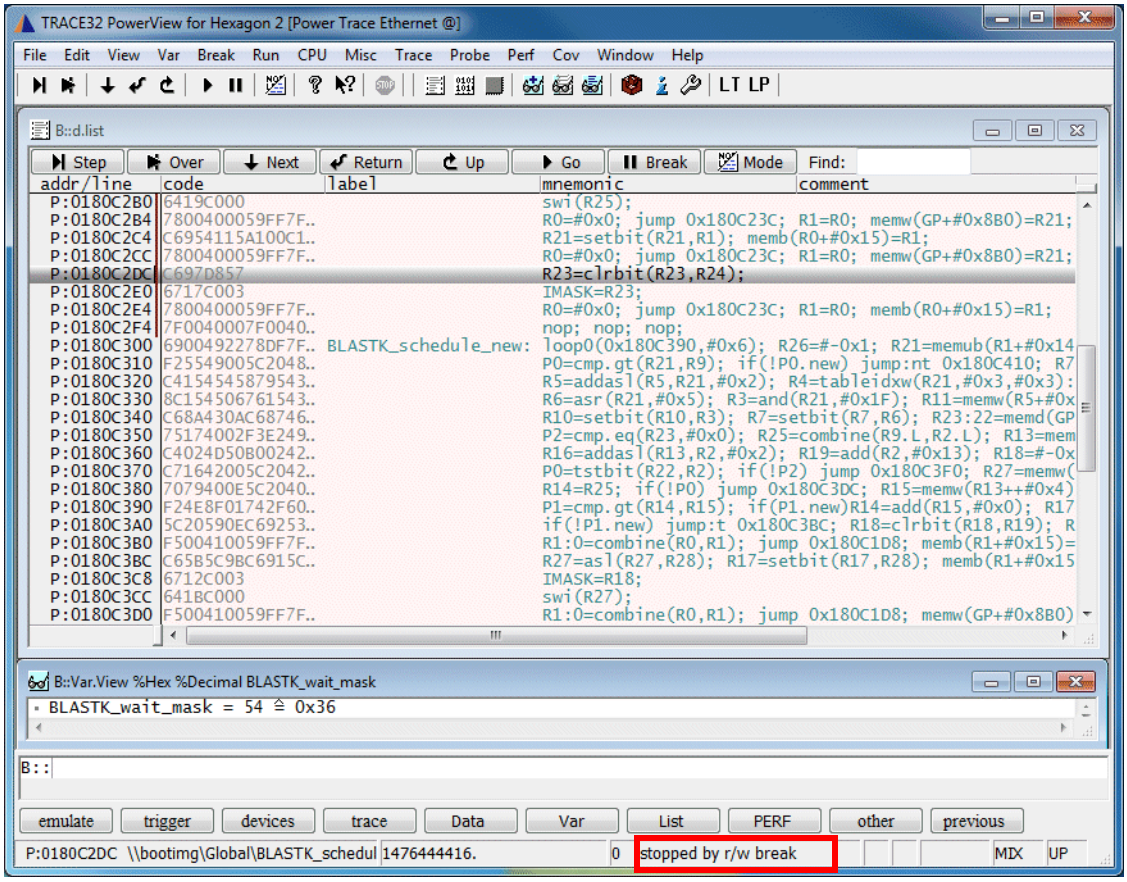


- Specify the function's address range in the **address / expression** field.
- Specify **DATA** value.
- Select **MemoryWrite**.
- Specify the variable in the **memory / register / var** field.

2. List the breakpoint settings.



3. Start the program execution.

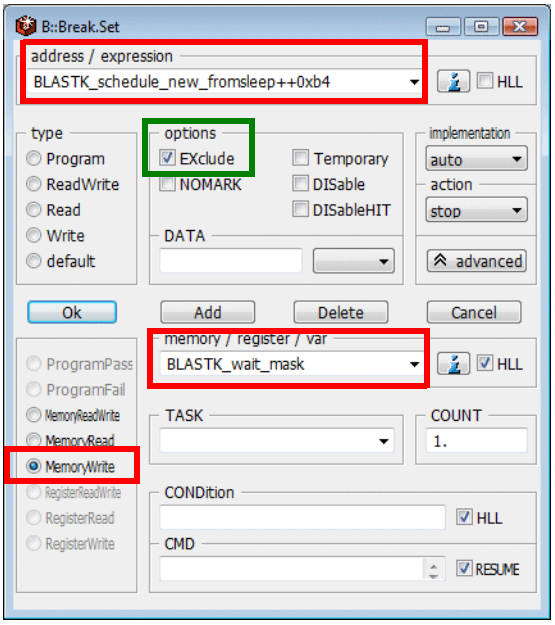



```
Break.Set   0x180C240--0x180C2F4 /VarWrite  BLASTK_wait_mask;
            /DATA.auto 0x34

Go
```

Example 2

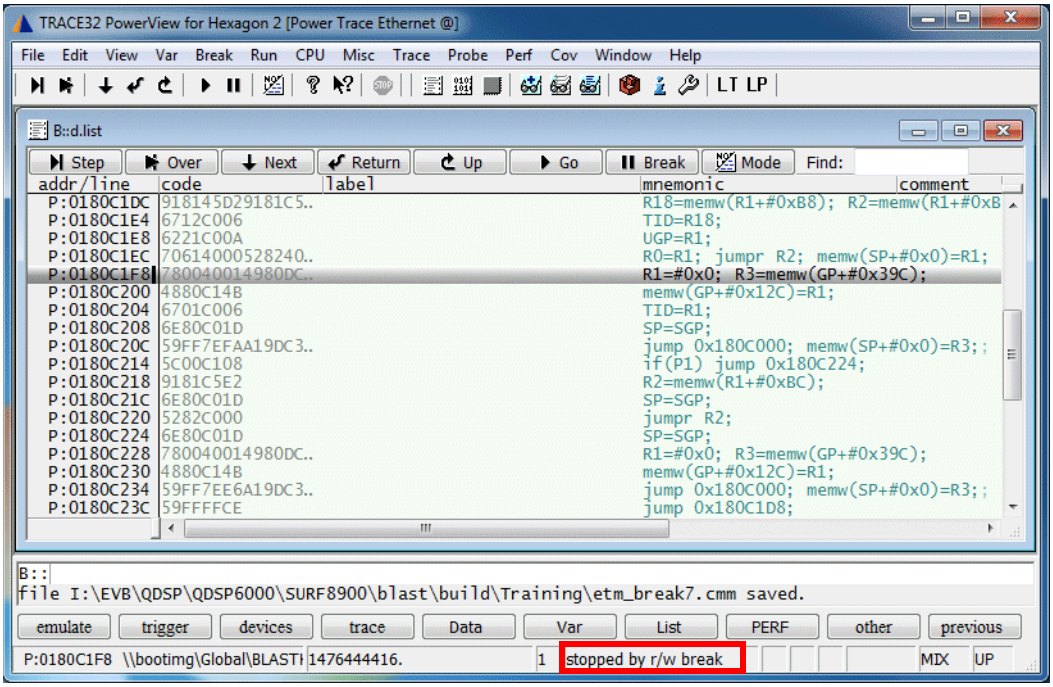
Stop the program execution if any function, but not *BLASTK_schedule_new_fromsleep*, writes to the variable *BLASTK_wait_mask* (etm_break7.cmm).

1. Specify the breakpoint.



   - Specify the function's address range in the **address / expression** field.
   - Select **EXclude** to negate the function's address range.
   - Select **MemoryWrite**.
   - Specify the variable name in the **memory / register / var** field.

2. Start the program execution.




```
Break.Set  BLASTK_schedule_new_fromsleep++0xB4
           /VarWrite BLASTK_wait_mask /EXclude

Go
```

; Set combined instruction/data access breakpoint, data value possible, negation possible
(up to 1)

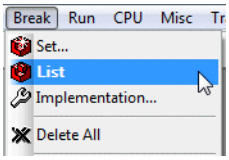
Break.Set *<i_address>* | *<i_range>* **/MemoryReadWrite** *<d_address>* | *<d_range>* *<data_def>* [**/EXclude**]
Break.Set *<i_address>* | *<i_range>* **/MemoryRead** *<d_address>* | *<d_range>* *<data_def>* [**/EXclude**]
Break.Set *<i_address>* | *<i_range>* **/MemoryWrite** *<d_address>* | *<d_range>* *<data_def>* [**/EXclude**]

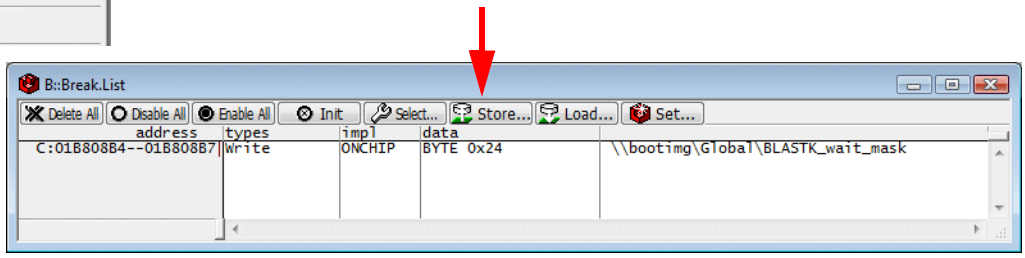
Var.Break.Set *<function>* **/VarReadWrite** *<variable>* **DATA.auto** *<value>* [**/EXclude**]
Var.Break.Set *<function>* **/VarRead** *<variable>* **DATA.auto** *<value>* [**/EXclude**]
Var.Break.Set *<function>* **/VarWrite** *<variable>* **DATA.auto** *<value>* [**/EXclude**]

Saving the Breakpoint Settings as a PRACTICE Script

You can save breakpoint settings via the TRACE32 PowerView GUI or via the TRACE32 command line. To save them via the GUI, take the following steps:

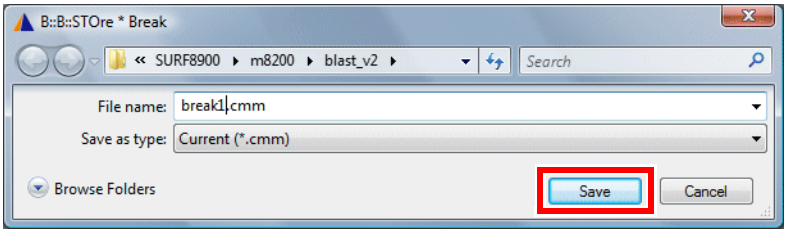
1. **Choose Break menu > List to open a breakpoint listing.**



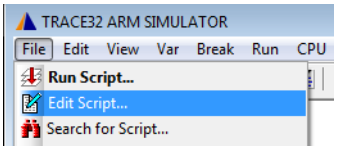


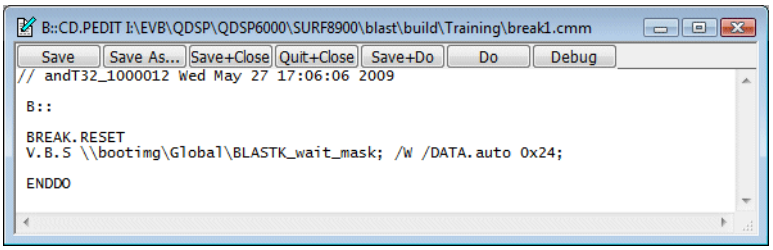
2. **Click the Store button to generate a PRACTICE script for all set breakpoints.**

3. **Specify the name for the PRACTICE script, and then click Save.**



4. **To display the contents of the PRACTICE script, choose File menu > Edit Script.**





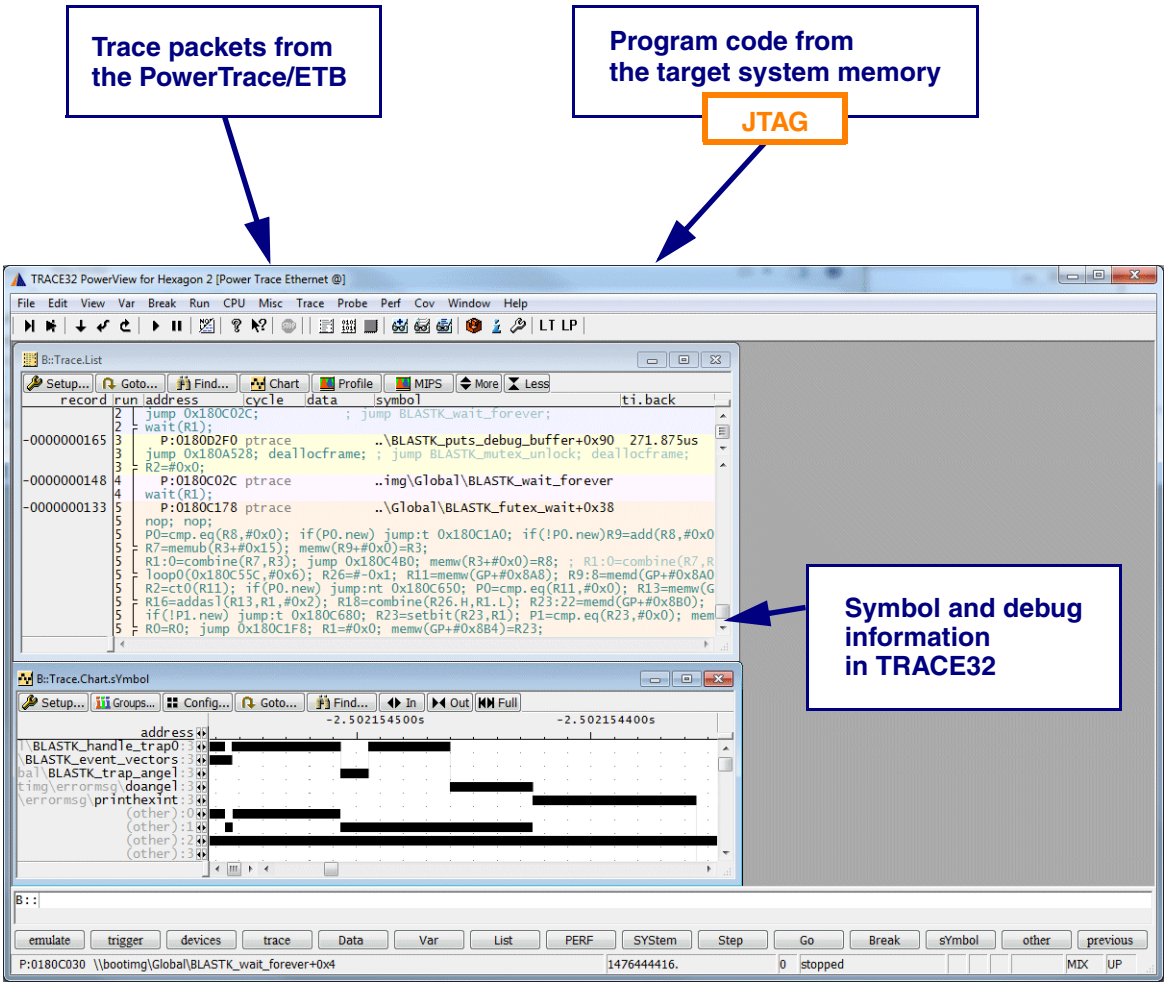
The following commands are available to save breakpoint settings via the TRACE32 command line:

STOre *<file>* **Break**	Save breakpoint settings to file.
ClipSTOre Break	Save breakpoint settings to clipboard.

Fundamentals

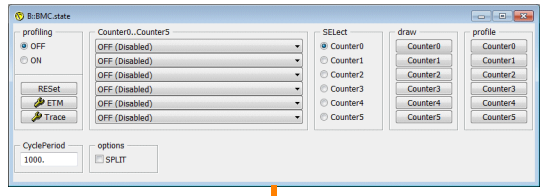
In order to provide an intuitive trace display the following sources of information are merged:

- The **trace packets** stored in the trace memory of the PowerTrace/ETB. The trace packets provide only the addresses of the executed instruction packets (instruction flow).
- The **program** code from the target memory read via JTAG.
- The **symbol and debug information** already loaded to TRACE32 from a file.

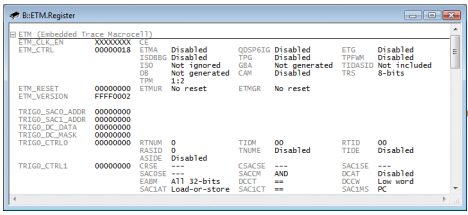


The following functional units have an effect on the trace recording:

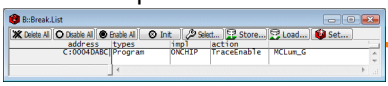
Benchmark counters



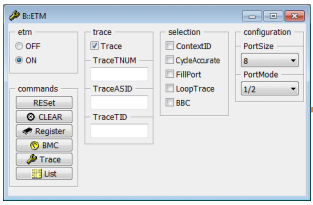
Filter via the ETM.Set command



Filter breakpoints



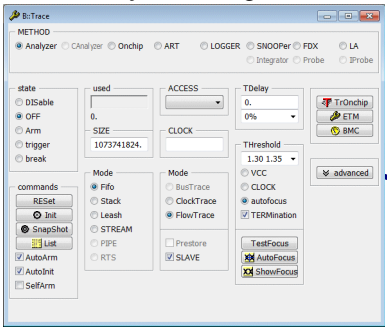
ETM configuration



ETM trace packet generation

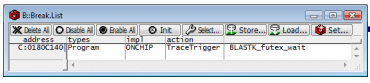
[0..n-1]

Trace/Analyzer configuration in TRACE32

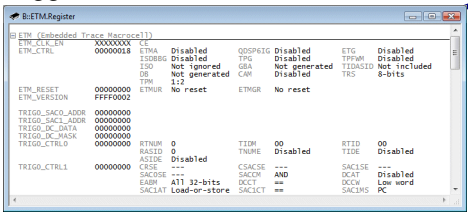


Trace memory of PowerTrace/ETB

Trigger breakpoints



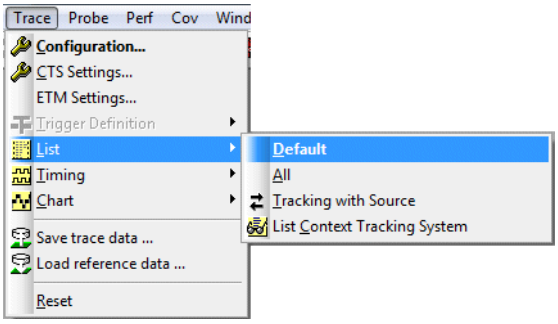
Trigger via the ETM.Set command

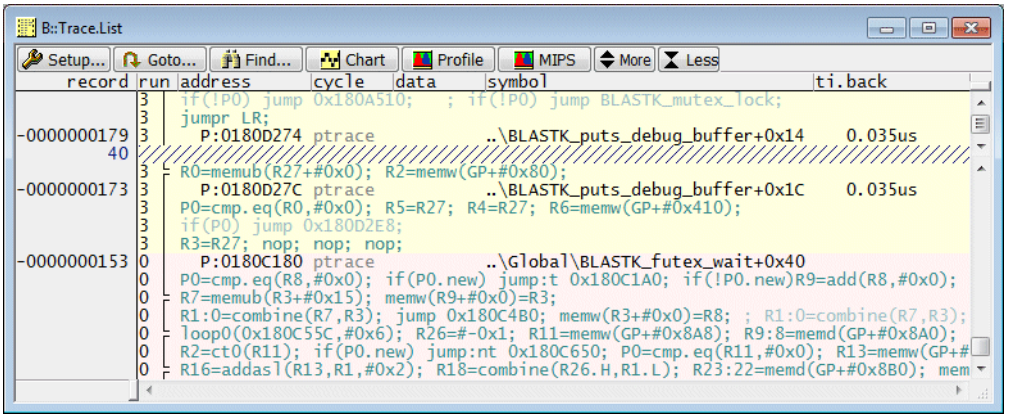


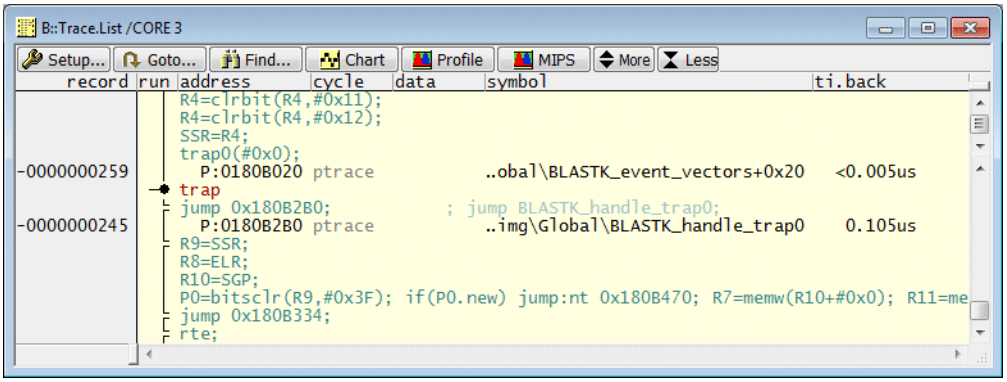
Display Commands

The following commands are available to display a trace listing:

Trace.List	Display a trace listing by merging the trace information of all hardware threads
Trace.List /CORE 0	Display the trace listing based on the trace information generated for hardware thread 0
Trace.List /CORE 1	Display the trace listing based on the trace information generated for hardware thread 1
Trace.List /CORE 2	Display the trace listing based on the trace information generated for hardware thread 2
Trace.List /CORE 3	Display the trace listing based on the trace information generated for hardware thread 3
Trace.List /CORE 4	Display the trace listing based on the trace information generated for hardware thread 4
Trace.List /CORE 5	Display the trace listing based on the trace information generated for hardware thread 5



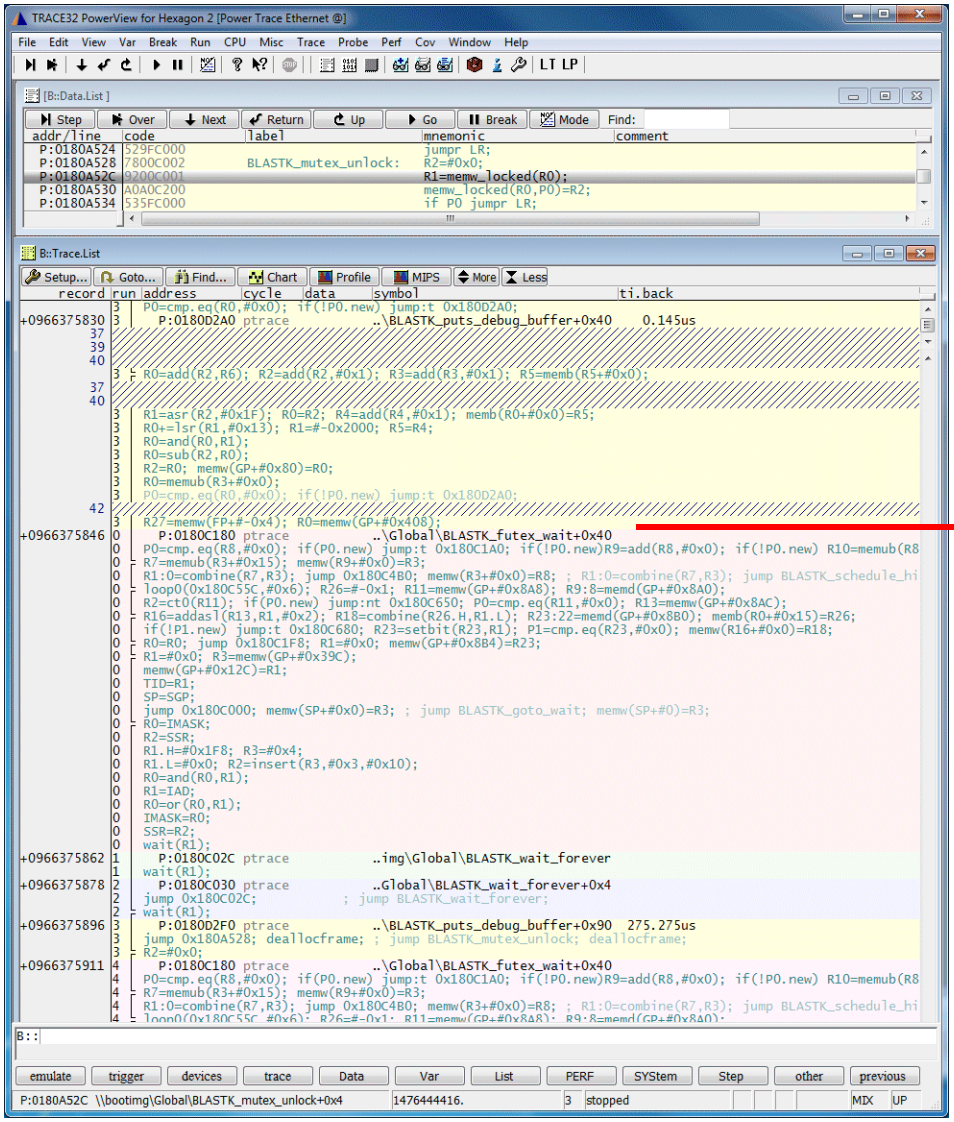





```
Trace.List

Trace.List /CORE 3
```

Please Note

TRACE32 flushes all trace information stuck in the ETM fifos when the recording to the trace repository is stopped because the program execution stopped. These delayed exported trace packets can be identified by no **TIme.Back** value or by a large **TIme.Back** value.



Flushed trace packets

On the one hand, flushing the ETM fifos is necessary to get the correct state of a hardware thread. In most cases wait instructions are stuck.

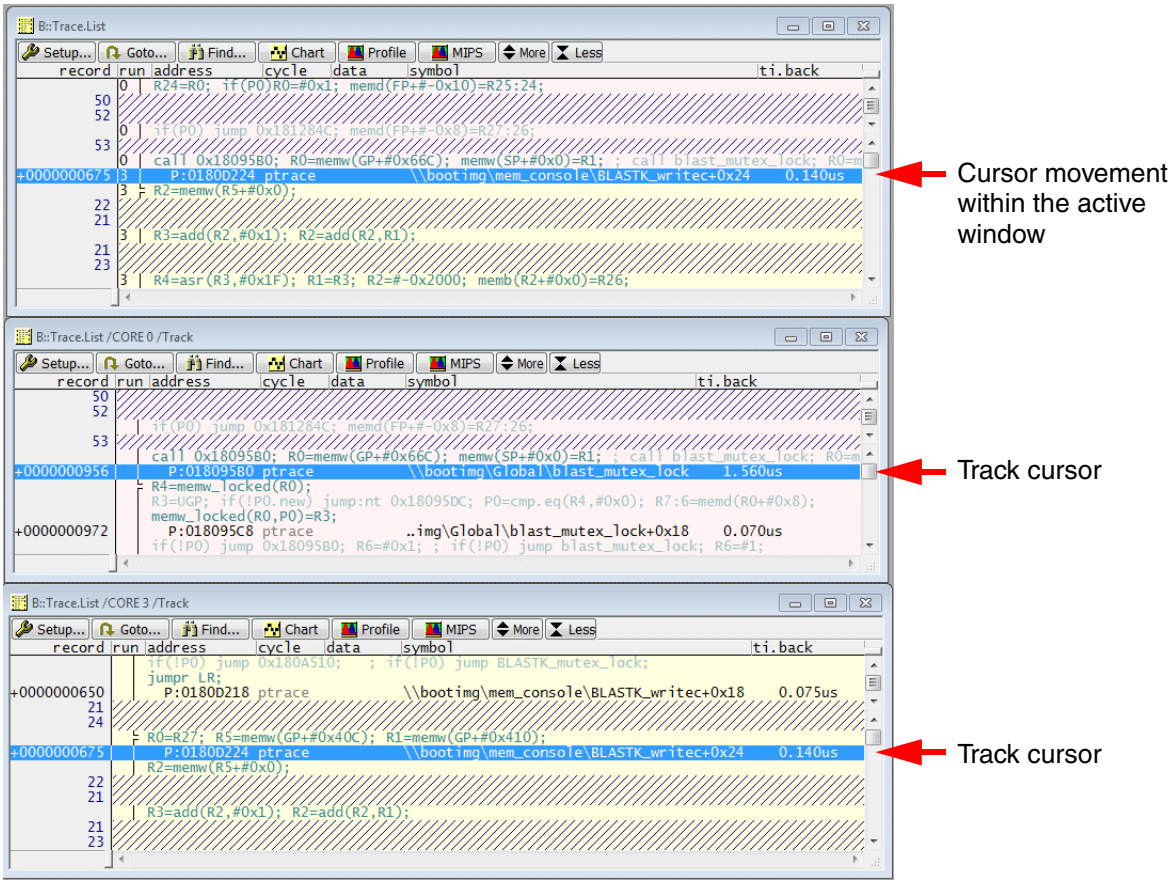
On the other hand, run-time measurements can be falsified due to incorrect (too large) timestamps. Please refer to **"Did you know?"** to learn how to exclude flushed trace packets from the run-time measurement.

Correlating Different Trace Displays

The **/Track** option allows to establish a timing relation between different trace displays. The cursors of all **Trace.List** windows with the option **/Track** track the cursor movement within the active window.

Example:

```
Trace.List

Trace.List /CORE 0 /Track

Trace.List /CORE 3 /Track
```



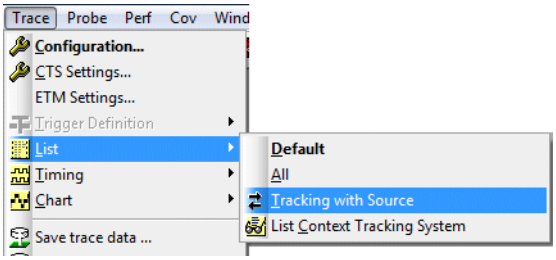
If a trace record in the **Trace.List** window is selected, the cursors in the **Trace.List /CORE 0** and **Trace.List /CORE 3** windows mark the record that was executed by their hardware thread nearly at the same time.

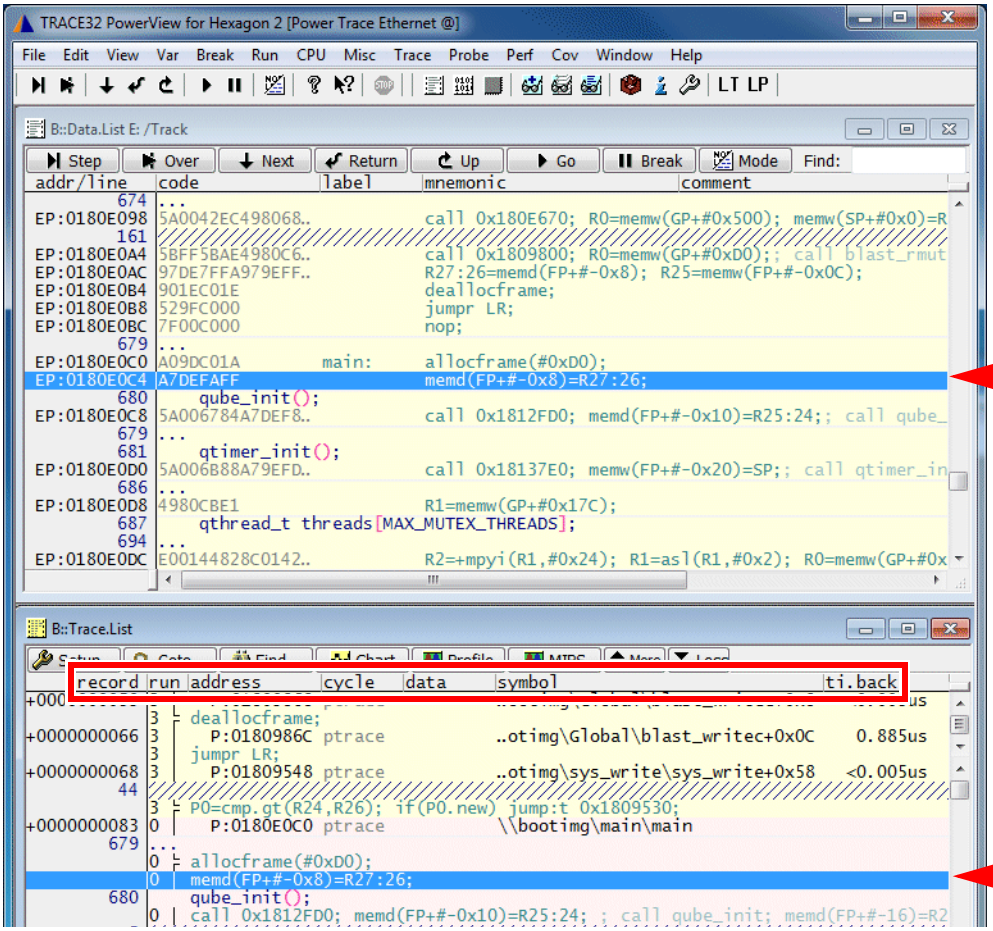
Correlating the Trace Display and the Source Code

The **/Track** option also allows to establish a logical relation between a trace listing and a source code listing. If a trace record is selected in the **Trace.List** window, the corresponding source code line is automatically highlighted with a blue cursor.

Example:

```
Trace.List
List /Track
```





For a description of the highlighted columns, see "**Default Display Items**".

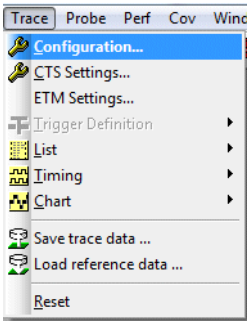
Columns	Description
record	Record number (For details, click **here**.)
run	Run-time information (For details, click **here**.)
address	Logical address of the executed instruction packet.
cycle	Cycle type. The only available cycles type is **ptrace**. **ptrace** stands for program trace information.
data	(No data access information is exported by the Hexagon ETM)
symbol	Symbolic address of the executed instruction packet
ti.back (TIme.Back)	Distance of time between a trace record and its preceding trace record (For details, click **here**.)

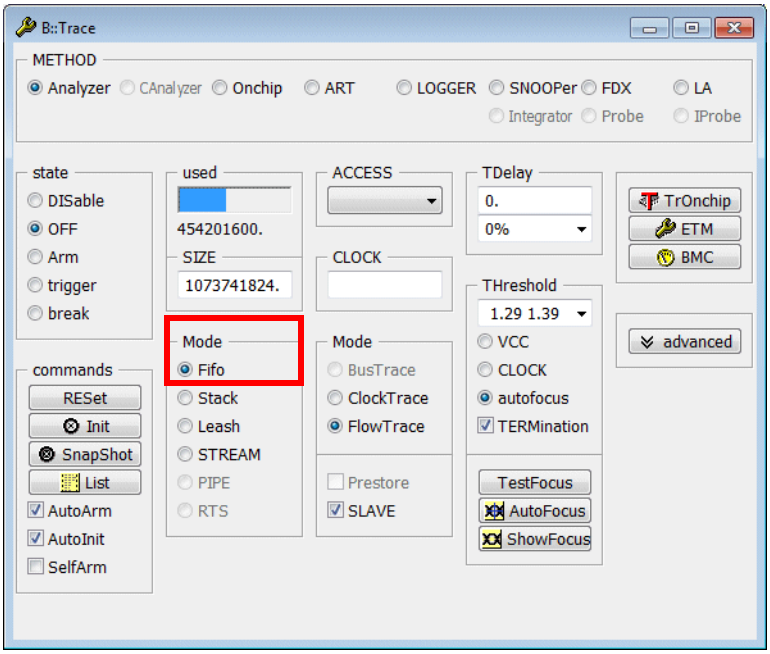
record

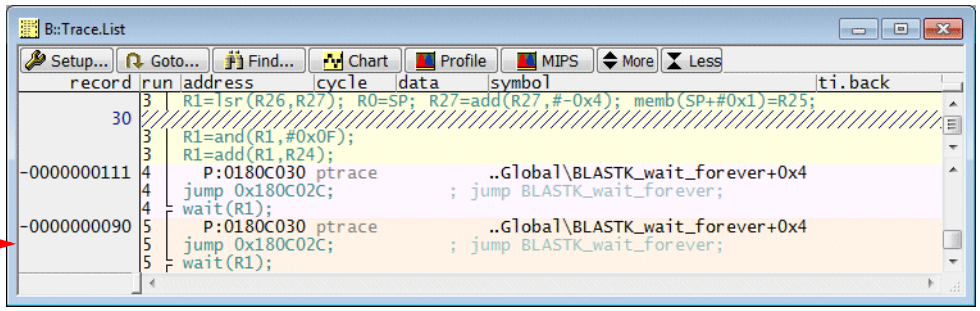
Trace records are numbered consecutively in the trace display. The numbering scheme depends on the selected trace mode. The following trace modes are available:

- **Fifo Mode**
- **Stack Mode**
- **Leash Mode**
- **STREAM Mode**

```
Trace.Mode Fifo                          ; Default mode

                                         ; When the trace repository is full
                                         ; the newest trace information
                                         ; overwrites the oldest

                                         ; The trace repository contains
                                         ; all information exported
                                         ; until the program execution
                                         ; stopped
```

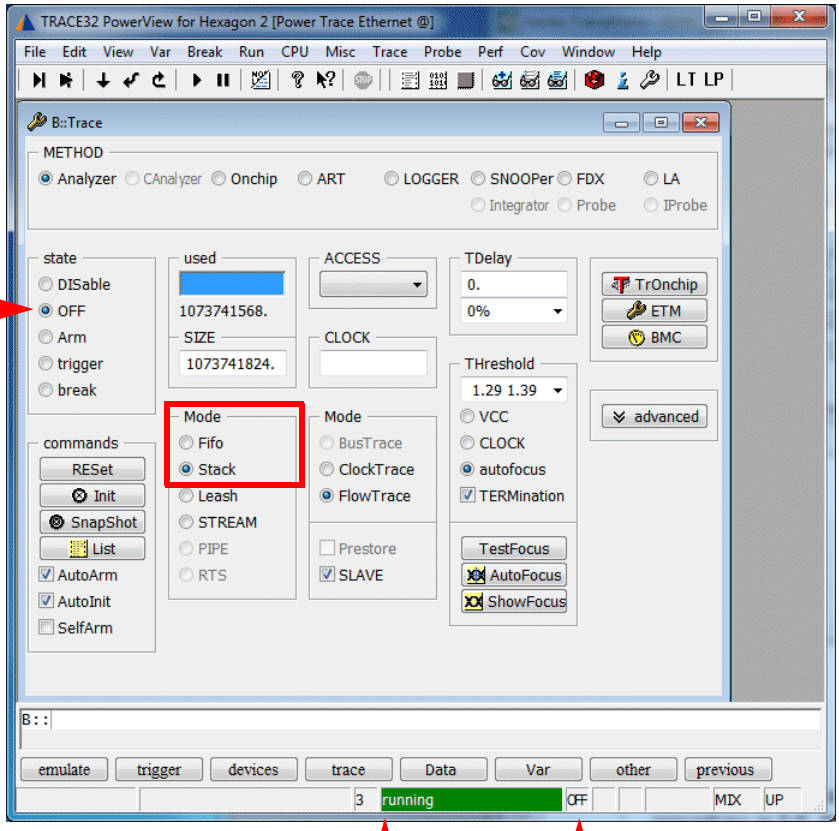






In **Fifo** mode negative record numbers are used.
The last record gets the smallest negative number.

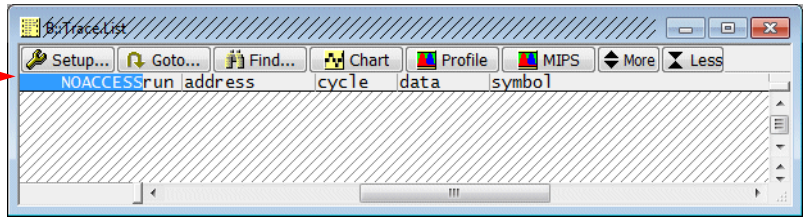
```
Trace.Mode Stack                        ; When the trace repository is full
                                        ; the trace recording is stopped

                                        ; The trace repository contains
                                        ; all information exported
                                        ; directly after the start of
                                        ; the program execution
```



As soon as the trace repository is full, the trace capturing is stopped (OFF state)

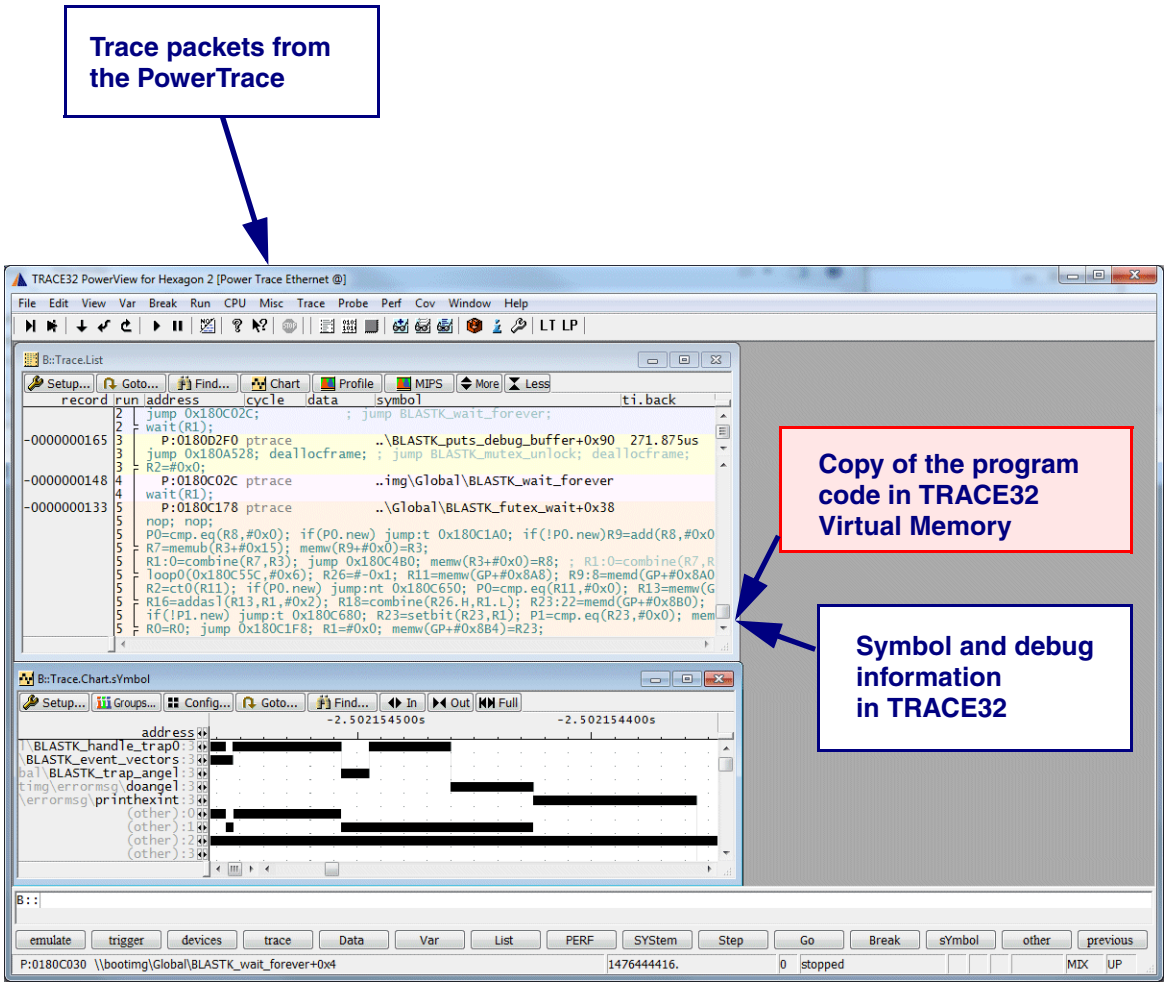
OFF in the Trace State Field indicates that the trace capturing is stopped

running in the Debug State Field indicates that the program execution is running



Trace information can not be displayed while the program is running, since TRACE32 has **NOACCESS** to the program code in the target system memory

In order to display the trace information, you can either **stop the program execution**, or you can set up TRACE32 for displaying the trace information while the program execution is running. This is done by copying the program code to the TRACE32 Virtual Memory (VM:).




```
; Copy the program code from the target system memory into the TRACE32
; Virtual Memory (VM:) in order to get access to the program code
; while the program execution is running
Data.COPY 0x1800000--0x182afff VM:
```

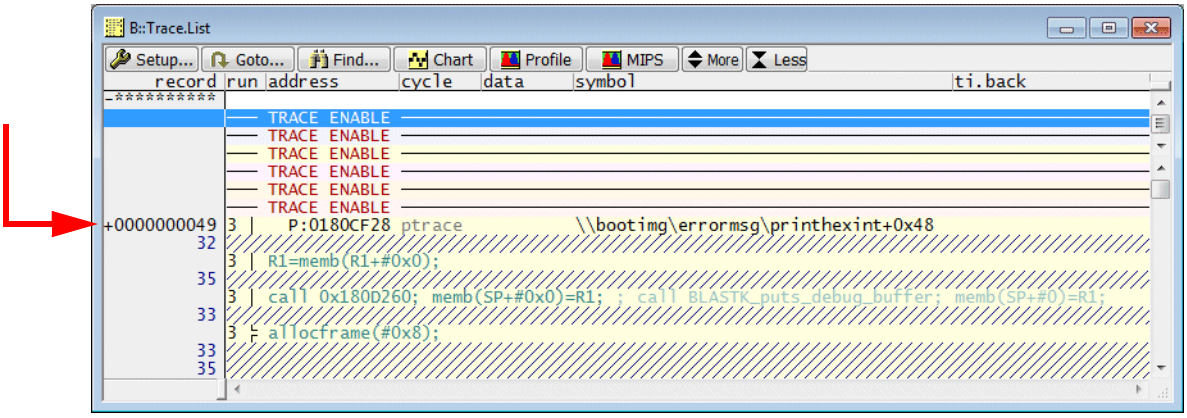
Alternatively:

```
; Load the program code into the TRACE32 Virtual Memory (VM:)
Data.LOAD.Elf blast/bootimg.pbn /VM /NOREG /NOMAP
```

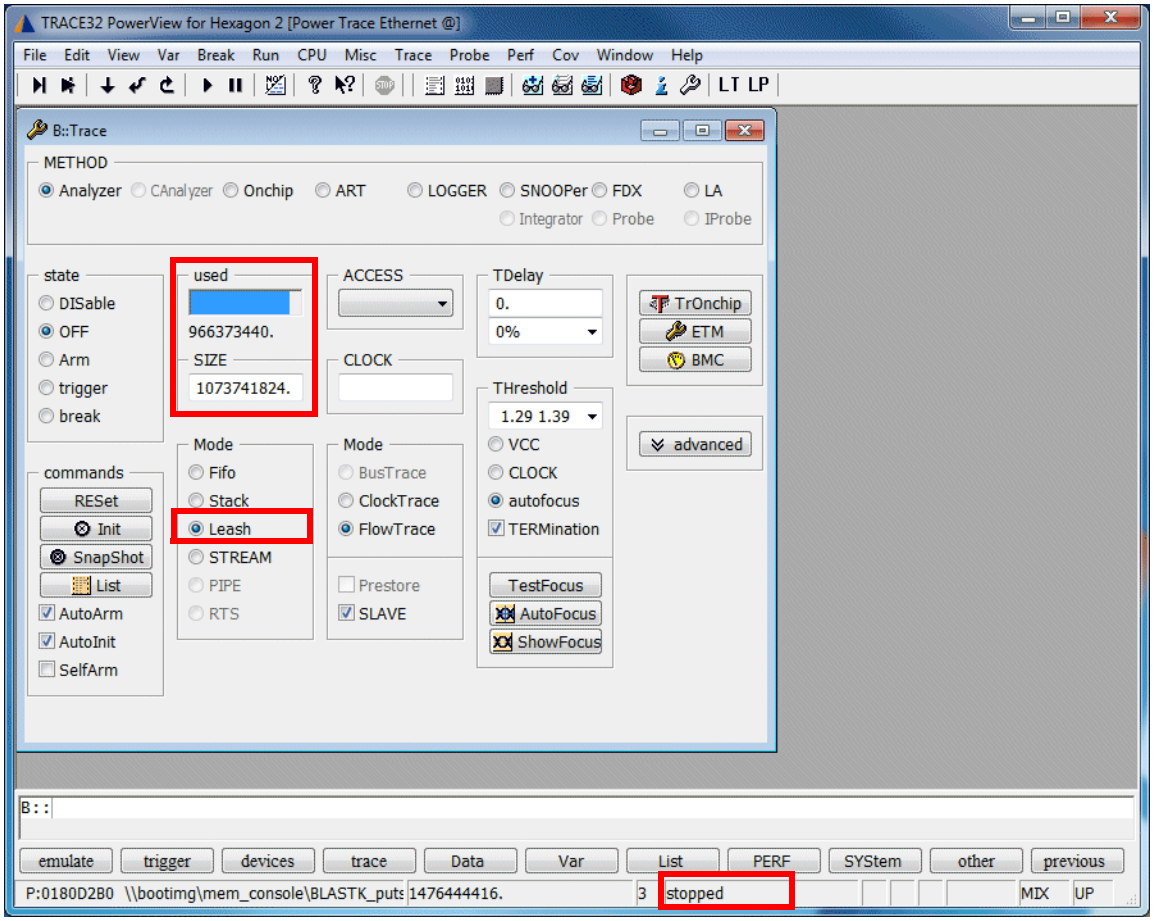
Loading the program code into the virtual memory is also recommended if the JTAG interface is very slow or if there is no access to the target system memory due to any reasons.

NOTE:	Please make sure that the TRACE32 Virtual Memory always provides an up-to-date version of the program code. Out-of-date program versions will cause FLOW ERRORs (see **“FLOW ERROR”** (training_hexagon_etm.pdf) on **page 29**.

Back to **Stack** mode now: Since the trace recording starts with the program execution and stops when the trace repository is full, positive record numbers are used in **Stack** mode. The first record in the trace gets the smallest positive number.



```
Trace.Mode Leash                        ; When the trace repository is
                                        ; nearly full the program execution
                                        ; is stopped

                                        ; Same record numbering as for
                                        ; Stack mode
```



```
Trace.Mode STREAM                       ; STREAM the recorded trace
                                        ; information to a file on the host
                                        ; computer

                                        ; STREAM mode uses the same record
                                        ; numbering scheme as Stack mode
```

The trace information is immediately streamed to a file on the host computer after it was placed into the trace memory of TRACE32 PowerTrace. This procedure extends the size of the trace memory to up to 1 T Frames.

Streaming mode requires 64-bit host computer and a 64-bit TRACE32 executable to handle the large trace record numbers.

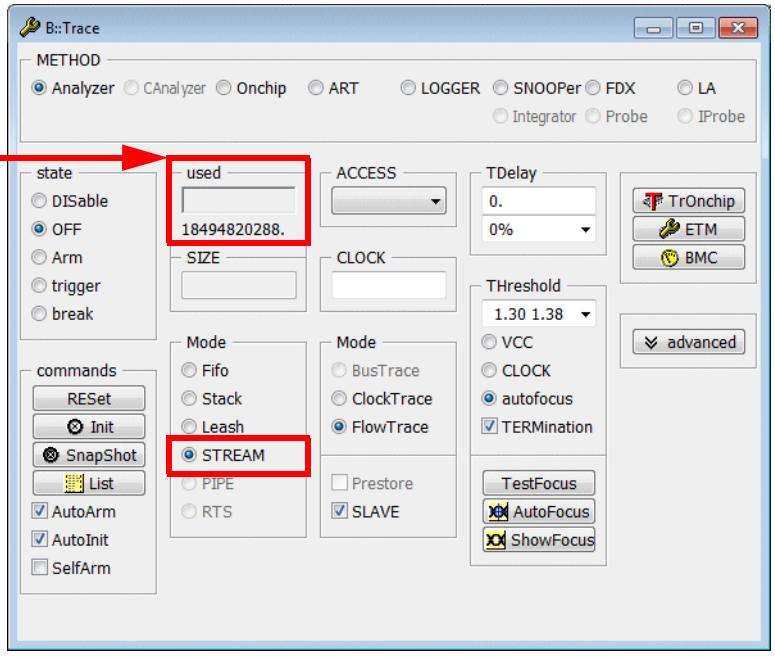
By default the streaming file is placed into the TRACE32 temporary directory (**OS.PresentTemporaryDirectory()**).

The command **Trace.STREAMFILE** *<file>* allows to specify a different name and location for the streaming file.

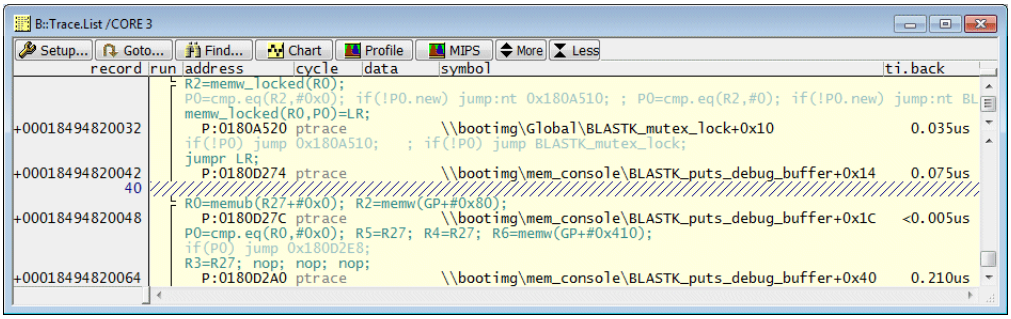
```
Trace.STREAMFILE d:\temp\mystream.t32      ; Specify the location for
                                           ; your streaming file
```

Please be aware that the streaming file is deleted as soon as you de-select the STREAM mode or when you exit TRACE32.

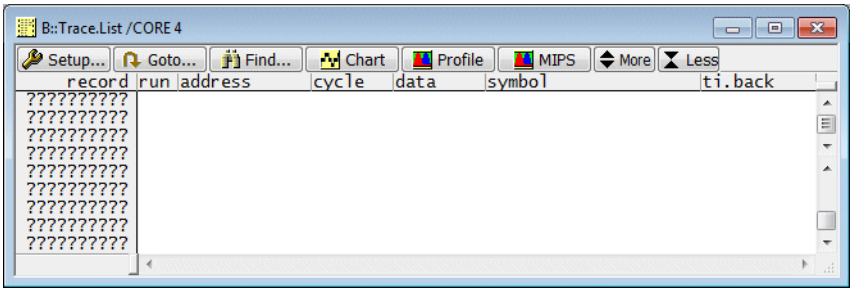
STREAM mode can only be used if the average data rate at the trace port does not exceed the maximum transmission rate of the host interface in use. Peak loads at the trace port are intercepted by the trace memory of the PowerTrace, which can be considered to be operating as a large FIFO.



used indicates how much trace information is buffered by the trace memory (used FIFO)



STREAM mode can generate very large record numbers



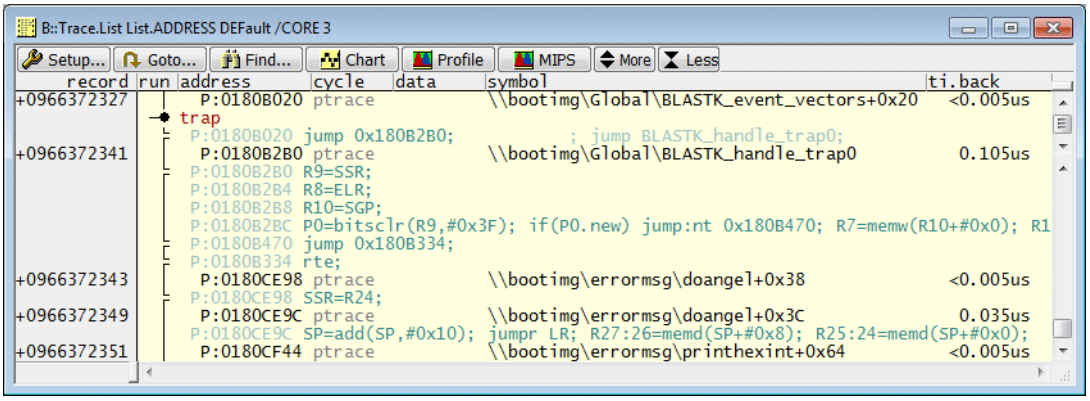
If no trace information was exported by a hardware thread within 50.000 records, the record column shows ????.

Graphic elements provide a quick overview on the program flow

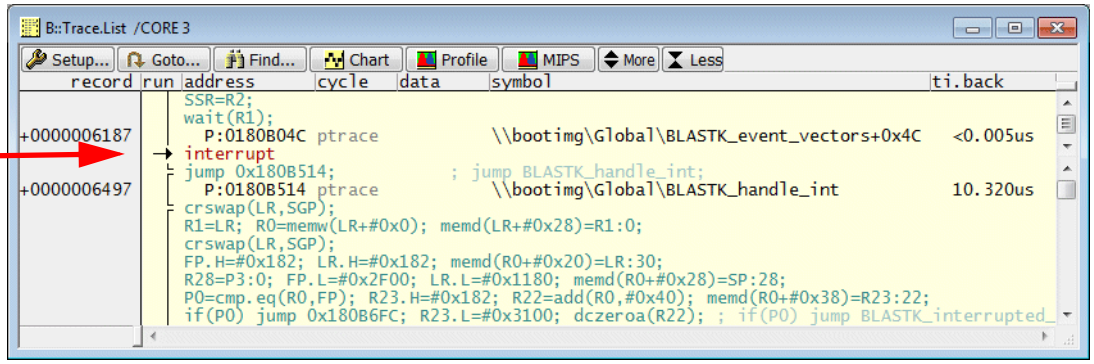
sequential instruction execution

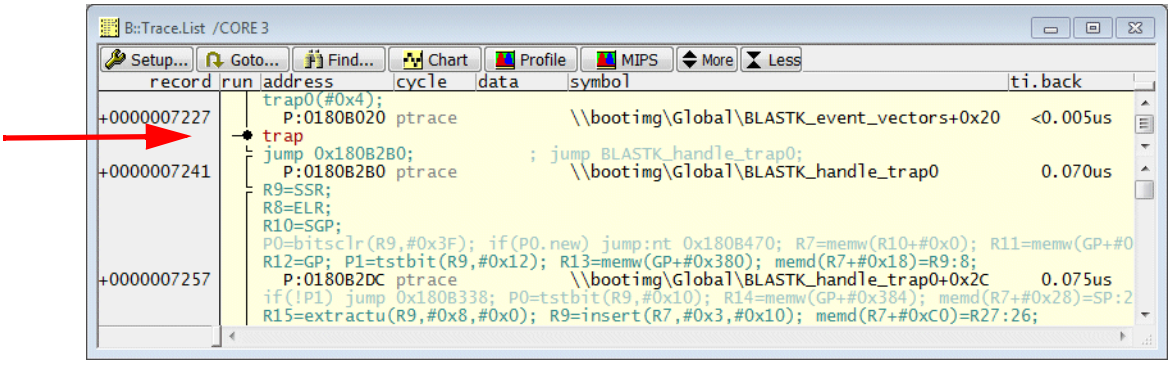
branch taken

```
; Display trace information for hardware thread 3
; (List.ADDRESS) display address information for all instruction packets
Trace.List List.ADDRESS DEFault /CORE 3
```

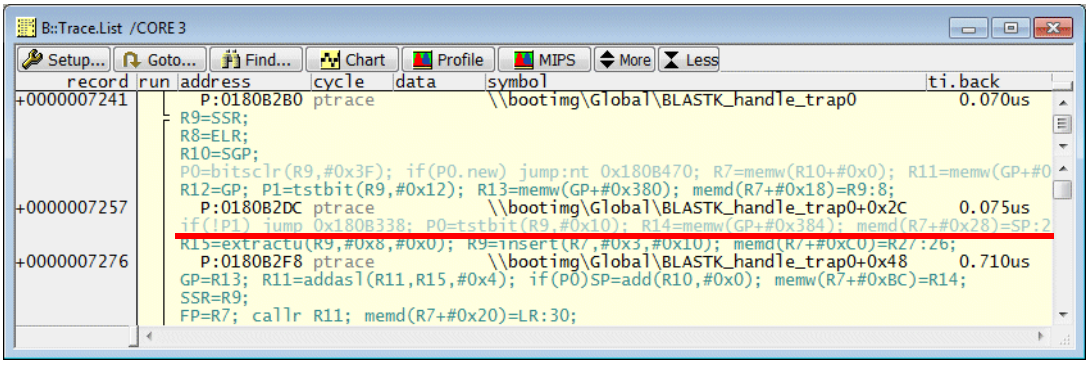


Interrupts/Traps are indicated in the **run** column.

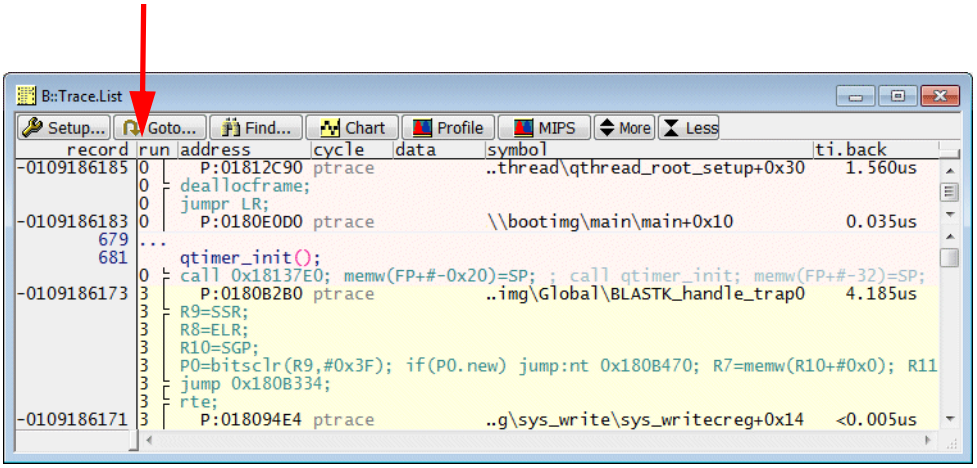




Pastel printed source code indicates that a branch was **not taken**.



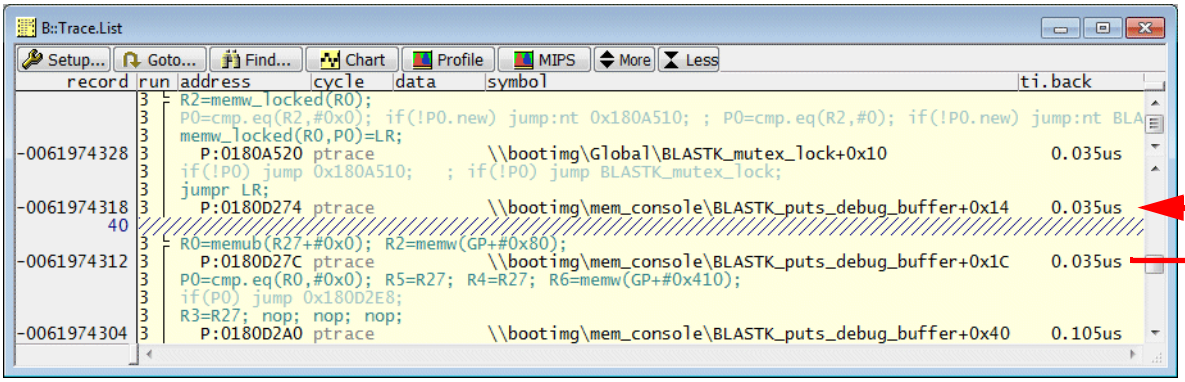
```
Trace.List                                  ; The run column indicates which
                                            ; hardware thread executed the
                                            ; exported instruction packet
```

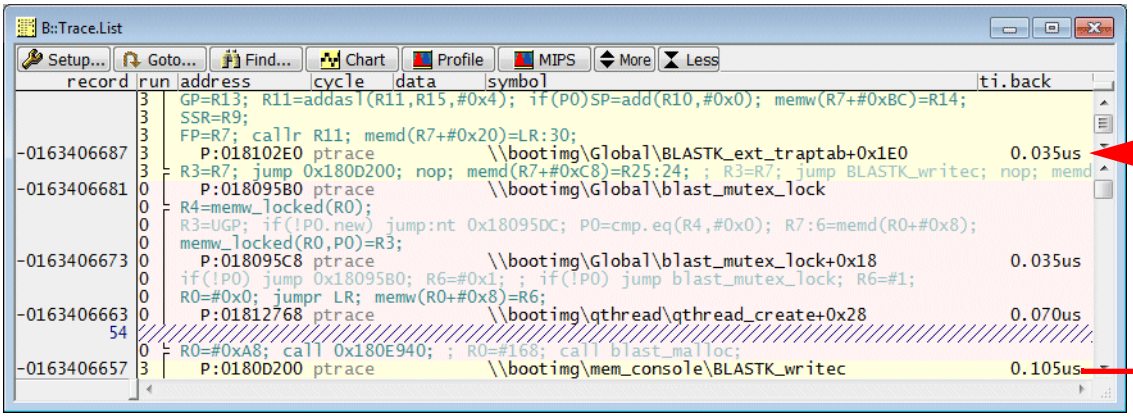


address/symbol

The address column shows the logical address of the executed instruction packet.
The symbol column shows the symbolic address of the executed instruction packet.

TIme.Back





TIme.Back indicates the distance of time between a trace record and its preceding trace record on the same core.

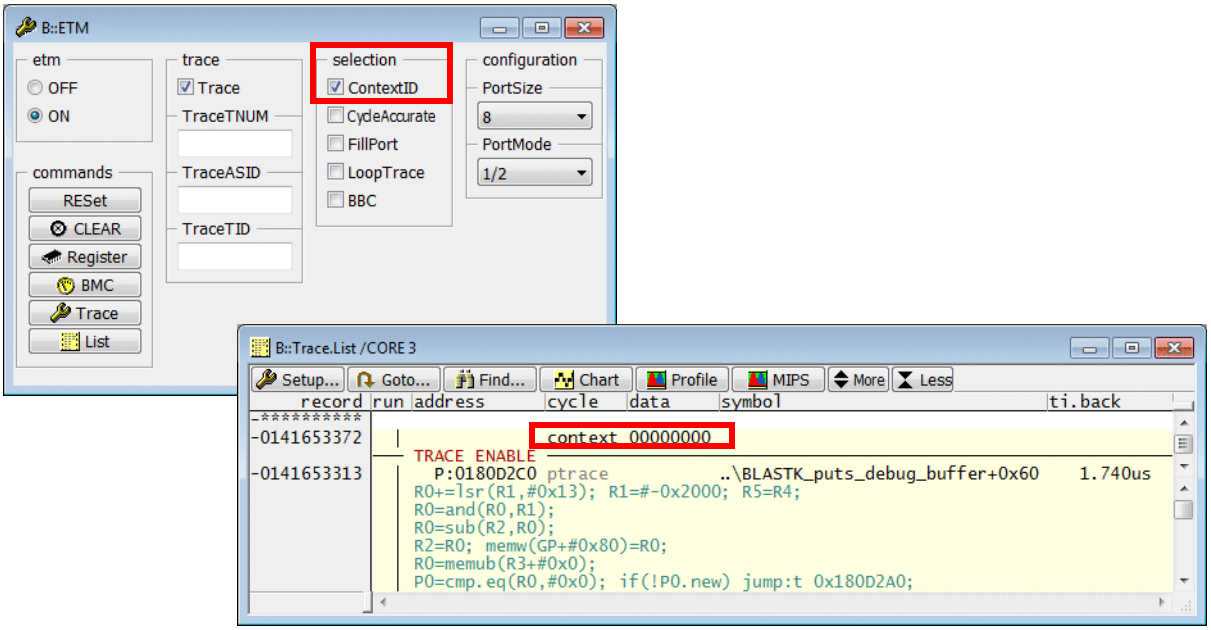
No **TIme.Back** information is displayed, if the preceding trace record on the same core is too far away.

Timestamp generation

- (**ETM.CycleAccurate OFF**): Trace records are time stamped when they are stored into the PowerTrace's memory. The resolution of the timestamp is 10 ns for PowerTrace and 5 ns for PowerTrace II / PowerTrace III.
- (**ETM.CycleAccurate ON**): The time information is calculated from the exported trace information and the core clock provided by the command **Trace.CLOCK** *<core_clock>*.

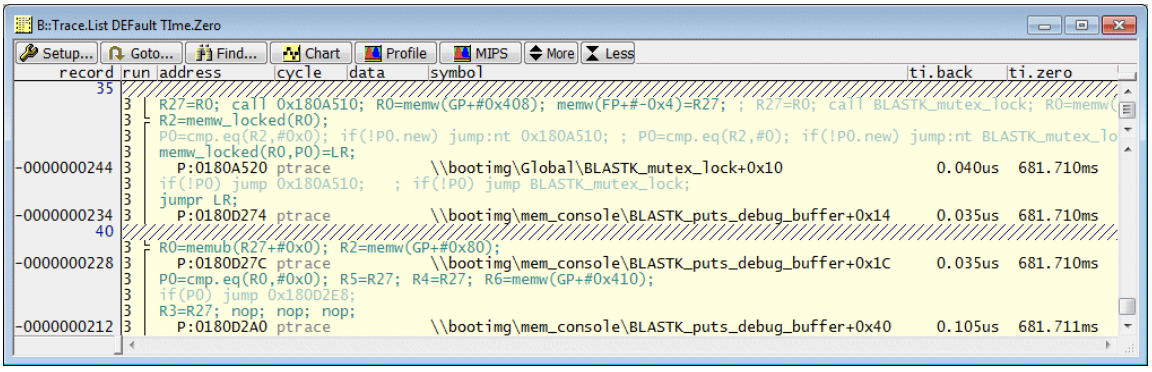
ASID and TID

If the **ContextID** check box is active in the **ETM.state** window, the ASID and TID are exported by the ETM.

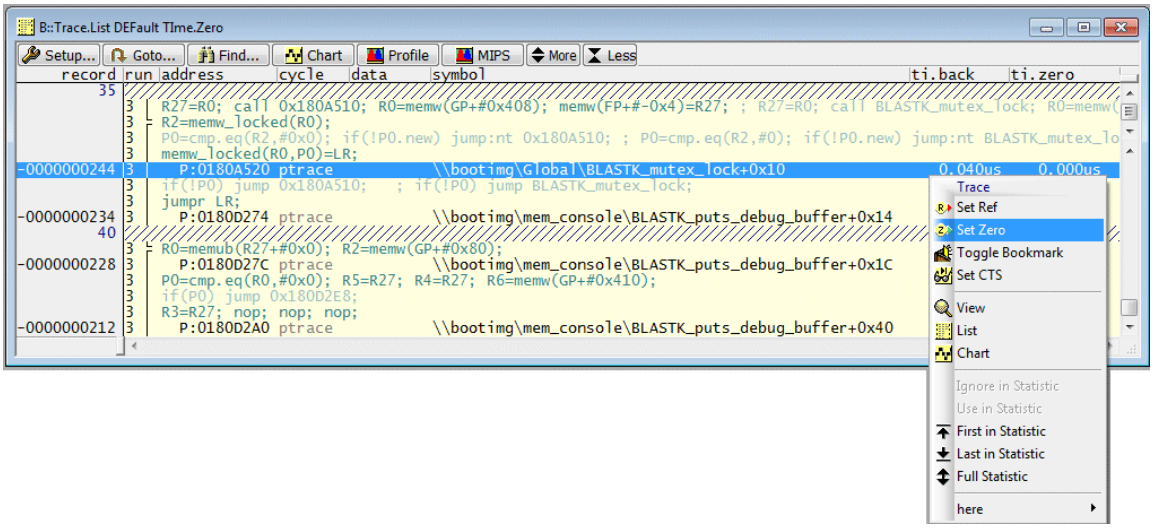


In addition to **TIme.Back** there is also a more global time information called **TIme.Zero**.

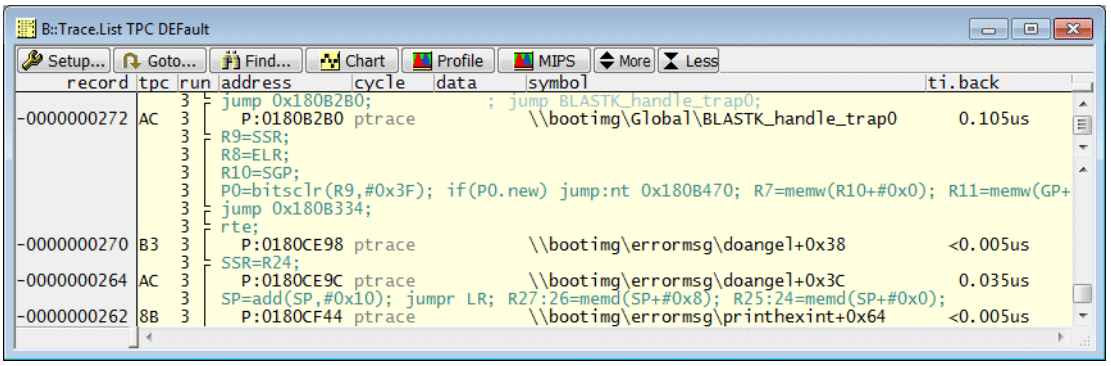
```
Trace.List DEFault TIme.Zero                    ; Add the TIme.Zero
                                                ; information to
                                                ; the default trace display
```



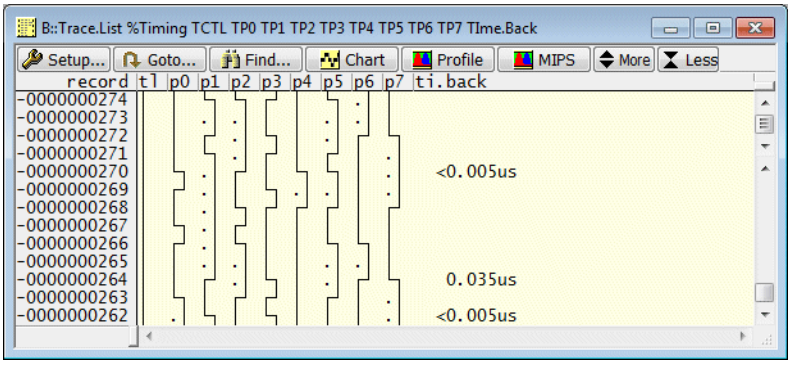
TRACE32 allows to mark a selected record as zero point within the trace. All other trace records are then time referenced to this record.



```
Trace.List TP DEFault /CORE 0          ; Add the trace packet information
                                       ; to the default trace display
```

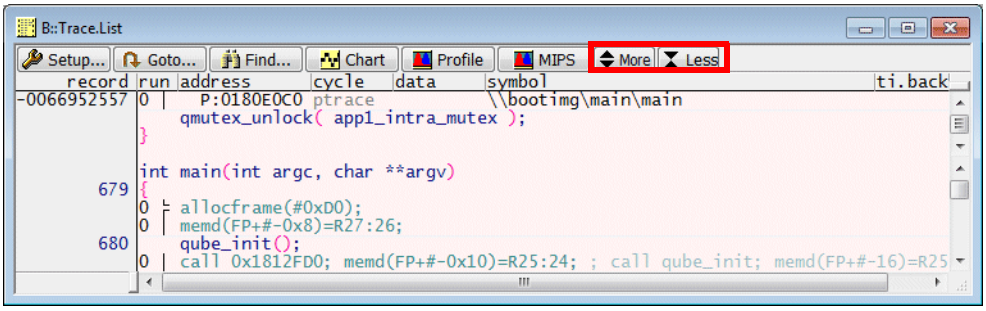



```
; Display trace control and the lowest 8 trace port pins with timestamp
Trace.List %Timing TCTL TP0 TP1 TP2 TP3 TP4 TP5 TP6 TP7 TIme.Back
```



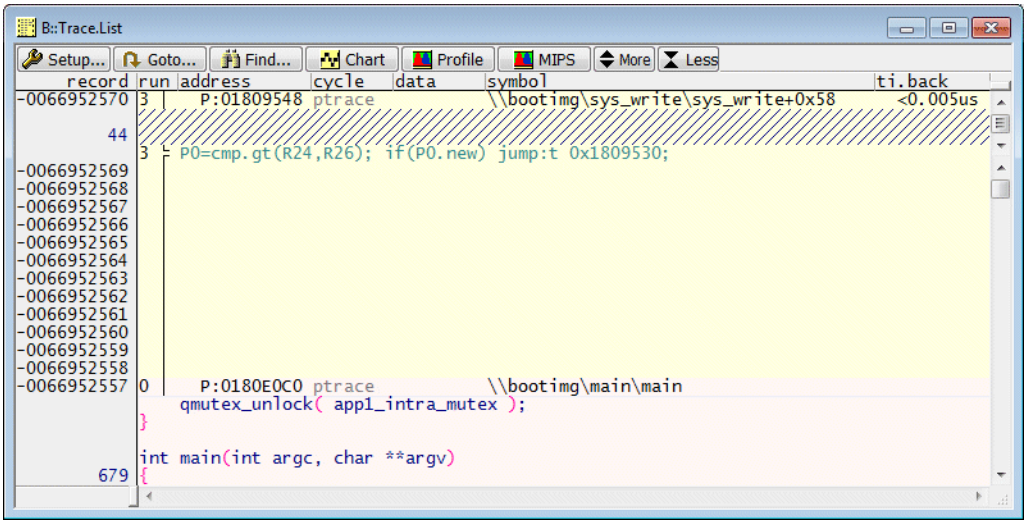
Formatting the Trace Display

The standard way to format the trace display is to use the **More/Less** buttons.



Pushing one time the More button

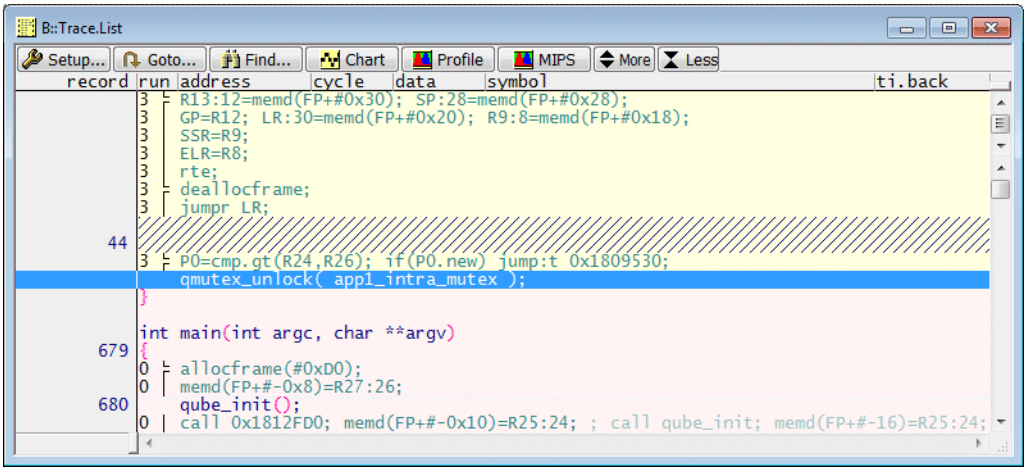
Pushing one time the **More** button will add the so-called dummy records to the trace display. Dummy records don't provide information with regards to the program execution. They are just empty in most cases.




```
Trace.List DEFault List.NoDummy.OFF
```

Pushing for the first time the Less button

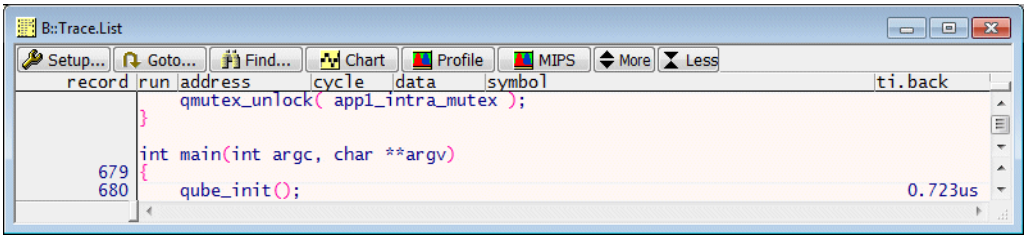
Pushing for the first time the **Less** button will remove the trace packet information (ptrace records) from the trace display.




```
Trace.List DEFault List.NoCycle
```

Pushing for the second time the Less button

Pushing for the second time the **Less** button will remove the assembly code from the trace display.



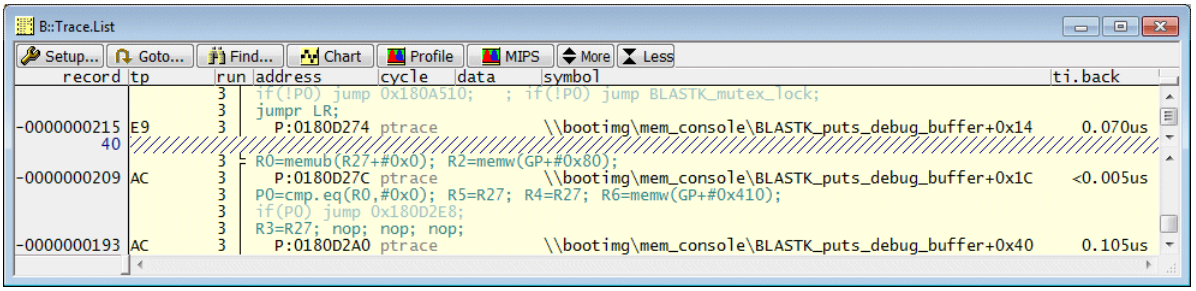

```
Trace.List List.HllOnly List.TIme TIme.Back
```

Changing the DEFault Display

The command **SETUP.ALIST** allows to change the DEFault display of the trace information preset by TRACE32.

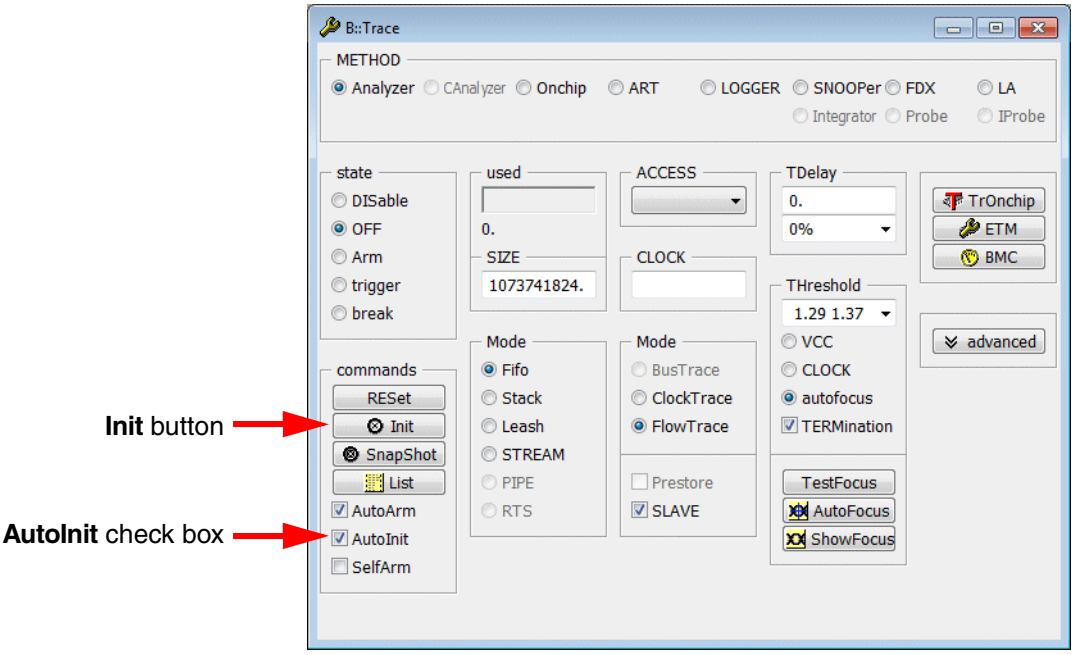
Examples:

```
; Add the column TIme.Zero after the default display
SETUP.ALIST DEFault TIme.Zero

; Add time and address information for every instruction packet
SETUP.ALIST DEFault List.ADDRESS List.TIme

; Add ETM trace packet information before the default display
; See picture below
SETUP.ALIST TP DEFault

; Increase the width of the symbol column (60 characters)
SETUP.ALIST %LEN 60 DEFault
```



The AutoInit Option

While testing it might be helpful to clear the trace memory of the PowerTrace/ETB before a new test is started. Instead of pushing manually the **Init** button in the **Trace.state** window, it is more convenient to activate the **AutoInit** check box.

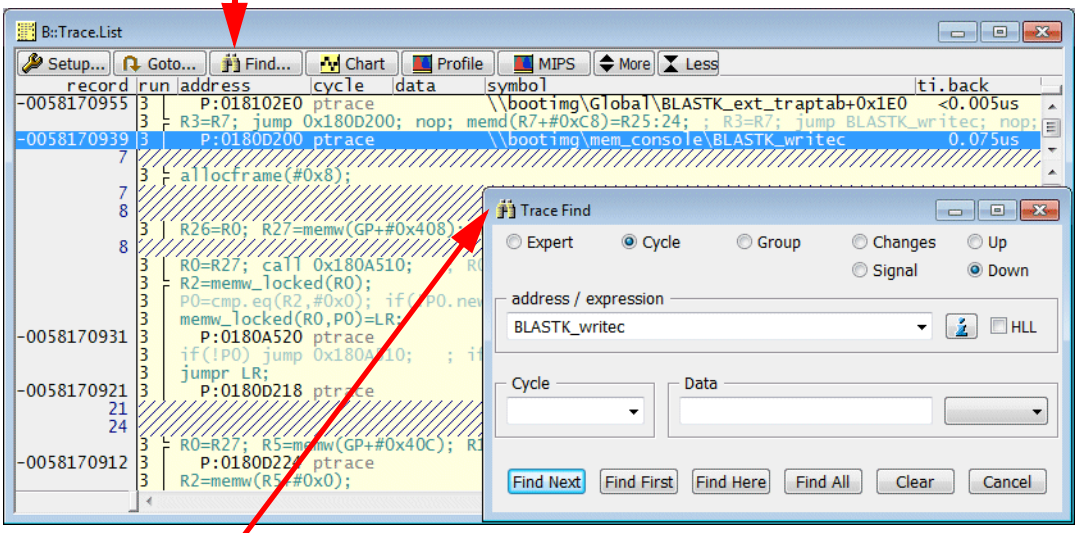



```
Trace.AutoInit ON                       ; The trace memory is
                                        ; automatically cleared before
                                        ; the program execution is started
```

Searching in the Trace

TRACE32 provides fast search algorithms to find a specific event in the trace quickly.

Push the **Find...** button

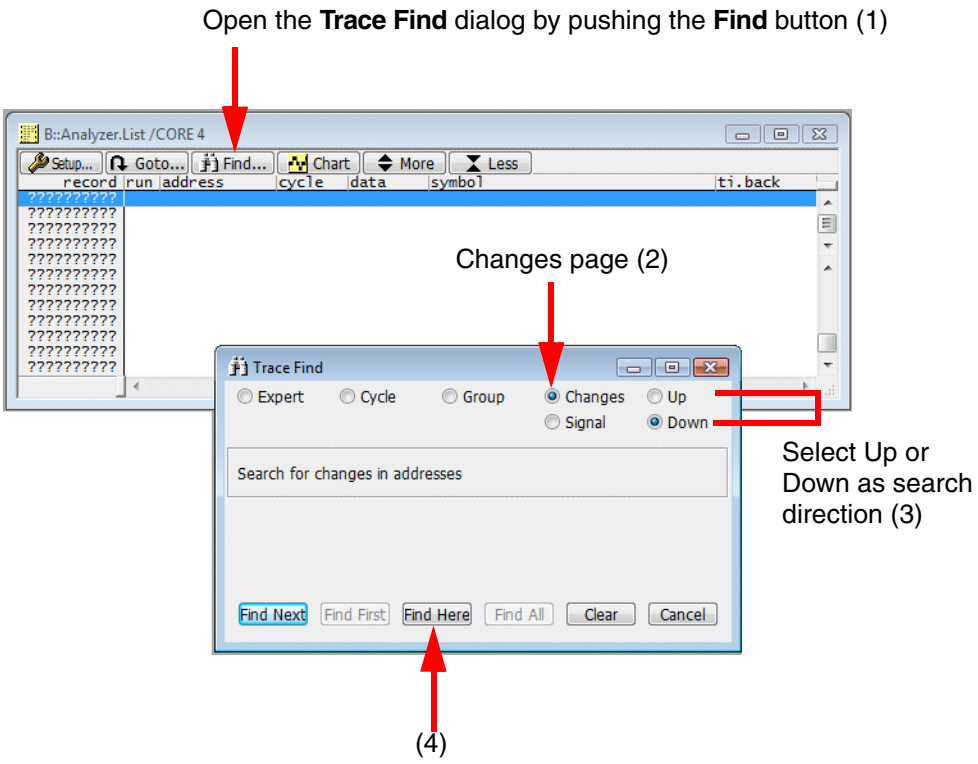


Use the **Trace Find** dialog to specify your event

Did you know?

If no trace information is available for the hardware thread, you can get to a trace area with information as follows:

1. Open the **Trace Find** dialog by pushing the **Find** button.
2. Select the **Changes** page.
3. Select either **Up** or **Down** as search direction.
4. Push **Find Here** to start the search.



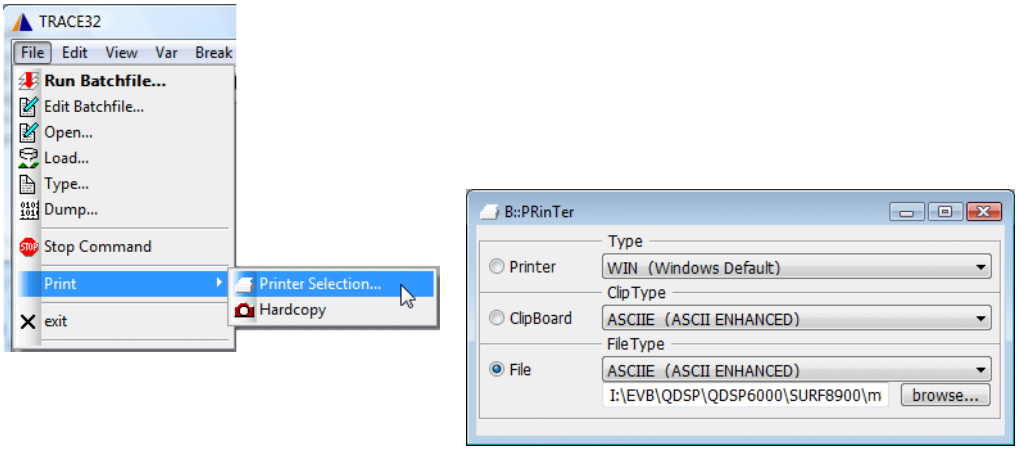
Belated Trace Analysis

There are several ways for a belated trace analysis:

1. Save a part of the trace contents into an ASCII file and analyze this trace contents by reading.
2. Save the trace contents in a compact format into a file. Load the trace contents at a subsequent date into a TRACE32 Instruction Set Simulator and analyze it there.
3. Export the ETMv3 byte stream to postprocess it with an external tool.

Saving part of the trace contents to an ASCII file requires the following steps:

1. Choose **File** menu > **Print**, and then specify the file name and the output format.




```
PRinTer.FileType ASCIIE                 ; Specify output format
                                        ; here (ASCII enhanced)

PRinTer.FILE testrun1.lst               ; Specify the file name
```

2. It only makes sense to save a part of the trace contents into an ASCII-file. Use the record numbers to specify the trace part you are interested in.

   TRACE32 provides the command prefix **WinPrint.** to redirect the result of a display command into a file.

```
; Save the trace record range (-8976.)--(-2418.) into the
; specified file
WinPrint.Trace.List (-8976.)--(-2418.)
```

3. Use an ASCII editor to display the result.

The following command allows you to save the trace information to a file:

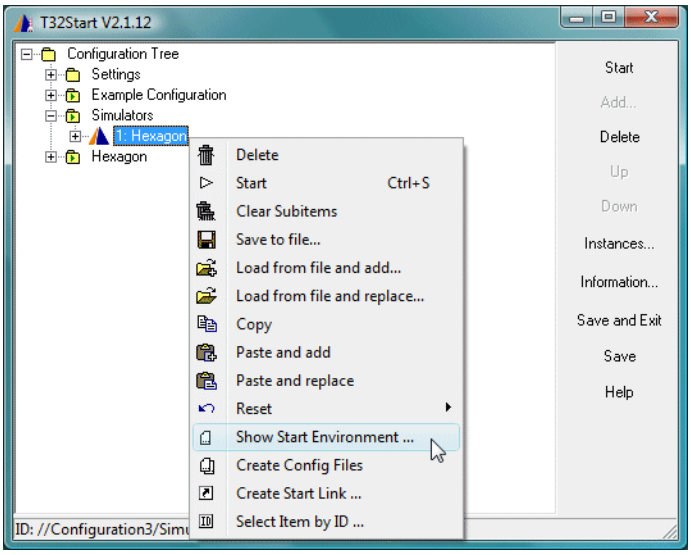
Trace.SAVE *<file>*

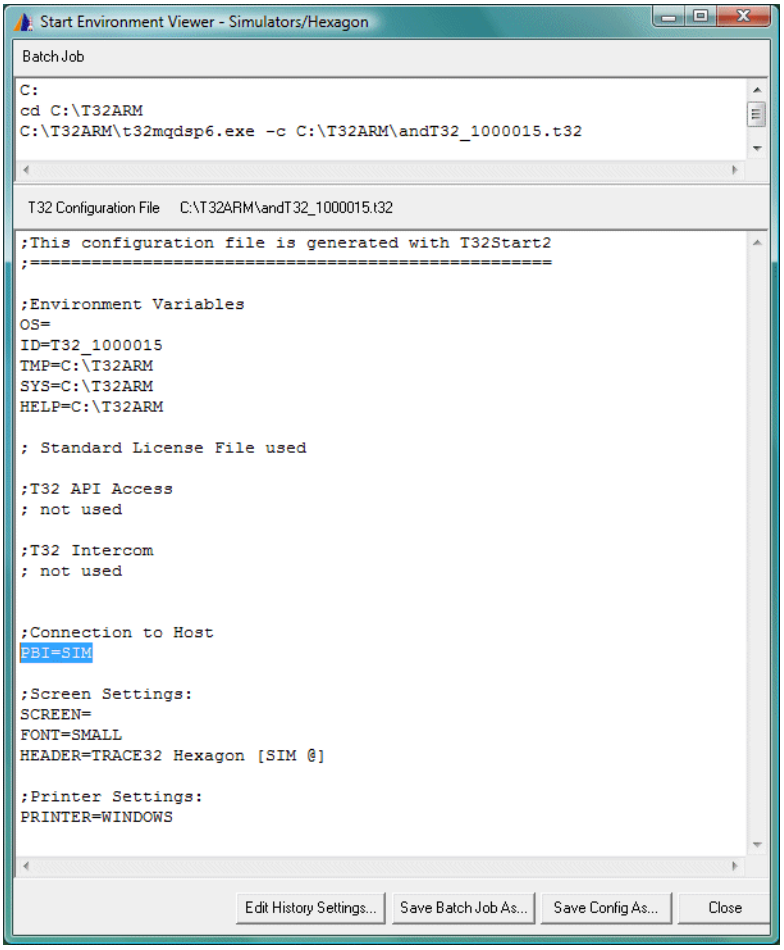
Analyzing the trace contents within a TRACE32 simulator requires the following three steps:

1. Save the contents of the trace memory to a file.

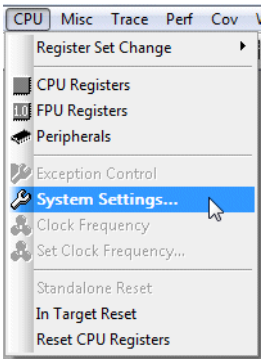
```
Trace.SAVE testrun1                    ; The following information
                                       ; is saved to file:
                                       ; - Raw data
                                       ; - Merged source code
                                       ; - Timing information
```

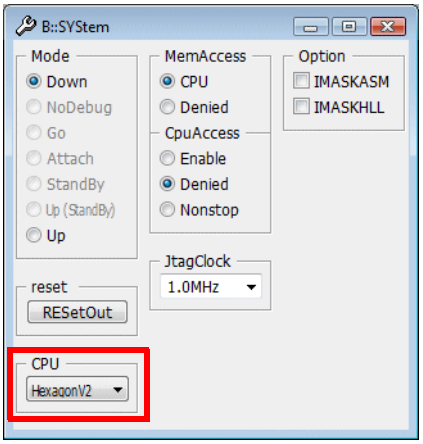
2. Start a TRACE32 Instruction Set Simulator (PBI=SIM).





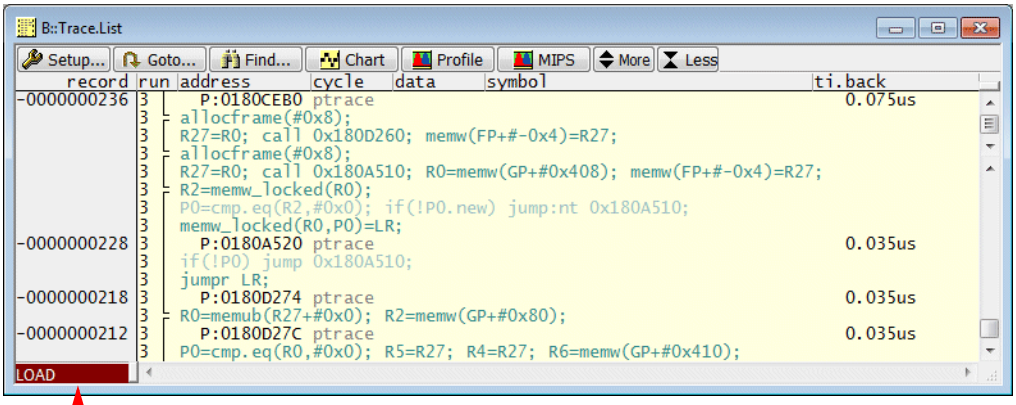
3. Select your target CPU within the simulator.





4. Load the trace file.

```
Trace.LOAD testrun1

Trace.List                                  ; Display a trace listing
```



LOAD indicates that the source for the trace information is the loaded file.

5. Load symbol and debug information if you need it.

```
Data.LOAD.Elf blast/bootimg.pbn /NoCODE
```

The TRACE32 Instruction Set Simulator provides the same trace display and analysis commands as the TRACE32 debugger.

	Please be aware that analyzing the trace in the TRACE32 Instruction Set Simulator will require a more complex setup if the MMU is used. (no example for testing available)

Export the Trace Information as ETMv3 Byte Stream

TRACE32 allows to save the ETMv3 byte stream into a file for further analysis by an external tool.

```
Trace.EXPORT testrun1.ad /ByteStream

; Export only a part of the trace contents
Trace.EXPORT testrun2.ad (-3456800.)--(-2389.) /ByteStream
```

Function Run-Times Analysis

All commands for the function run-time analysis introduced in this chapter use the contents of the trace repository as base for their analysis.

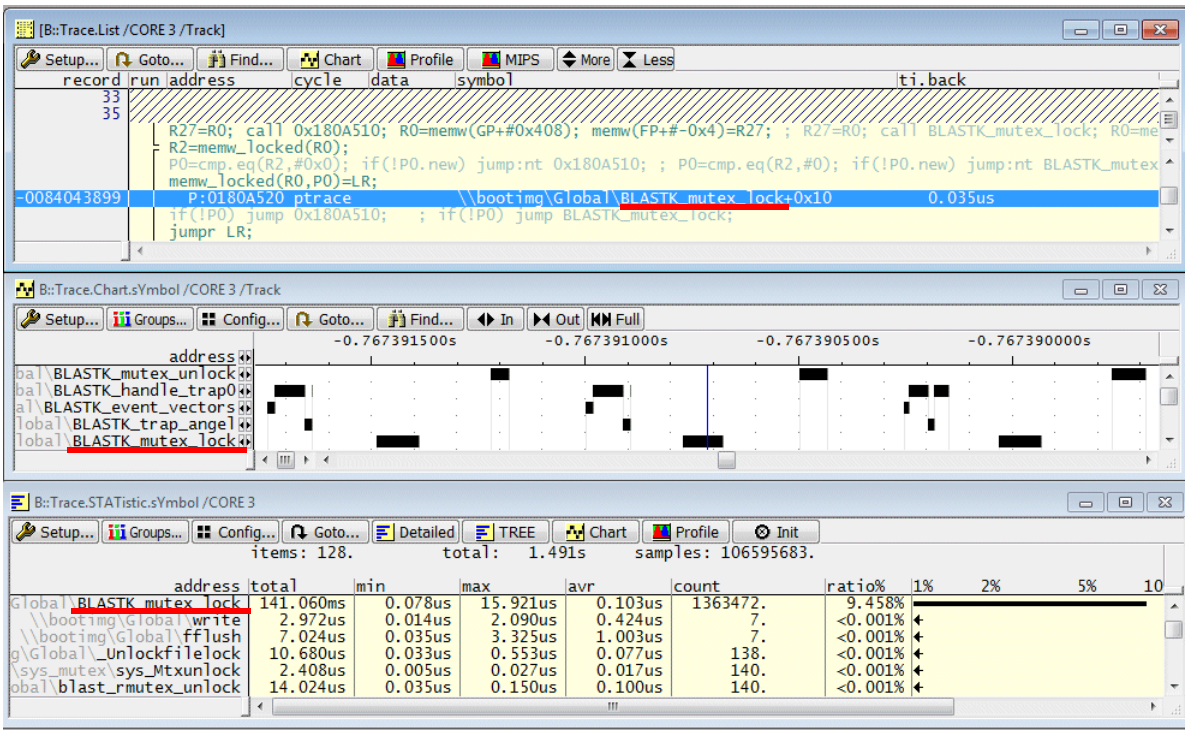
For the function run-time analysis it is helpful to differentiate between three types of application software:

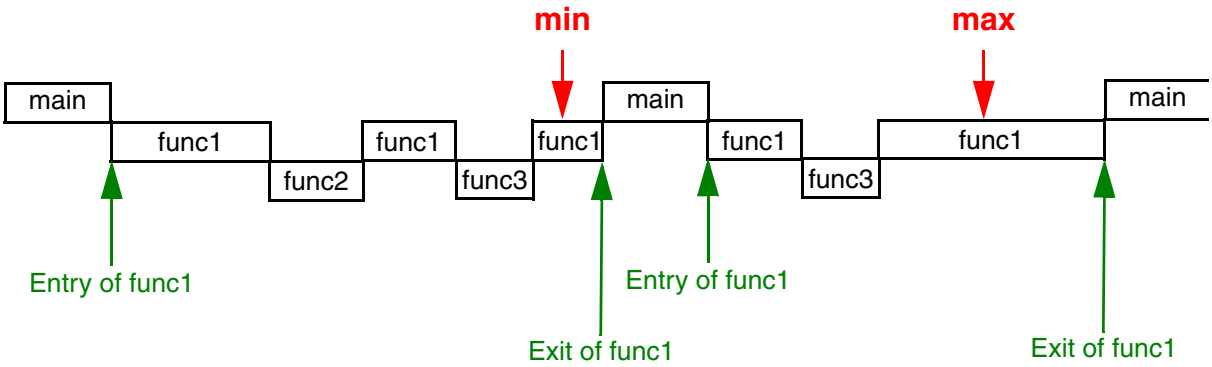
1. Software without operating system (abbreviation: **no OS**)
2. Software with an operating system without dynamic memory management (abbreviation: **OS**).
3. Software with an operating system that uses dynamic memory management to handle processes/tasks (abbreviation: **OS+MMU**). If an OS+MMU is used, several processes/tasks run at the same virtual addresses.

Flat vs. Nesting Analysis

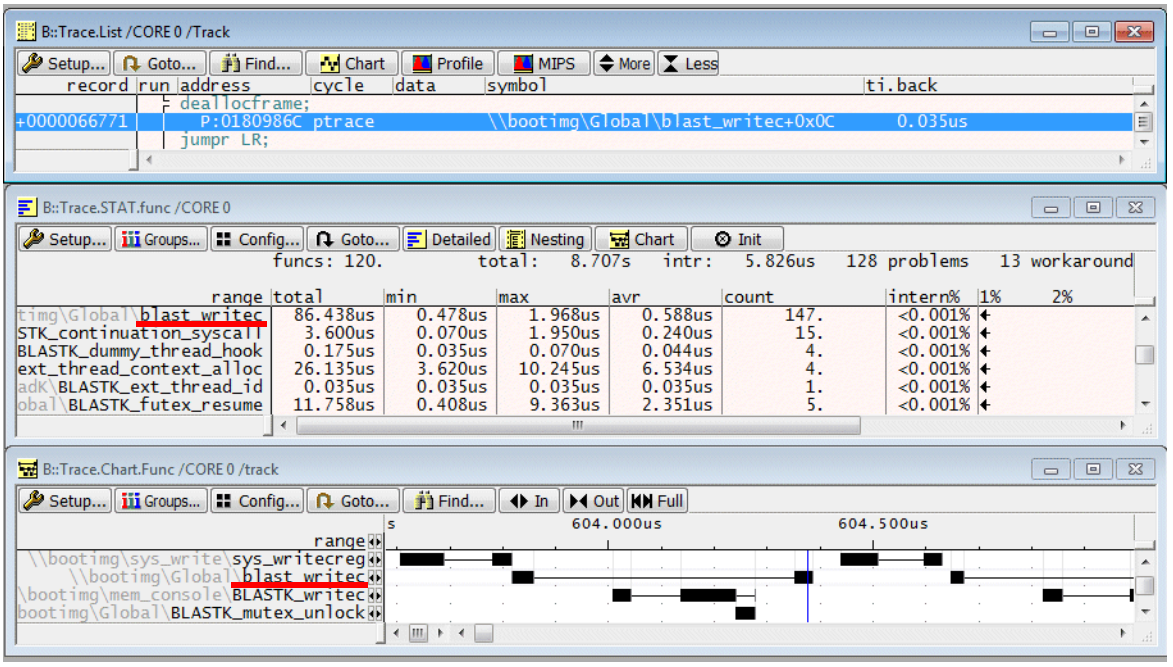
Basic Knowledge about the Flat Analysis

The flat analysis bases on the symbolic instruction addresses of the trace entries. The time spent by an instruction packet is assigned to the corresponding function.





min	shortest time continuously in the address range of a function/symbol range
max	longest time continuously in the address range of a function/symbol range



For the function run-time analysis with nesting, the TRACE32 software scans the trace contents in order to find:

1. **Function entries**

   The execution of the first instruction of an HLL function is regarded as function entry.

   Additional identifications of function entries are implemented depending on the processor architecture and the compiler used.

2. **Function exits**

   A RETURN instruction within an HLL function is regarded as function exit.

   Additional identifications of function exits are implemented depending on the processor architecture and the compiler used.

3. **Entries to interrupt service routines (asynchronous)**

   Interrupts are identified as follows:

   - An entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt.

   The HLL function started following the interrupt is regarded as interrupt service routine.

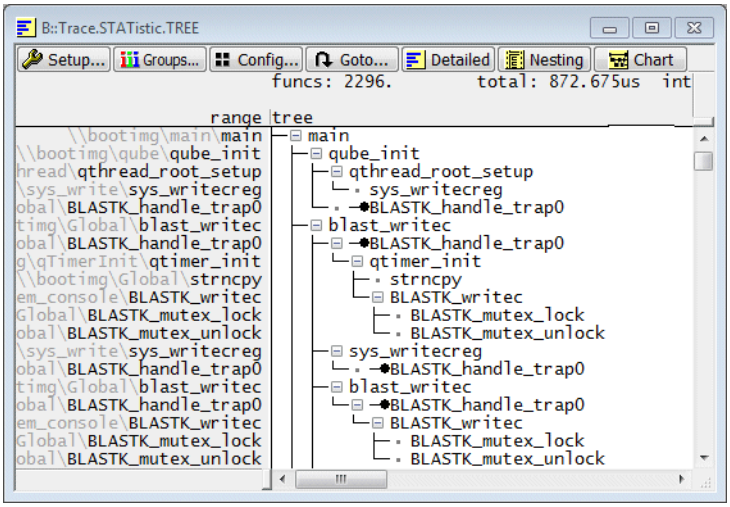
   If a RETURN is detected before the entry to this HLL function, TRACE32 assumes that there is an assembler interrupt service routine. This assembler interrupt service routine has to be marked explicitly if it should be part of the function run-time analysis (**sYmbol.MARKER.Create FENTRY/FEXIT**).

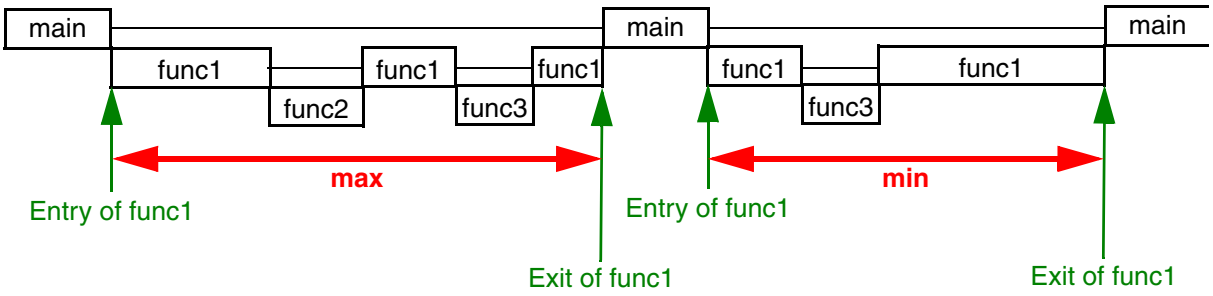
4. **Exits of interrupt service routines**

5. **Entries to TRAP handlers (synchronous)**

6. **Exits of TRAP handlers**

Based on the results a complete call tree is constructed.





min	shortest time within the function including all subfunctions and traps
max	longest time within the function including all subfunctions and traps

Summary

The nesting analysis provides more details on the structure and the timing of the program run, but it is much more sensitive then the flat analysis. Missing or tricky function exits for example result in a worthless nesting analysis.

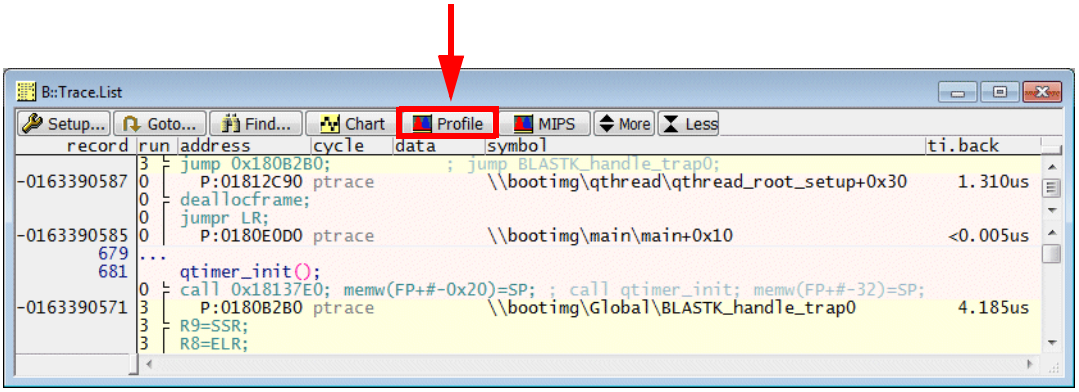
Flat Analysis

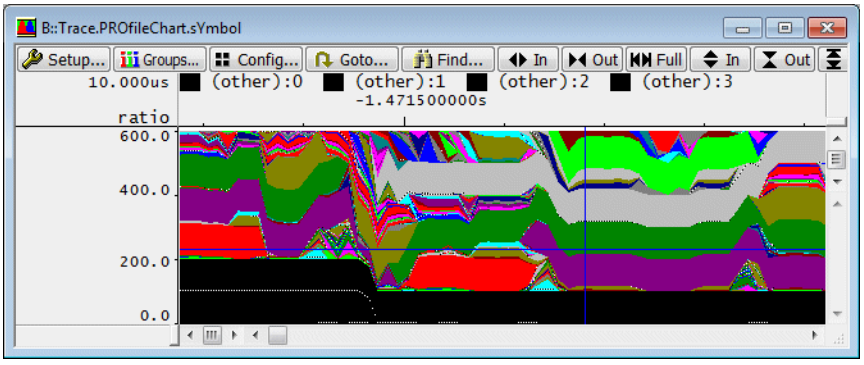
Flat function run-time analysis is easy to use and error-tolerant. It provides analysis results at different levels:

- Overview on the dynamic program behavior
- Timing diagrams of function execution order (function timing diagram)
- Details on the execution of single instructions (hot-spot analysis)

Dynamic Program Behavior (no OS and OS)

Push the **Profile** button to get information on the dynamic behavior of the program.





Trace.PROfileChart.sYmbol [/SplitCORE]

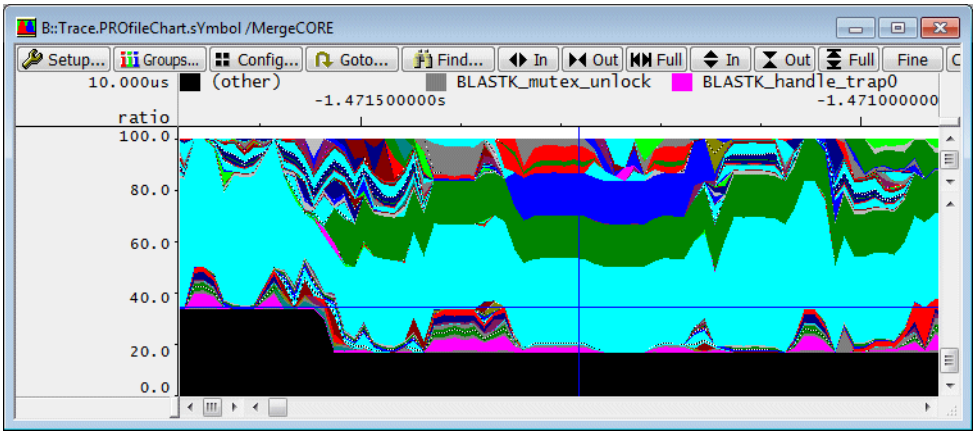
Graphic display of dynamic program behavior

- Analysis independently for each hardware thread
- Individual results for all hardware threads are displayed
- The number after "**:**" represents the hardware thread
- Default option

Trace.PROfileChart.sYmbol /MergeCORE

Graphic display of dynamic program behavior

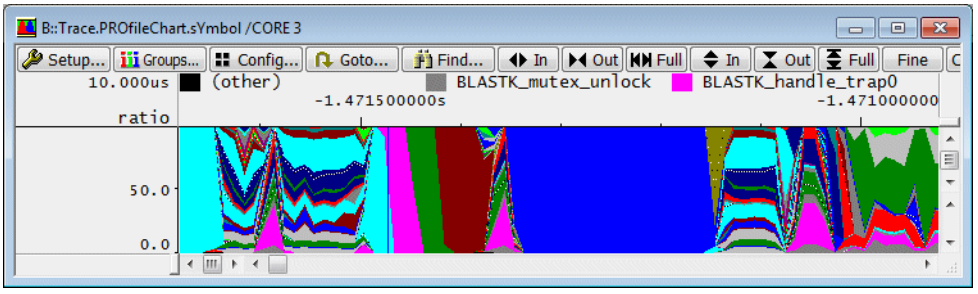
- Analysis independently for each hardware thread
- Results are summarized and displayed as a single result



Trace.PROfileChart.sYmbol /CORE *<n>*

Graphic display of dynamic program behavior

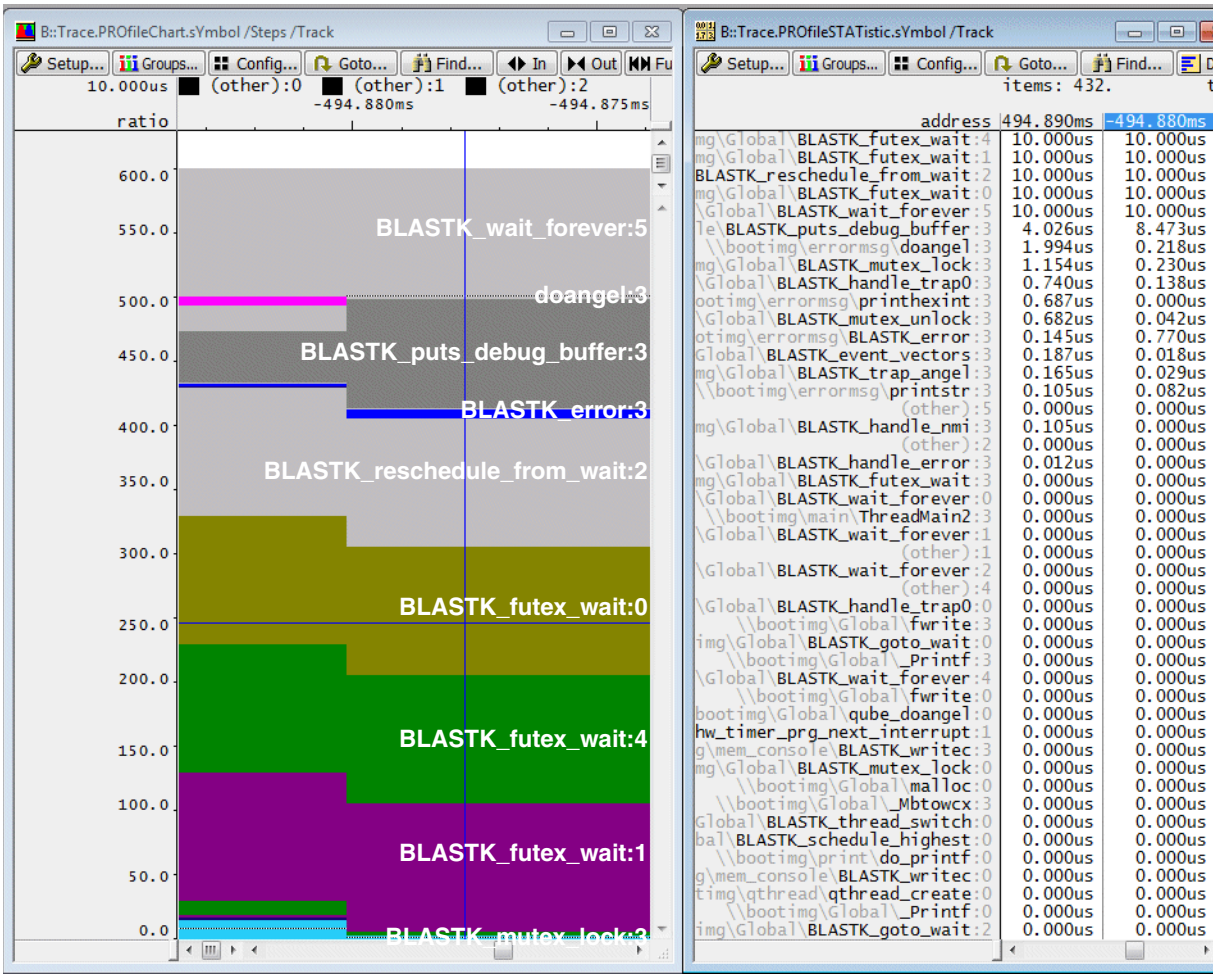
- Analysis for specified hardware thread

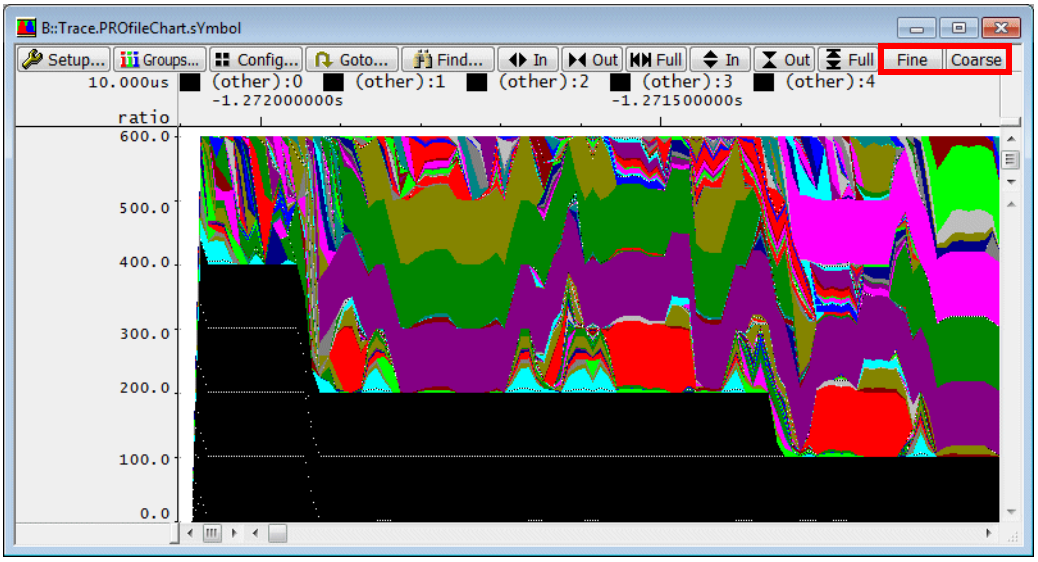


More Details

To draw the **Trace.PROfileChart.sYmbol** graphic, TRACE32 PowerView partitions the recorded instruction flow into time intervals. The default interval size is 10.us.

For each time interval rectangles are drawn that represent the time ratio the executed functions/symbol ranges consumed within the time interval. For the final display this basic graph is smoothed.





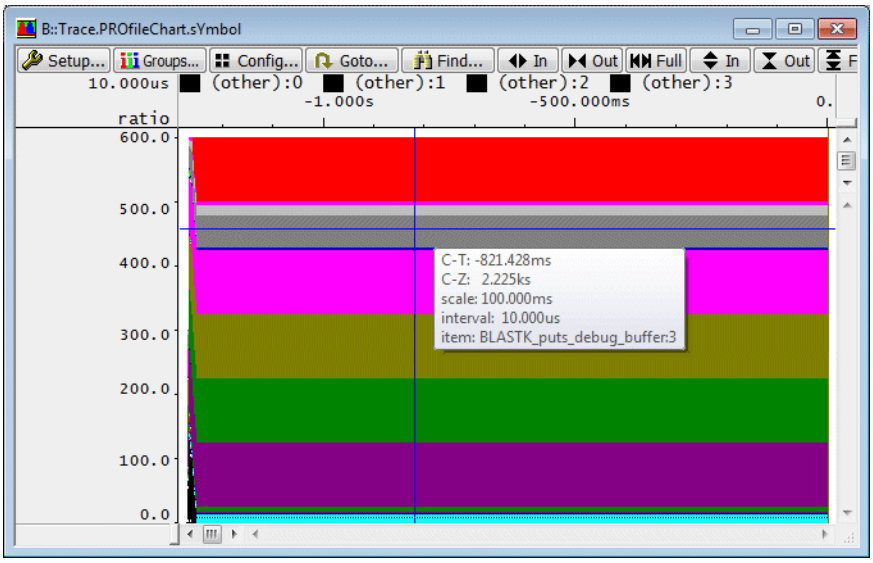
Fine	Decrease the time interval size by the factor 10
Coarse	Increase the time interval size by the factor 10

The time interval size can also be set manually.

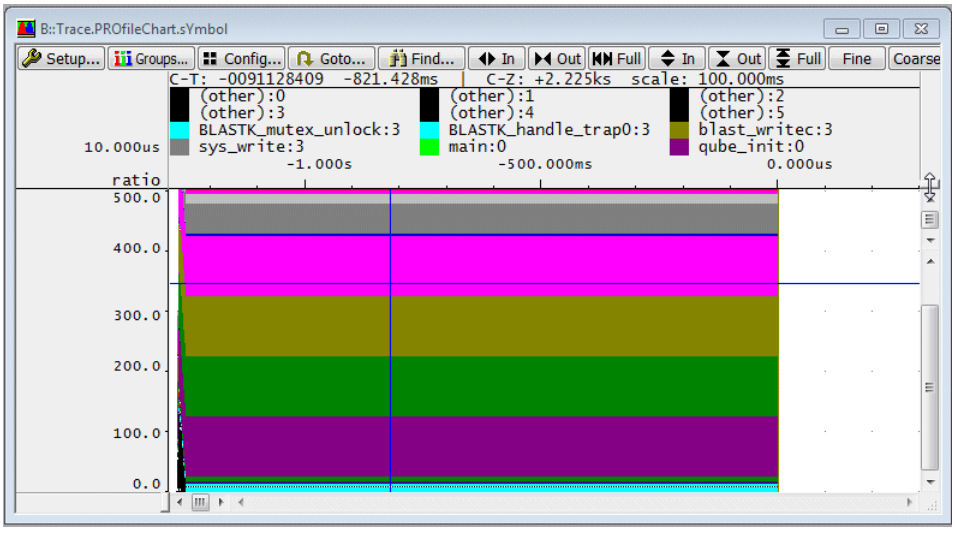
```
Trace.PROfileChart.sYmbol /InterVal 5.ms        ; Change the time
                                                ; segment size to 5.ms
```

Color Assignment - Basics

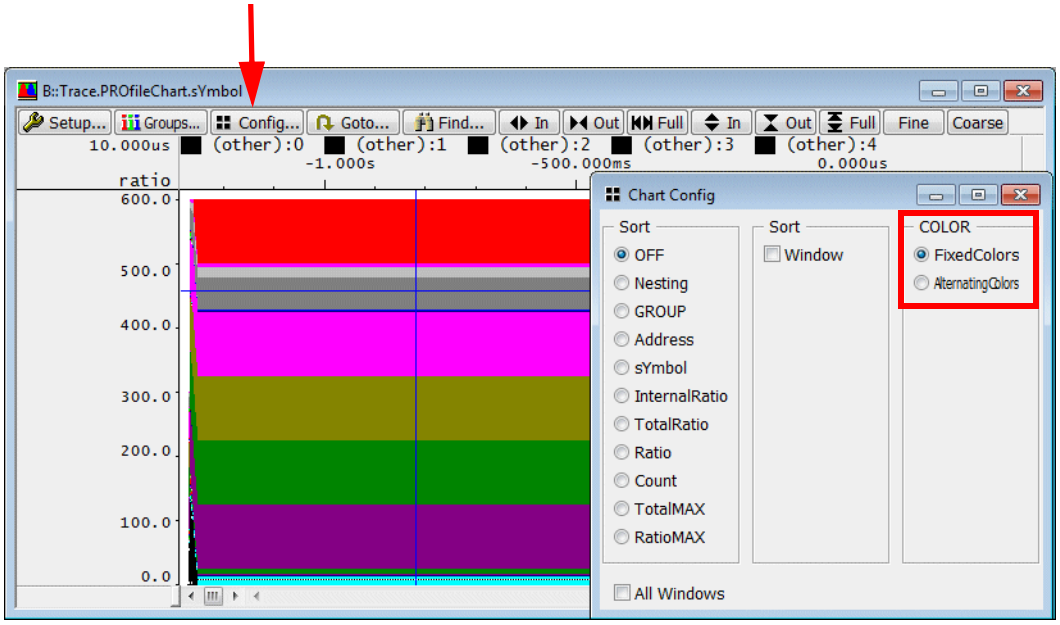
- The tooltip at the cursor position shows the function color assignment (item) and the used interval size.



- Use the control handle on the right upper corner of the **Trace.PROfileChart.sYmbol** window to get a color legend.



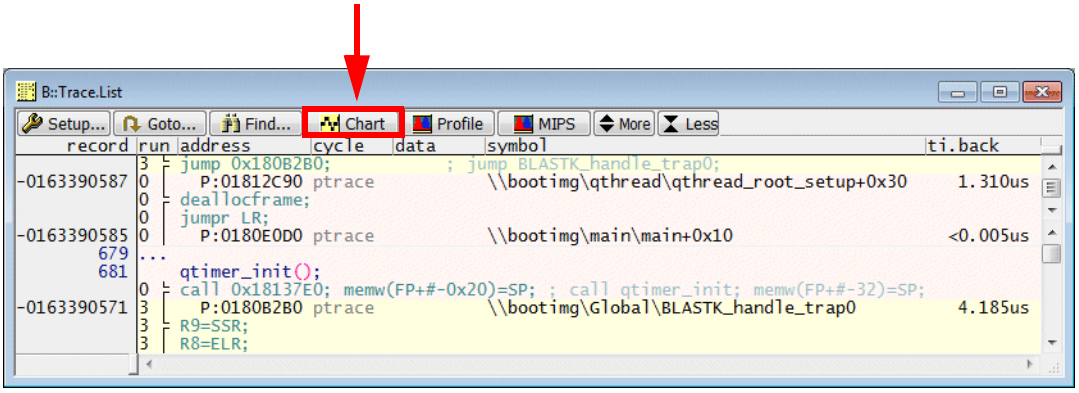
Function Color Assignment - Statically or Dynamically

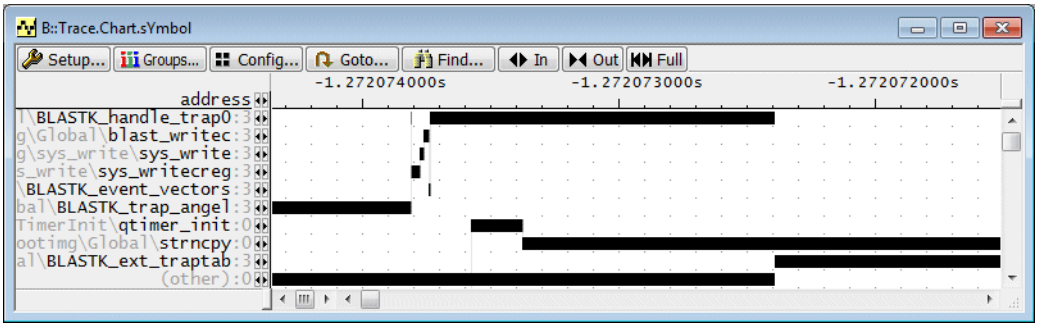


FixedColors	Colors are assigned fixed to functions (default). Fixed color assignment has the risk that two functions with the same color are drawn side by side and thus may convey a wrong impression of the dynamic behavior.
AlternatingColors	Colors are assigned by the recording order of the functions, again and again for each measurement.

Trace.PROfileChart.sYmbol [**/InterVal** *<time>*]	Overview on the dynamic behavior of the program • Graphical display
Trace.PROfileSTATistic.sYmbol [**/InterVal** *<time>*]	Overview on the dynamic behavior of the program • Numerical display for export as comma-separated values
Trace.STATistic.COLOR FixedColors | **AlternatingColors**	Color assignment method

Push the **Chart** button to get a function timing diagram for the captured instruction flow.





Trace.Chart.sYmbol [**/SplitCORE**]

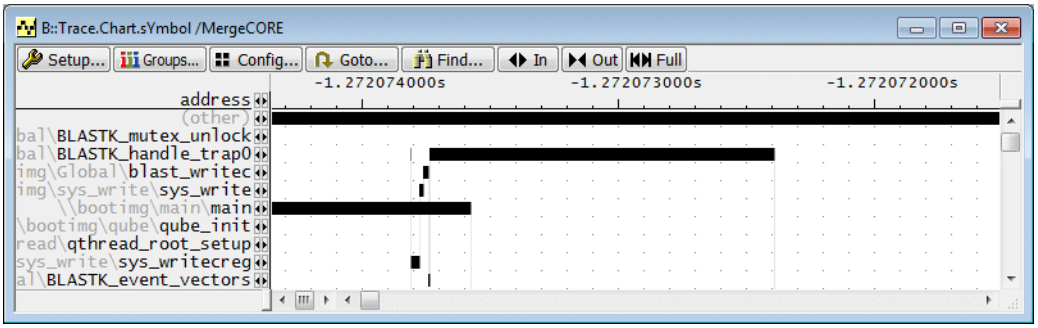
Graphic display of function timing

- Analysis independently for each hardware thread
- Individual results for all hardware threads are displayed
- The number after "**:**" represents the hardware thread
- Default option

Trace.PROfileChart.sYmbol /MergeCORE

Graphic display of function timing

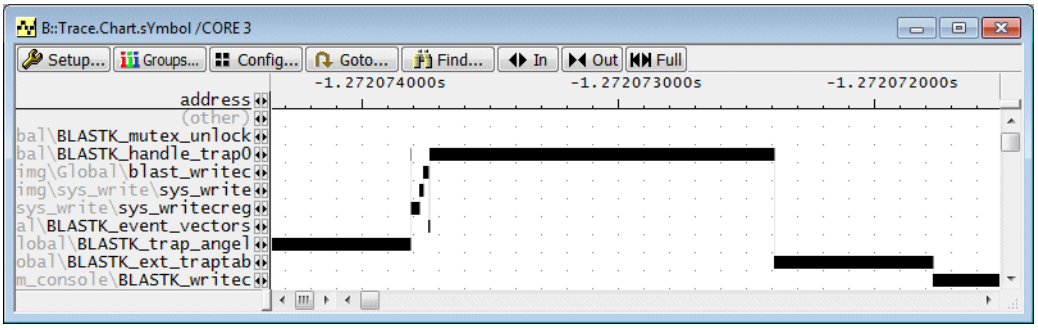
- Analysis independently for each hardware thread
- Results are summarized and displayed as a single result



Trace.PROfileChart.sYmbol /CORE *<n>*

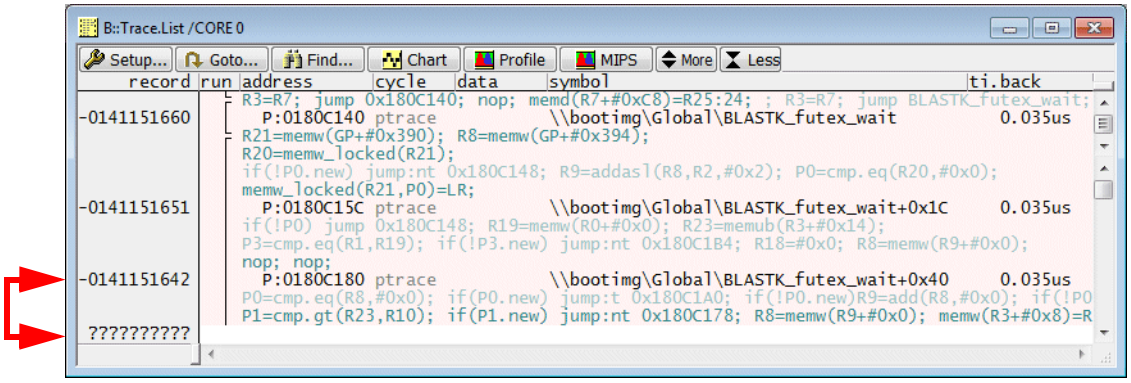
Graphic display of function timing

- Analysis for specified hardware thread



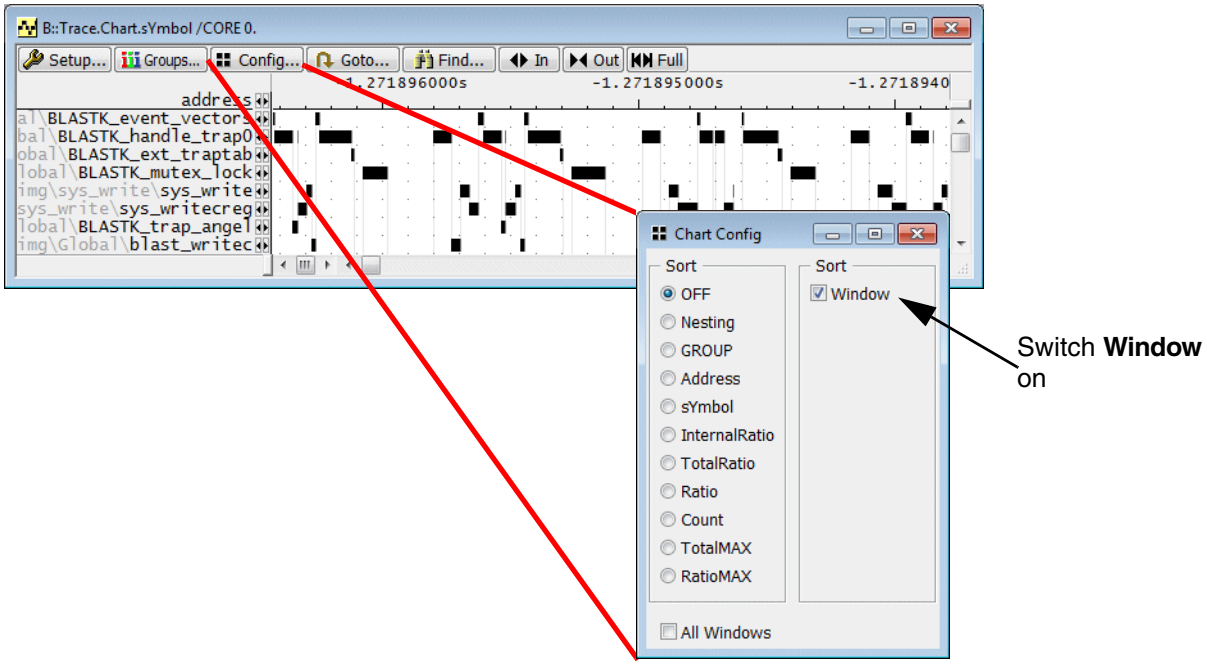
Did you know?

Periods of time for which no trace information is exported (?????) are assigned to the last running function (here *BLASTK_futex_wait*).



Did you know?

If the **Window** check box is selected in the **Chart Config** window, the functions that are active at the selected point of time are visualized in the **Trace.Chart.sYmbol** window. This is helpful especially if you scroll horizontally.



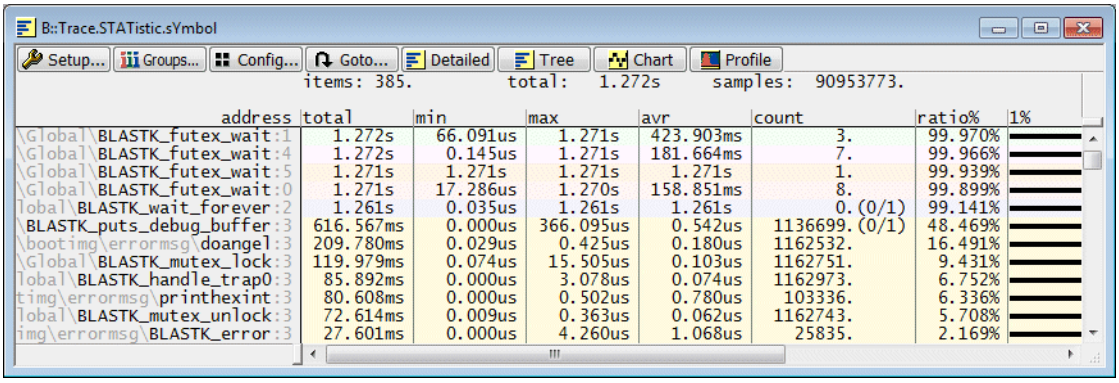
Some trace analysis commands that provide a graphical result have a numerical counterpart.

Trace.Chart.sYmbol	Graphic display of function timing
Trace.STATistic.sYmbol	Numerical display of function timing

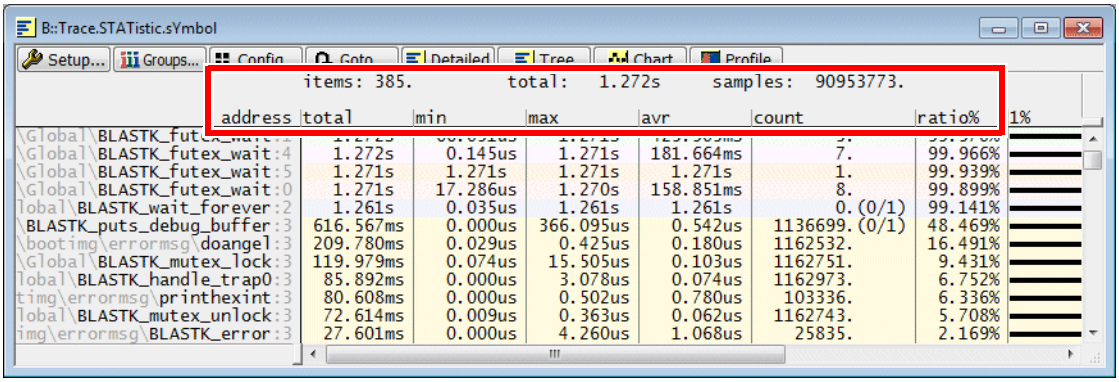
Trace.STATistic.sYmbol [**/SplitCORE**]

Numerical display of function timing

- Analysis independently for each hardware thread
- Individual results for all hardware threads are displayed
- The number after "**:**" represents the hardware thread
- Default option



address	total	min	max	avr	count	ratio%	1%
\Global\BLASTK_futex_wait:1	1.272s	66.091us	1.271s	423.903ms	3.	99.970%	
\Global\BLASTK_futex_wait:4	1.272s	0.145us	1.271s	181.664ms	7.	99.966%	
\Global\BLASTK_futex_wait:5	1.271s	1.271s	1.271s	1.271s	1.	99.939%	
\Global\BLASTK_futex_wait:0	1.271s	17.286us	1.270s	158.851ms	8.	99.899%	
lobal\BLASTK_wait_forever:2	1.261s	0.035us	1.261s	1.261s	0.(0/1)	99.141%	
\BLASTK_puts_debug_buffer:3	616.567ms	0.000us	366.095us	0.542us	1136699.(0/1)	48.469%	
\bootimg\errormsg\doangel:3	209.780ms	0.029us	0.425us	0.180us	1162532.	16.491%	
\Global\BLASTK_mutex_lock:3	119.979ms	0.074us	15.505us	0.103us	1162751.	9.431%	
lobal\BLASTK_handle_trap0:3	85.892ms	0.000us	3.078us	0.074us	1162973.	6.752%	
timg\errormsg\printhexint:3	80.608ms	0.000us	0.502us	0.780us	103336.	6.336%	
lobal\BLASTK_mutex_unlock:3	72.614ms	0.009us	0.363us	0.062us	1162743.	5.708%	
img\errormsg\BLASTK_error:3	27.601ms	0.000us	4.260us	1.068us	25835.	2.169%	

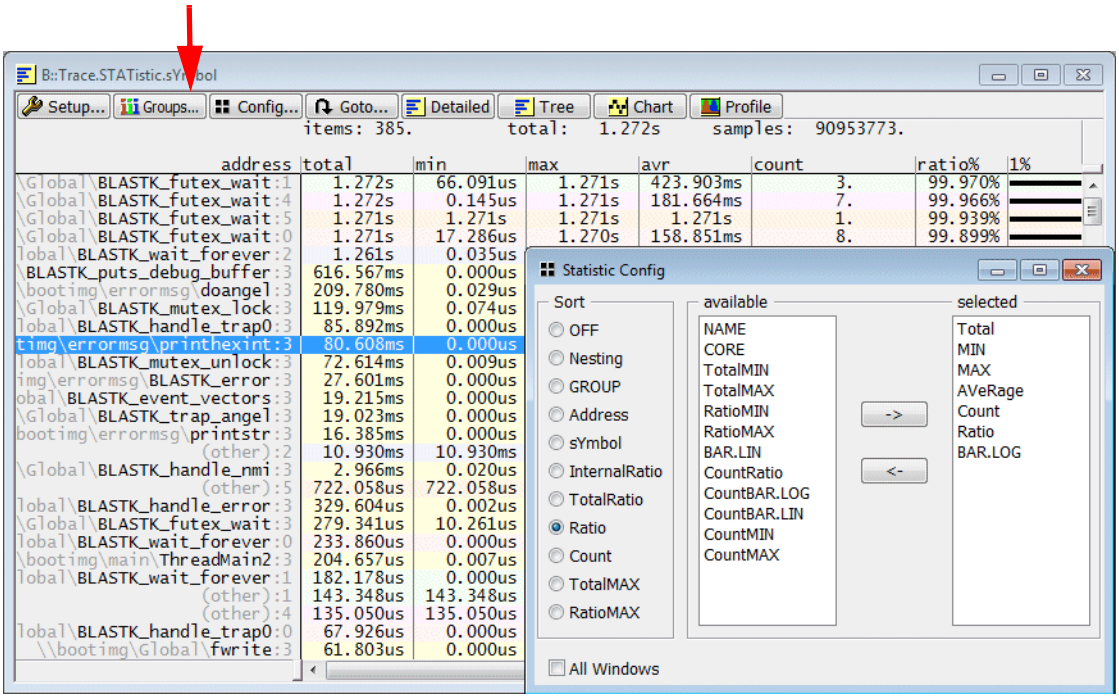


For a description of the list summary and the highlighted columns, see tables below.

List Summary	
item	Number of recorded functions/symbol regions
total	Time period recorded by the trace
samples	Total number of recorded changes of functions/symbol regions (instruction flow continuously in the address range of a function/symbol region)

Columns with function details	
address	Function name (other) program sections that can not be assigned to a function/symbol region
total	Time period in the function/symbol region during the recorded time period
min	Shortest time continuously in the address range of the function/symbol region
max	Longest time continuously in the address range of the function/symbol region
avr	Average time continuously in the address range of the function/symbol region (calculated by total/count)
count	Number of new entries into the address range of the function/symbol region (start address executed)
ratio	Ratio of time in the function/symbol region with regards to the total time period recorded

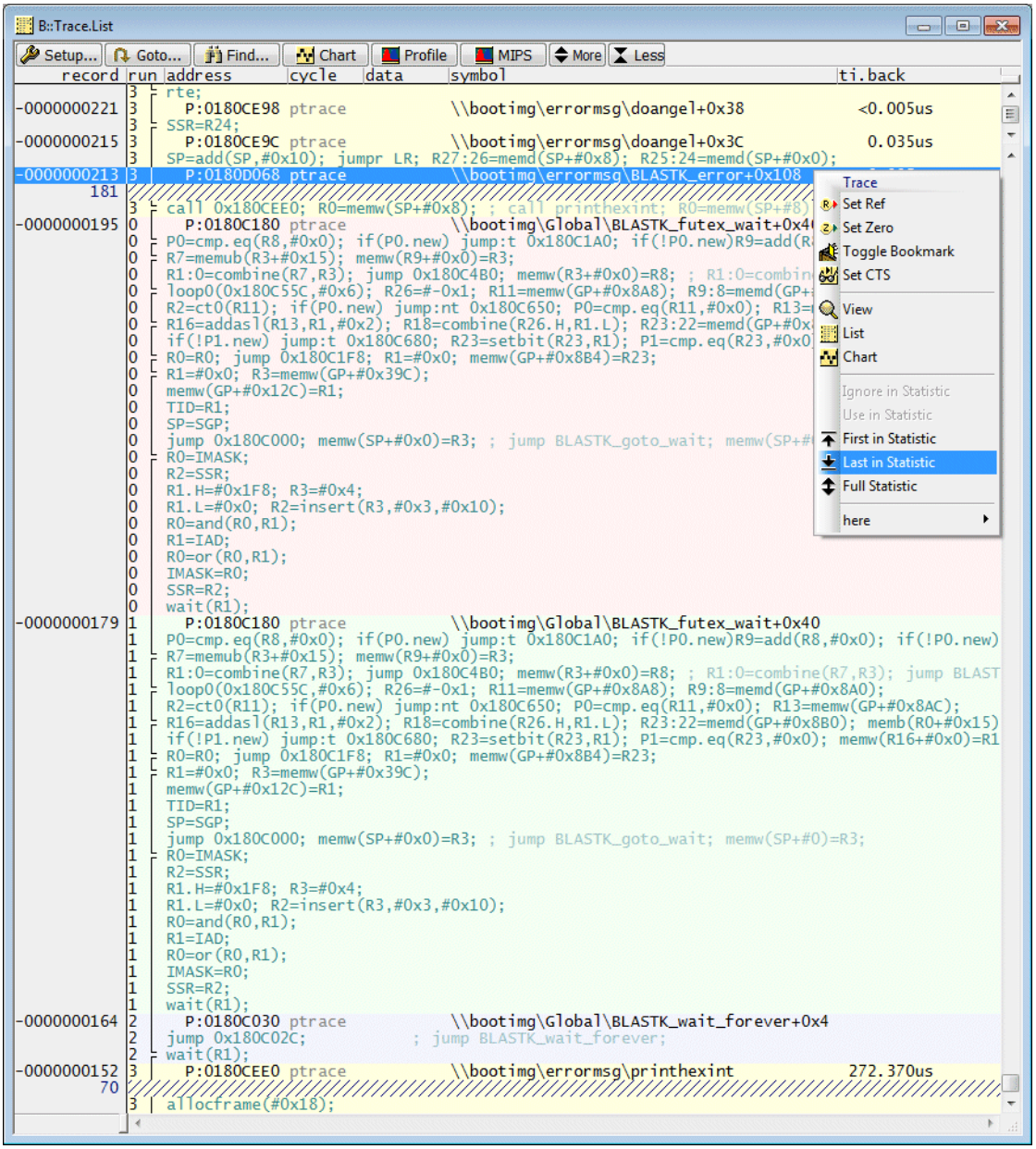
Pushing the **Config** button provides the possibility to specify a different sorting criterion or a different column layout



Trace.STATistic.sYmbol /MergeCORE	Numerical display of function timing • Analysis independently for each hardware thread • Results are summarized and displayed as a single result
Trace.STATistic.sYmbol /CORE *<n>*	Numerical display of function timing • Analysis for specified hardware thread

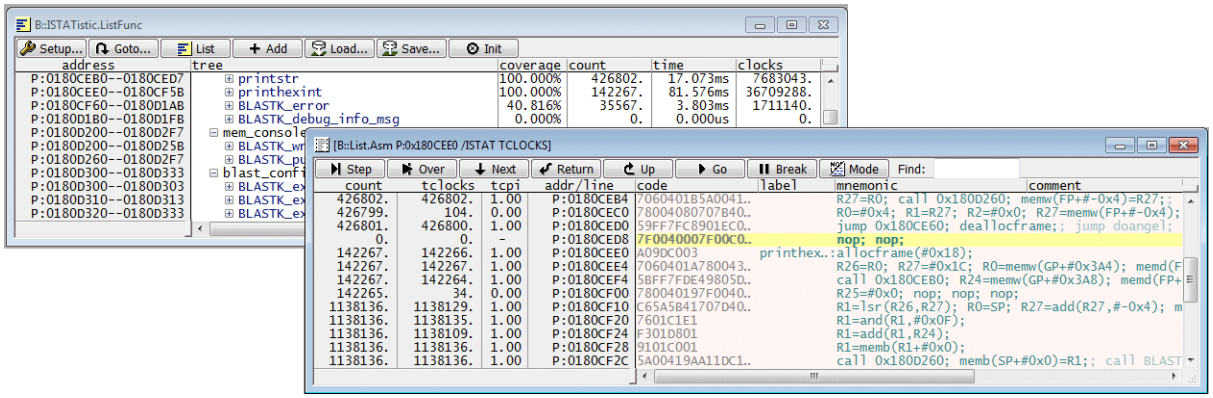
Did you know?

TRACE32 flushes all trace information stuck in the ETM fifos when the recording to the trace repository is stopped because the program execution stopped. These delayed exported trace packets can be identified by no **TIme.Back** value or by a large **TIme.Back** value.
These delayed exported trace packets can falsify the run-time analysis. So it is recommended to exclude them from the analysis. This is done by tagging the last not-delayed trace packet as **"Last in Statistic"**:




```
Trace.STATistic.LAST -213.                ; Specify the last record that
                                          ; should be included into the
                                          ; statistic analysis, the rest
                                          ; will be ignored
```

If a function seems to be very time consuming, details on the run-time of single instruction packets can be displayed with the help of the **ISTATistic** command group.



Preparation

The run-time results on single instruction packets are more accurate if cycle-accurate tracing is used.

```
ETM.CycleAccurate ON                    ; Switch cycle accurate tracing on

Trace.CLOCK 600.MHz                     ; Inform TRACE32 about your core
                                        ; frequency
```

A high number of local FIFOFULLs might affect the result of the instruction statistic.

Processing

The command group **ISTATistic** works with a database. The measurement includes the following steps:

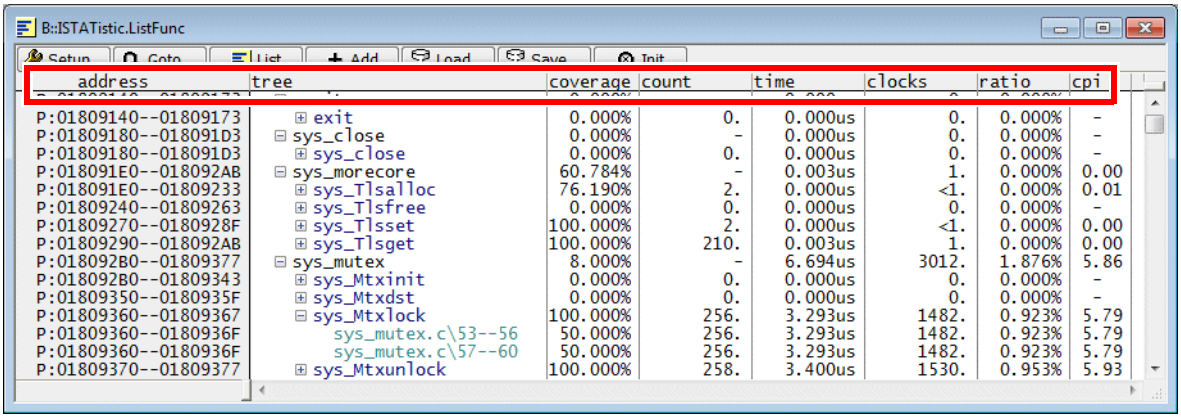
1. Enable cycle-accurate tracing.
2. Specify the core clock frequency.
3. Clear the database.
4. Fill the trace repository.
5. Transfer the contents of the trace repository to the database.
6. Display the result.
7. (Repeat step 4-6 if required).

Main commands:

ETM.CycleAccurate ON	Switch cycle-accurate tracing on.
Trace.CLOCK *<core_clock>*	Inform TRACE32 about your core frequency.
Trace.FLOWPROCESS	Upload the complete trace contents to the host and merge it with the program code/debug information
ISTATistic.RESet	Clear the Instruction Statistic database.
ISTATistic.ADD [**/MergeCORE**]	Transfer the trace information of all hardware threads from the trace repository to the Instruction Statistic database. Default
ISTATistic.ADD /CORE *<n>*	Transfer the trace information of the specified hardware thread from the trace repository to the Instruction Statistic database.
ISTATistic.ListFunc	List flat function run-time analysis based on the added trace information.
Data.List *<address>* **/ISTAT TCLOCKS**	List flat run-time analysis for the single instruction packets.

A detailed flat function run-time analysis for all hardware threads can be performed as follows:

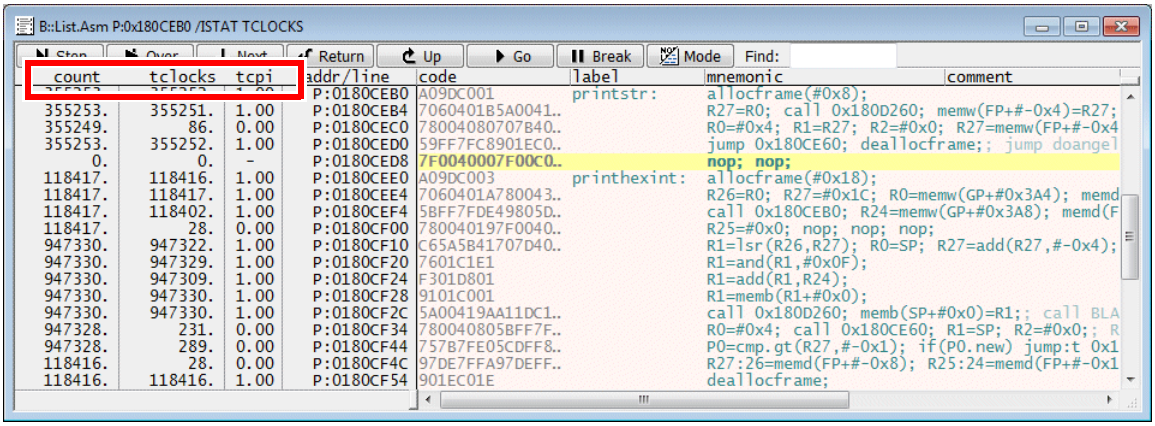
```
ETM.CycleAccurate ON                    ; Switch cycle accurate tracing on

Trace.CLOCK 600.MHz                     ; Inform TRACE32 about your core
                                        ; frequency

ISTATistic.RESet                        ; Reset Instruction Statistic Data
                                        ; Base

Trace.Mode Leash                        ; Switch trace to Leash mode

Go                                      ; Start program execution

WAIT !RUN()                             ; Wait until program stops

Trace.FlowProcess                       ; Process the trace information

IF Trace.FLOW.FIFOFULL>6000.
     PRINT "Warning: Please control the FIFOFULLS"

ISTATistic.ADD                          ; Add trace information for all
                                        ; hardware threads to Instruction
                                        ; Statistic database

ISTATistic.ListFunc                     ; List flat function run-time
                                        ; statistic
```



For a description of the highlighted columns, see table below.

Columns	Description
address	Address range of the module, function or HLL line
tree	Flat module/function/HLL line tree
coverage	Code coverage of the module, function or HLL line
count	Number of module/function/HLL line executions
time	Total time spent by the module, function or HLL line
clocks	Total number of clocks spent by the module, function or HLL line
ratio	Percentage of the total measurement time spent in the module, function or HLL line
cpi	Average clocks per instruction packet for the function or the HLL line

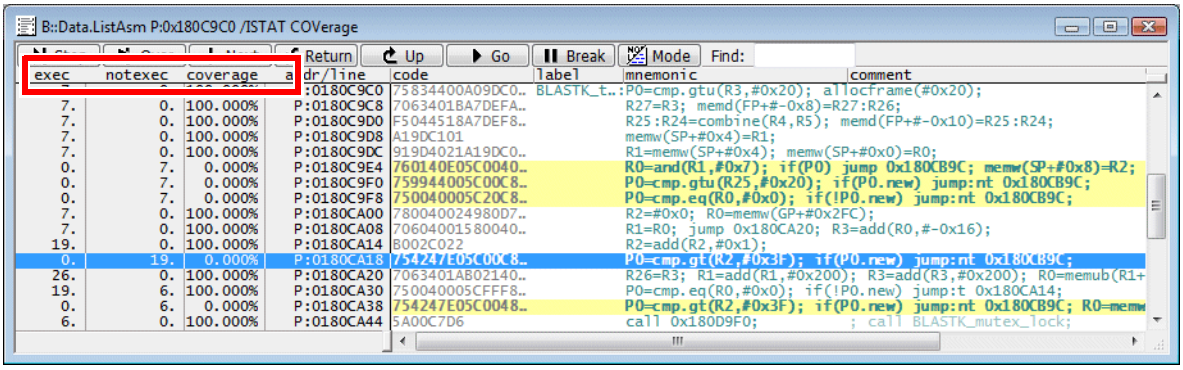
```
List.Asm /ISTAT TCLOCKS                ; List instruction packet run-time
                                       ; statistic
                                       ; - Display time information per
                                       ;   thread
```



For a description of the highlighted columns, see below.

Columns	Description
count	Total number of instruction packet executions
tclocks	Total number of thread clocks for the instruction packet (tclocks = 1/6 clocks)
tcpi	Average thread clocks per instruction packet

```
Data.ListAsm /ISTAT COVerage          ; List instruction packet coverage
```



For a description of the highlighted columns, see below.

Columns	Description
exec	**Conditional instructions:** number of times the instruction packet was executed because the condition was true. **Other instructions:** number of times the instruction packet was executed
notexec	**Conditional instructions:** number of times the instruction packet wasn't executed because the condition was false.
coverage	Instruction packet coverage

If exec or/and notexec is 0 for an instruction packet with condition, the instruction packet is bold-printed against a yellow background. All other instruction packets are bold-printed on a yellow background if they were not executed.

Nesting Analysis

Fundamentals

1. **The nesting analysis analyses only HLL functions.**
2. **The nesting analysis expects common ways to enter/exit functions.**
3. **The result of the nesting analysis is sensitive with regards to FIFOFULLs.**

No OS

Trace.Chart.Func	Graphic display of nested function run-time analysis
Trace.STATistic.Func	Numerical display of nested function run-time analysis

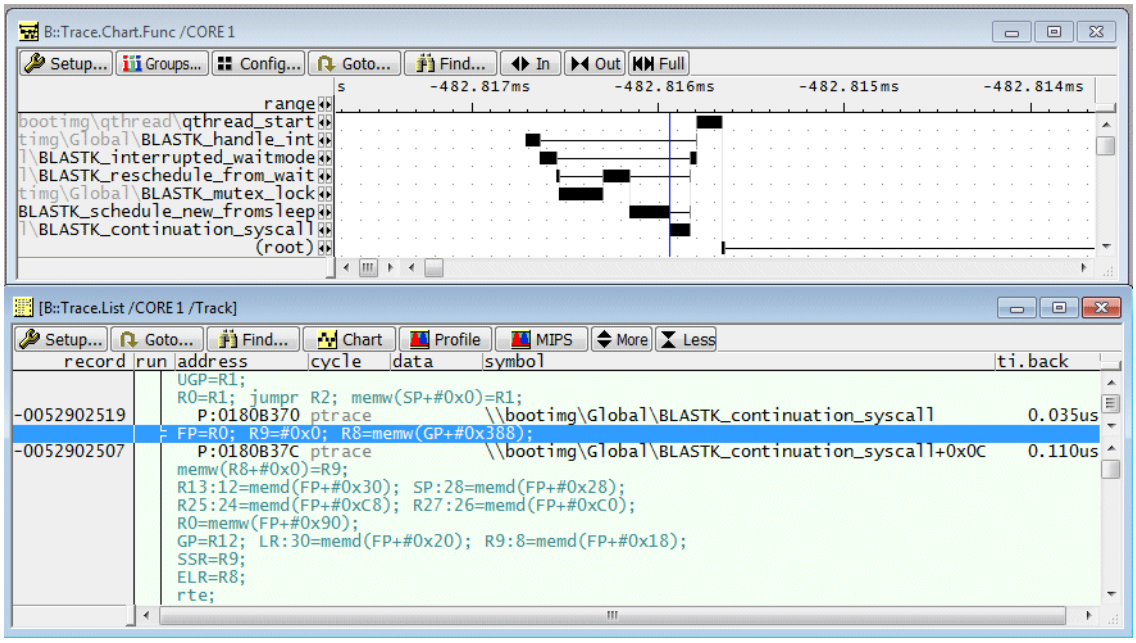
The TRACE32 software scans the trace contents in order to find:

- **Function entries**

  The execution of the first instruction of an HLL function is regarded as function entry.

  Additional identifications for function entries are implemented depending on the processor architecture and the used compiler.

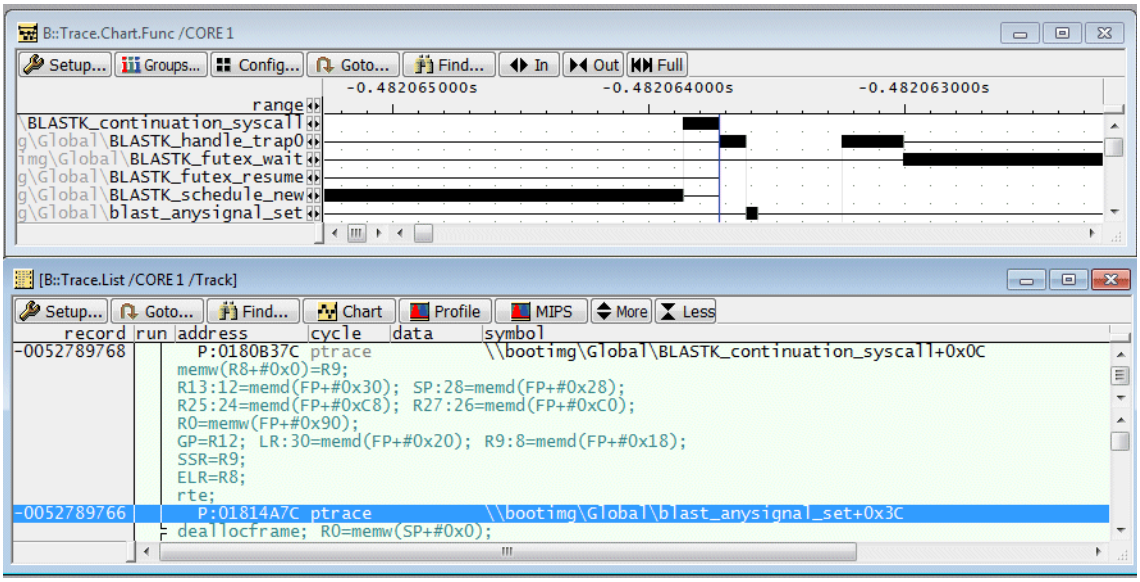
```
Trace.Chart.Func /CORE 1                ; Function
                                        ; BLASTK_continuation_syscall
                                        ; as example

Trace.List /CORE 1 /Track
```



- **Function exits**

  A RETURN instruction within an HLL function is regarded as function exit.

  Additional identifications for function exits are implemented depending on the processor architecture and the used compiler.



- **Entries to interrupt service routines (asynchronous)**

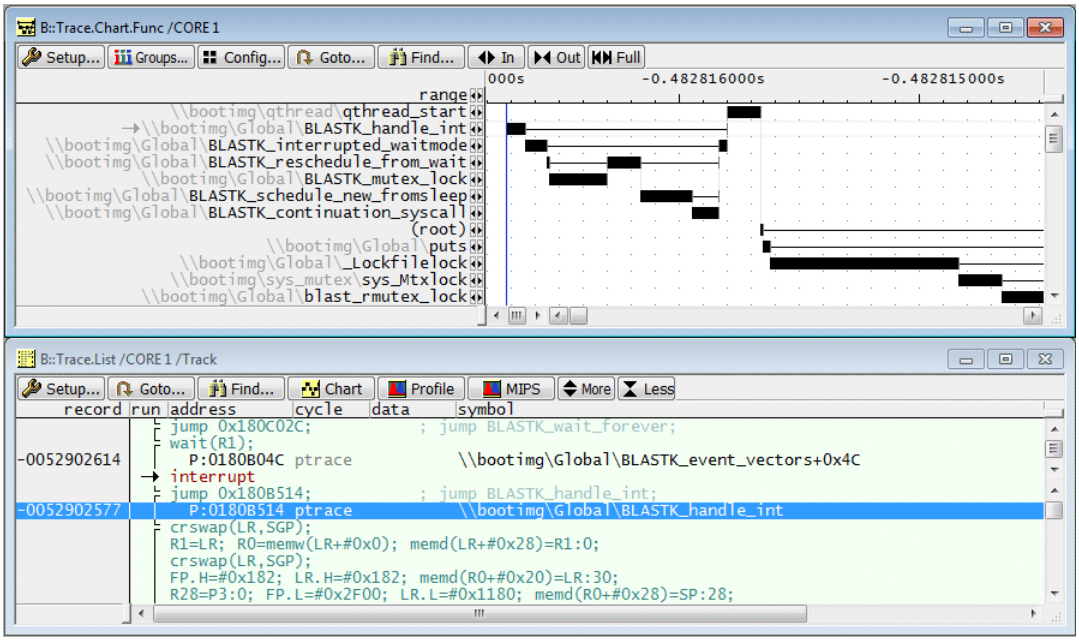
  Interrupts are identified as follows:

  - An entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt.

  The HLL function started following the interrupt is regarded as interrupt service routine.

  If a RETURN is detected before the entry to this HLL function, TRACE32 assumes that there is an assembler interrupt service routine. This assembler interrupt service routine has to be marked explicitly if it should be part of the function run-time analysis (**sYmbol.MARKER.Create FENTRY/FEXIT**).

```
Trace.Chart.Func /CORE 1                ; Function BLASTK_handle_int
                                        ; as example

Trace.List /CORE 1 /Track
```



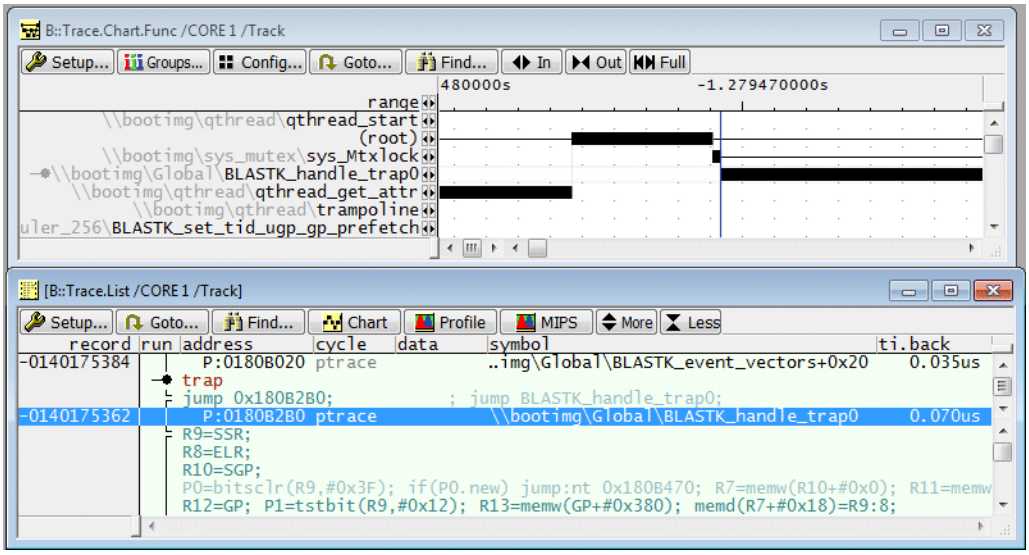
- **Exits of interrupt service routines**

  A RETURN / RETURN FROM INTERRUPT within the HLL interrupt service routine is regarded as exit of the interrupt service routine.

- **Entries to TRAP handlers (synchronous)**

  If an entry to the vector table is identified and if the vector address indicates a synchronous interrupt/trap the following entry to an HLL function is regarded as entry to the trap handler.

```
Trace.Chart.Func /CORE 0                ; Function BLASTK_handle_trap0
                                        ; as example

Trace.List /CORE 0 /Track
```

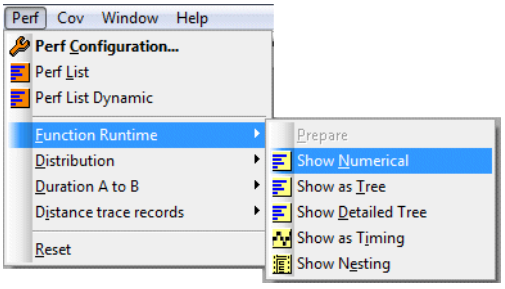


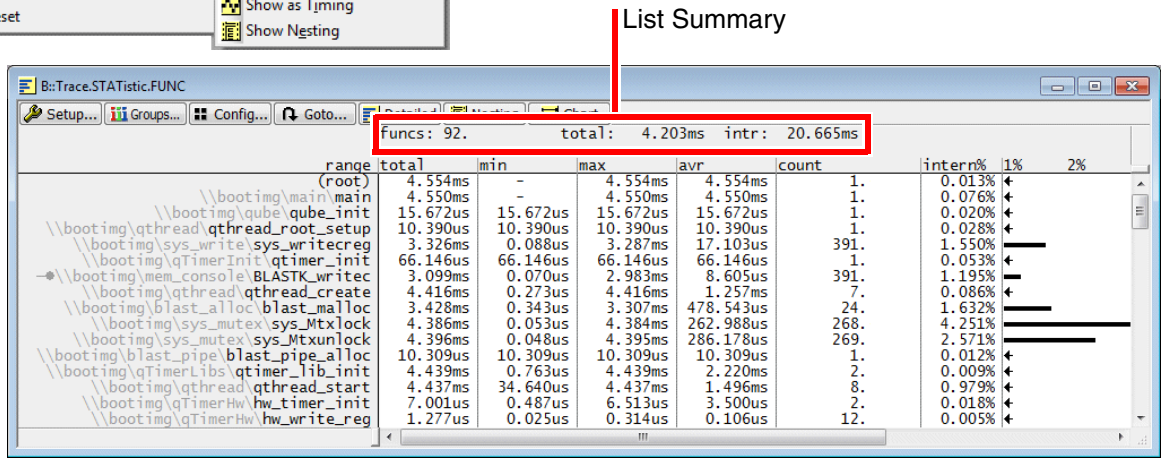
- **Exits of TRAP handlers**

  A RETURN / RETURN FROM INTERRUPT within the HLL trap handler is regarded as exit of the TRAP handler.

Numerical Analysis

Trace.STATistic.Func [**/MergeCORE**]	Numerical display of nested function run-time analysis • analysis for all hardware threads
Trace.STATistic.sYmbol /CORE *<n>*	Numerical display of function timing • analysis for specified hardware thread

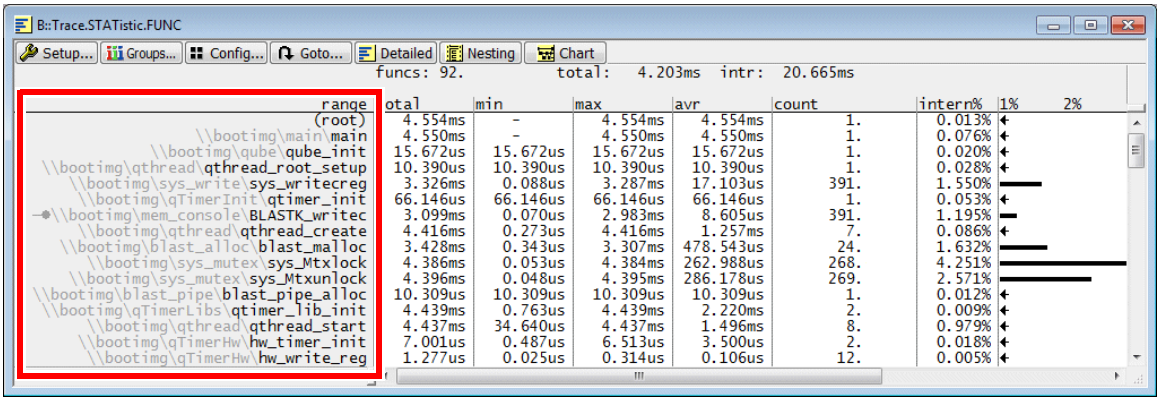





```
funcs: 92.          total:   4.203ms  intr:  20.665ms
```

For a description of the list summary, see table below.

List Summary	
func	Number of functions in the trace
total	Total measurement time
intr	Total time in interrupt service routines



For a description of the highlighted column, see table below.

Columns	Description
range (NAME)	Function name, sorted by their occurrence by default

- **(root)**

  (root)

  The function nesting is regarded as tree, root is the root of the function nesting.

- **HLL function**

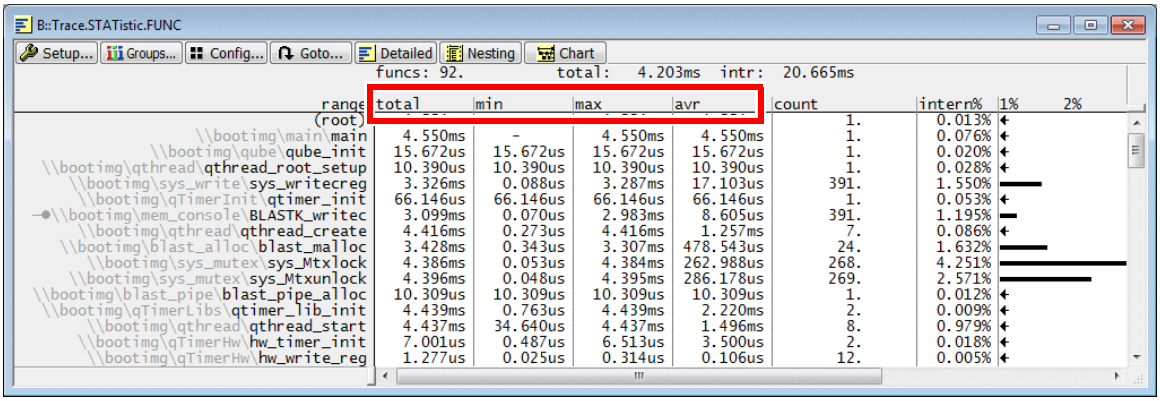
  \\bootimg\blast_alloc\blast_malloc

- **HLL interrupt service routine**

  →\\bootimg\Global\BLASTK_handle_int

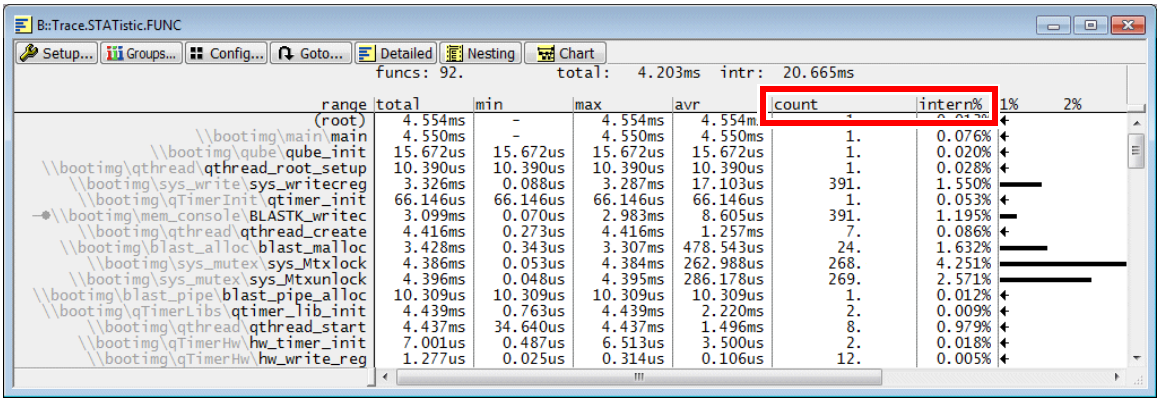
- **HLL trap handler**

  –◆BLASTK_handle_trap0



For a description of the highlighted columns, see below.

Columns *(cont.)*	**Description**
total	Total time within the function
min	Shortest time between function entry and exit, time spent in interrupt service routines is excluded. No **min** time is displayed if a function exit was never executed.
max	Longest time between function entry and exit, time spent in interrupt service routines is excluded.
avr	Average time between function entry and exit, time spent in interrupt service routines is excluded.

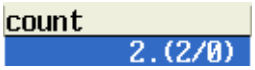


For a description of the highlighted columns, see below.

Columns *(cont.)*	Description
count	Times within the function

If function entries or exits are missing, this is displayed in the following format:

<times within the function >. (*<number of missing function entries>*/*<number of missing function exits>*).



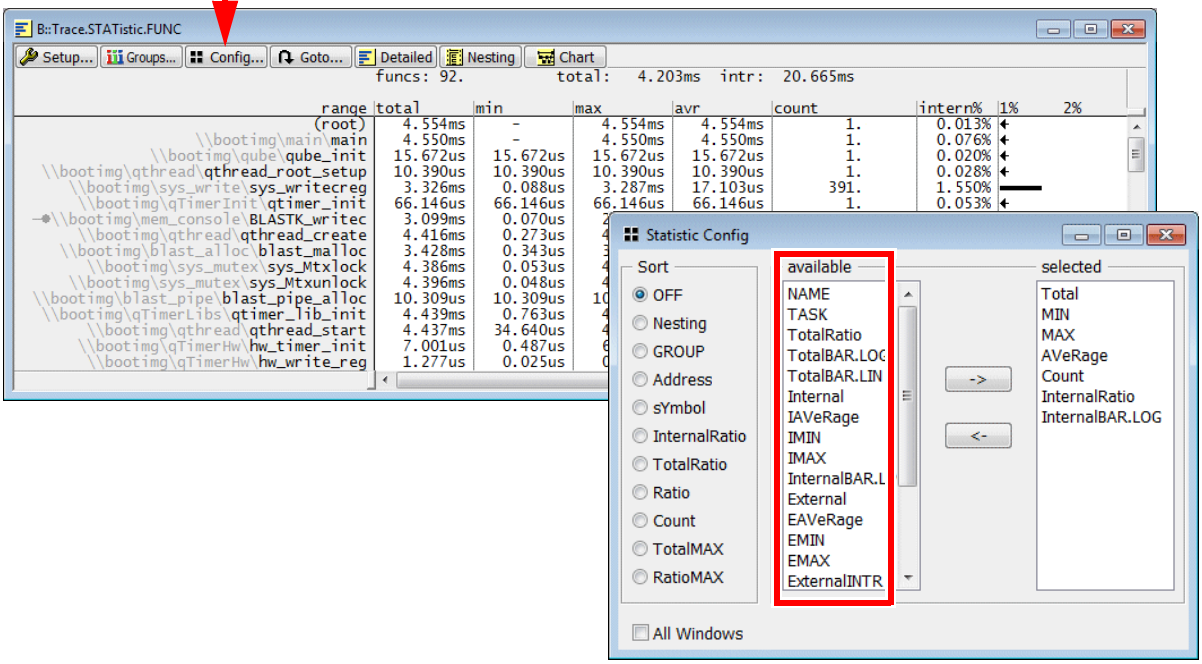
Interpretation examples:

1. 2. (2/0): 2. times within the function, 2 function entries missing
2. 4. (0/3): 4. times within the function, 3 function exits missing
3. 11. (1/1): 11. times within the function, 1 function entry and 1 function exit is missing.

If the number of missing function entries or exits is higher the 1. the analysis performed by the command **Trace.STATistic.Func** might fail due to nesting problems. A detailed view to the trace contents is recommended.

Columns *(cont.)*	Description
intern% (InternalRatio, InternalBAR.LOG)	Ratio of time within the function without subfunctions, TRAP handlers, interrupts

Pushing the **Config…** button allows to display additional columns.



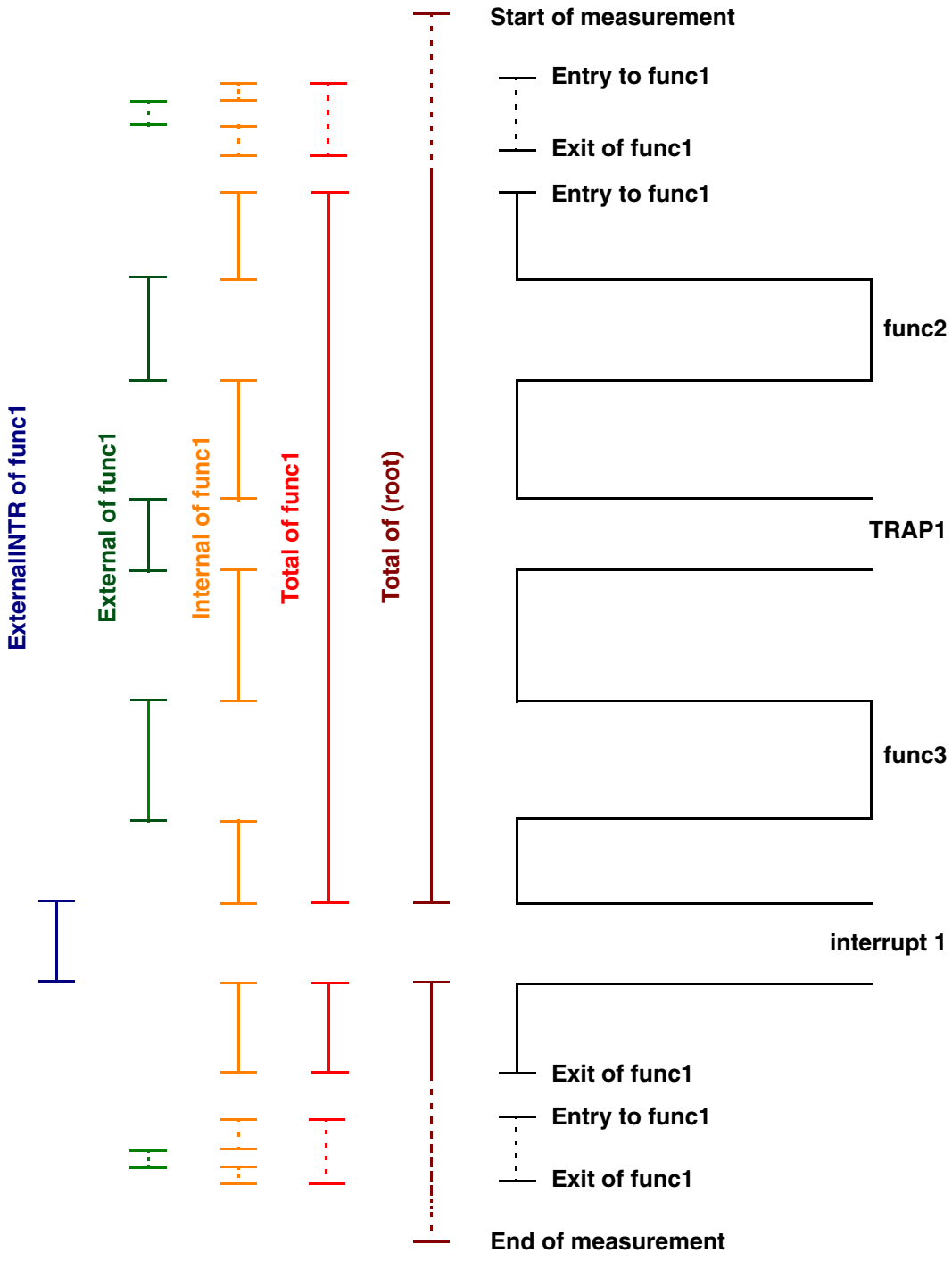
For a description of the additional columns, see tables below.

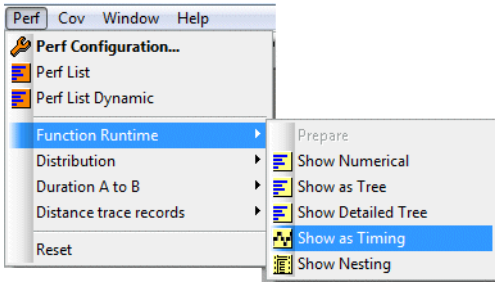
Columns *(cont.)* - times only in function	
Internal	Total time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IAVeRage	Average time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IMIN	Shortest time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IMAX	Longest time spent in the function between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
InternalRatio	*<Internal time of function>/<Total measurement time>* as a numeric value.
InternalBAR	*<Internal time of function>/<Total measurement time>* graphically.

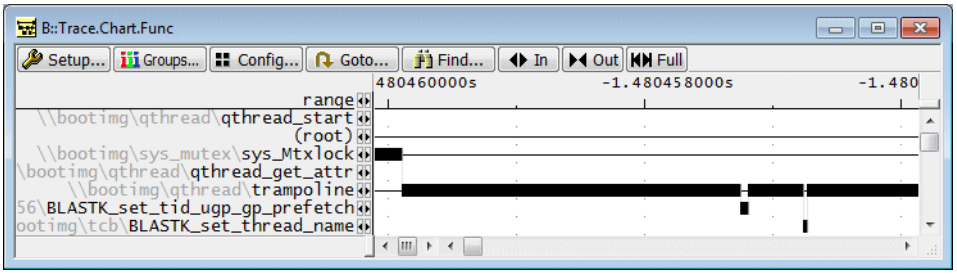
Columns *(cont.)* - times in sub-functions and TRAP handlers	
External	Total time spent within called sub-functions/TRAP handlers
EAVeRage	Average time spent within called sub-functions/TRAP handlers
EMIN	Shortest time spent within called sub-functions/TRAP handlers
EMAX	Longest time spent within called sub-functions/TRAP handlers

Columns *(cont.)* - interrupt times	
INTR	Total time the function was interrupted
ExternalINTRMAX	Max. time one function pass was interrupted
ExternalINTRCount	Number of interrupts that occurred during the function run-time

The following graphic give an overview how times are calculated:

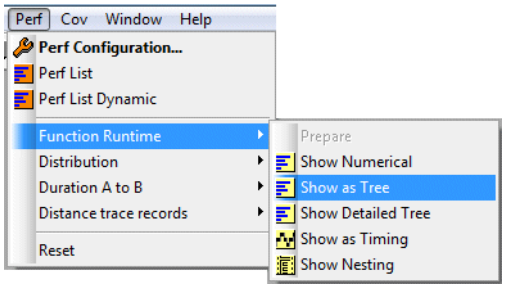


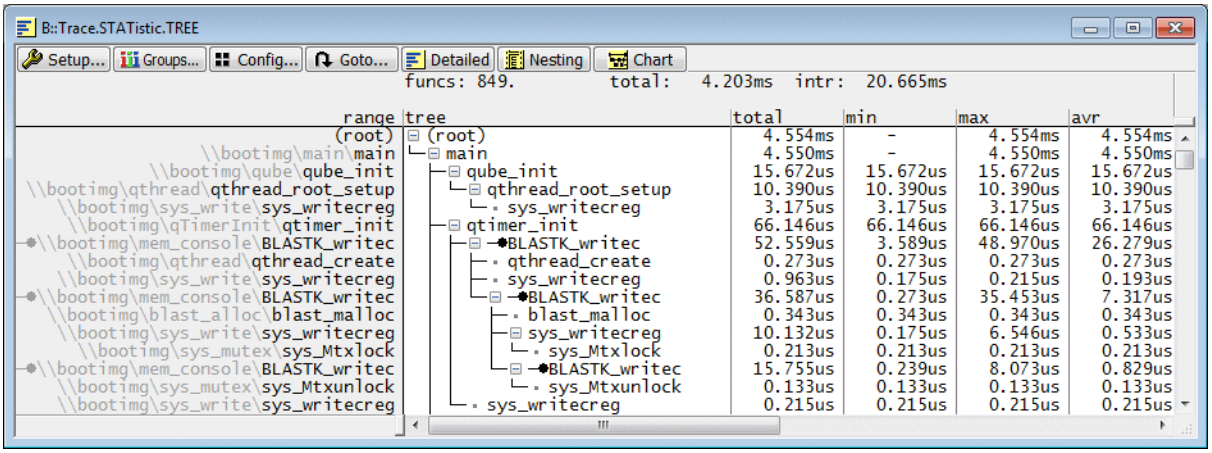




Legend	
solid black bar	Function running
thin black line	Subfunction or TRAP handler running

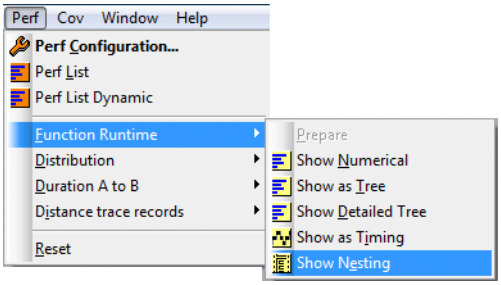
Trace.Chart.Func [**/MergeCORE**]	Graphical display of nested function run-time analysis • Analysis for all hardware threads
Trace.Chart.Func /CORE *<n>*	Graphical display of nested function run-time analysis • Analysis for specified hardware thread

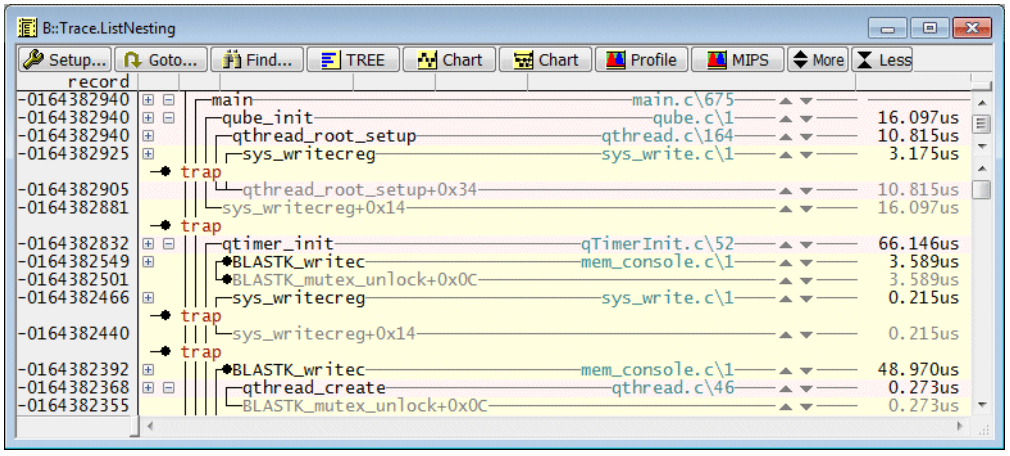




Trace.STATistic.TREE [**/MergeCORE**] Tree display of nested function run-time analysis
- Analysis for all hardware threads

Trace.STATistic.TREE /CORE *<n>* Tree display of nested function run-time analysis
- Analysis for specified hardware thread





Trace.ListNesting [**/MergeCORE**]	Nesting display of nested function run-time analysis • Analysis for all hardware threads

Cycle Statistic

To perform a cycle statistic proceed as follows:

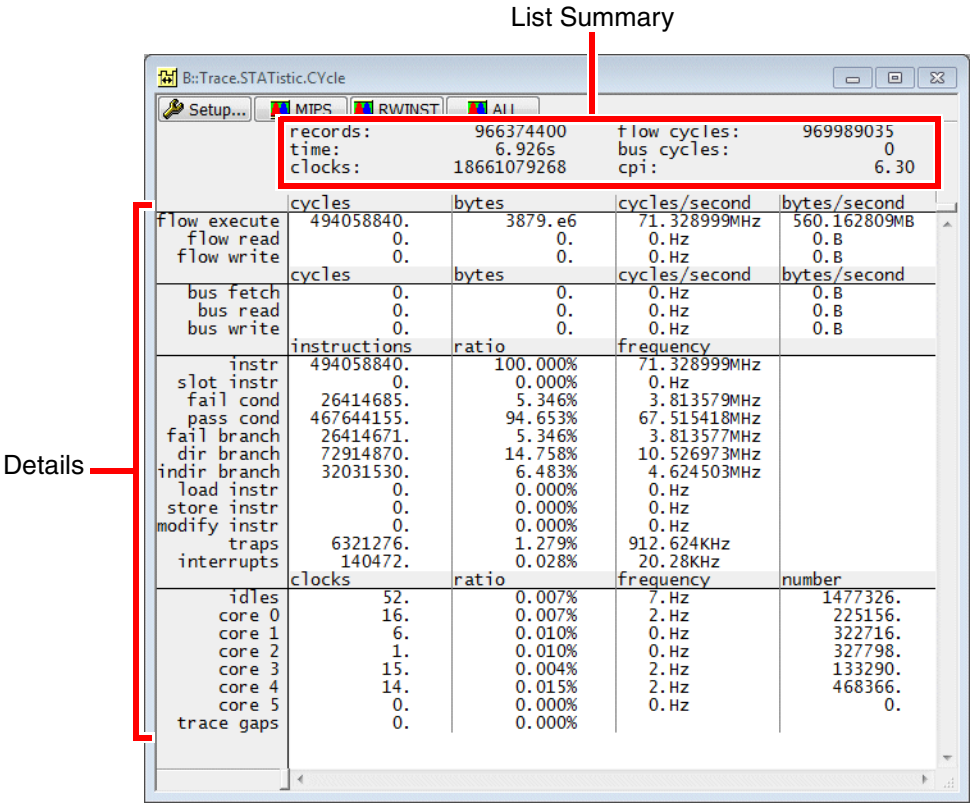
1. **Activate cycle-accurate tracing.**

```
ETM.CycleAccurate ON

Trace.CLOCK 600.MHZ
```

2. **Start and stop the program execution to fill the trace repository.**

3. **Display the result.**

```
Trace.STATistic.CYcle
```



For a description of the list summary and the details, see tables below.

List Summary	Description
records	Number of records in the trace
time	Time period recorded by the trace

List Summary	Description
clocks	Number of clock cycles in the trace
flow cycles	Number of ptrace packages
bus cycles	0 (no recording of bus cycles)
cpi	Average clocks per instruction packet (cpi/6 average thread clock per instruction packet)

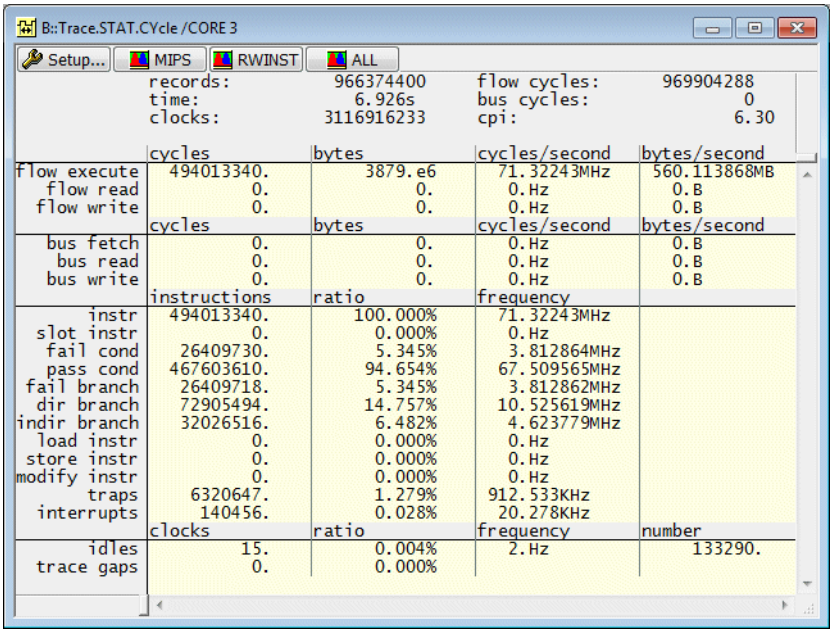
Details	Description
flow execute	Number of cycles that executed instructions
flow read	Number of cycles that performed a read access (not implemented yet)
flow write	Number of cycles that performed a write access (not implemented yet)
bus fetch	0 (no recording of bus cycles)
bus read	0 (no recording of bus cycles)
bus write	0 (no recording of bus cycles)
instr	number of instruction packages
slot instr	—
fail cond	Number of conditional instruction that failed (failed branch instructions included)
pass cond	Number of conditional instruction that passed (branch taken included)
fail branch	Number of failed branches
dir branch	Number of direct branches
indir branch	Number of indirect branches
load instr	Number of load instructions (not implemented yet)
store instr	Number of store instructions (not implemented yet)
modify instr	—

Details	Description
traps	Number of traps
interrupts	Number of interrupts
idles	Number of idle states • Wait instruction, under the assumption that the hardware thread put itself to idle state • More the 1000. clock cycles without trace information
core 0	Number of idle states for hardware thread 0
…	
trace gaps	Number of trace gaps (FIFOFULLs, filtered trace information …)

Trace.STATistic.CYcle [**/MergeCORE**] — Cycle statistic
- Analysis for all hardware threads

Trace.STATistic.CYcle /CORE *<n>* — Cycle statistic
- Analysis for specified hardware thread

```
Analyzer.STATistic.CYcle /CORE 3
```

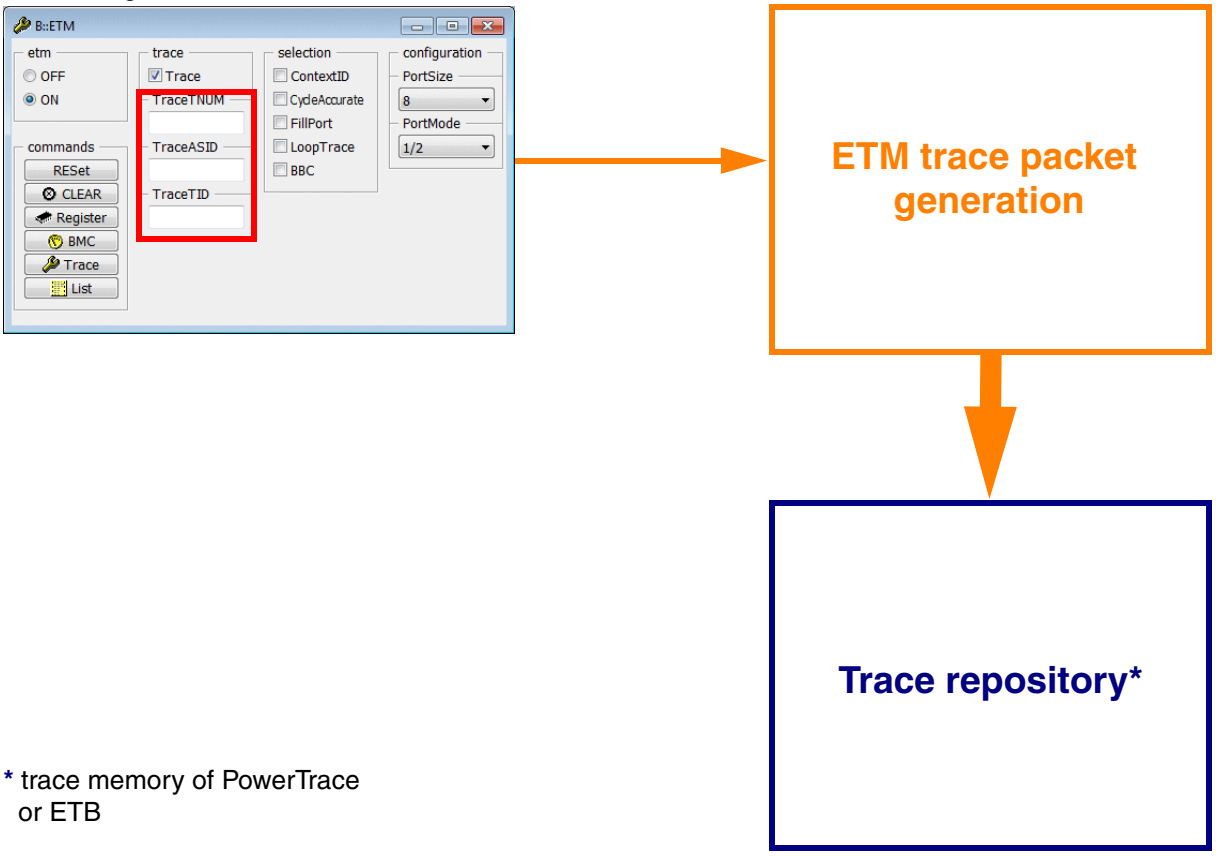


Filtering via the ETM Configuration Window

Filtering means to reduce the generated trace information to the information of interest.

Some basic filtering can be done via the ETM configuration window.

ETM configuration



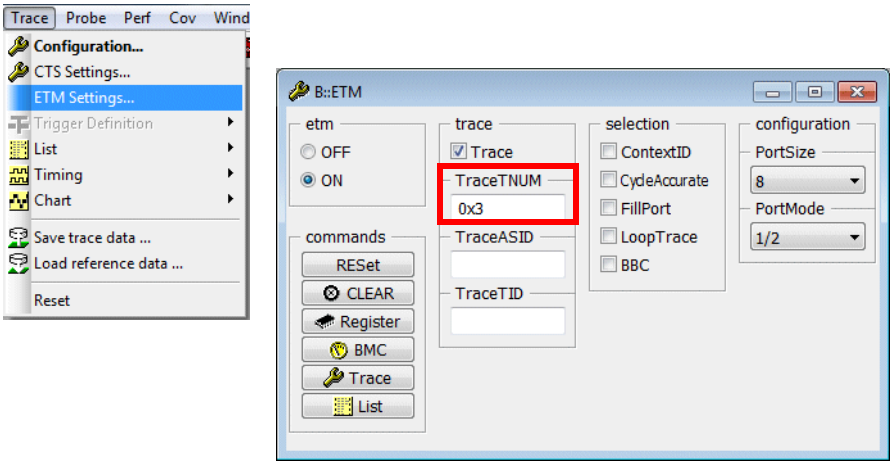
* trace memory of PowerTrace or ETB

The following setups in the ETM configuration window can be done to reduce the generation of the trace information:

ETM.state	Display the ETM configuration window
ETM.TraceTNUM *<hardware_thread>*	Program the ETM to export the instruction flow only for the specified *<hardware_thread>*
ETM.TraceASID *<asid>*	Program the ETM to export the instruction flow only for the specified *<asid>*
ETM.TraceTID *<tid_number>* | *<bitmask>*	Program the ETM to export the instruction flow only for the specified software thread(s)

To restrict the exported instruction flow to the specified hardware thread proceed as follows:

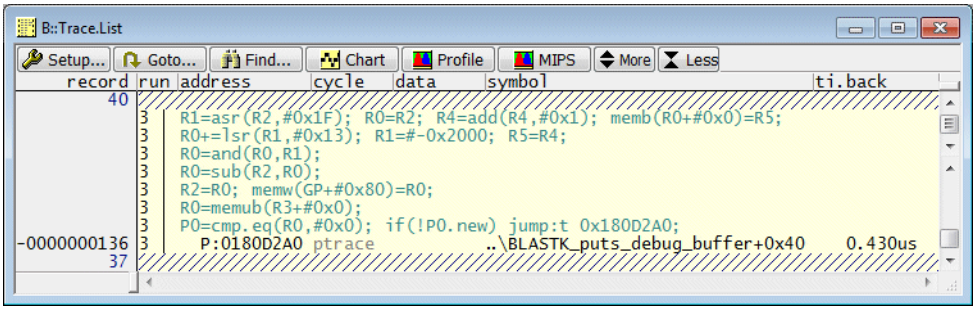
1. **Open the ETM configuration window and specify the hardware thread.**



2. **Start and stop the program execution.**

3. **Display the result.**

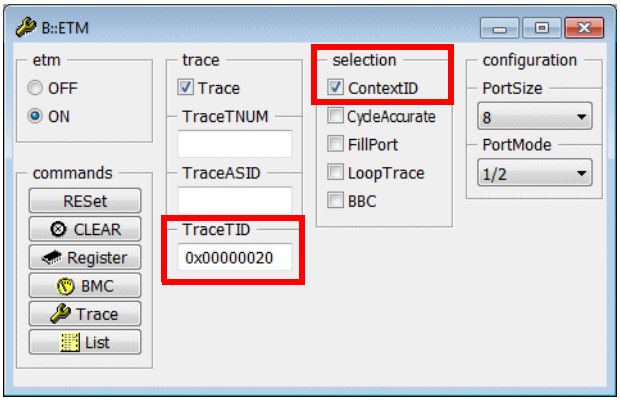
```
Trace.List
```



Software Thread Filter

To restrict the exported instruction flow to the specified software thread proceed as follows:

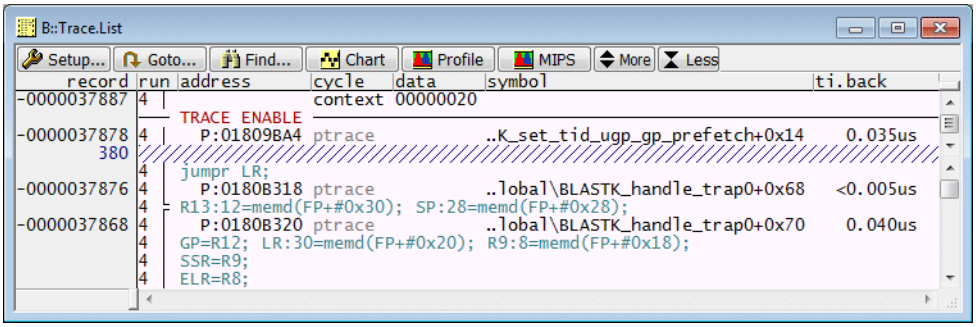
1. **Open the ETM configuration window and specify the software thread.**



2. **Start and stop the program execution.**

3. **Display the result.**

```
Trace.List
```



ASID Filter

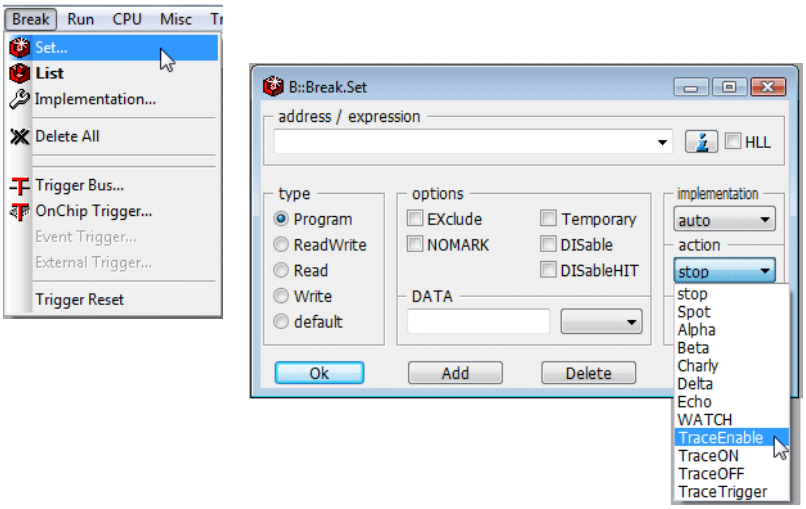
(no example available)

Filtering/Triggering with Break.Set

Filtering means to reduce the generation of trace information to the information of interest.

Filtering helps to prevent TARGET FIFO OVERFLOWs and enables a more effective utilization of the trace memory.

Triggering means to stop the recording to the trace repository.



The following actions provide filters:

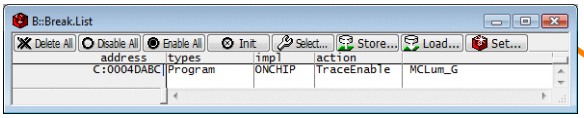
TraceEnable	Program the ETM to generate only trace information if the specified event matches.
TraceON	Program the ETM to start the generation of trace information if the specified event matches.
TraceOFF	Program the ETM to stop the generation of trace information if the specified event matches (restart possible).

The following action provides triggers:

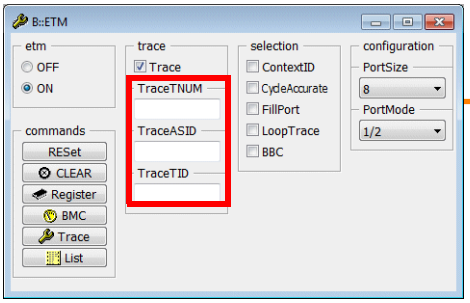
TraceTrigger	Stop the recording of trace information into the trace repository if the specified event matches (no restart possible). The stop can be delayed.

The filter/trigger breakpoints and the filters provided by the ETM configuration window can be combined.

Filter breakpoints

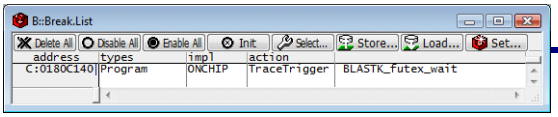


ETM configuration



ETM trace packet generation

Trigger breakpoints



Trace repository*

* trace memory of PowerTrace or ETB

Standard Usage

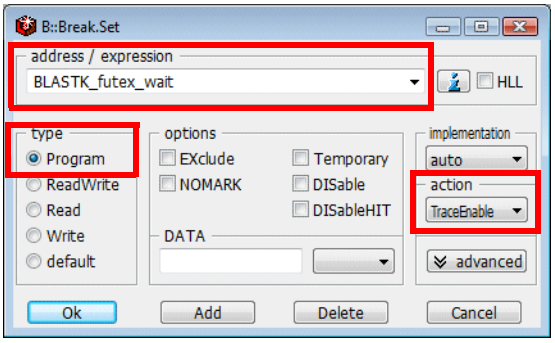
To illustrate the standard usage of the TraceEnable filter, the following examples are provided:

- Example 1: Program the ETM to export only trace information, if the instruction at a particular symbolic address is executed.
- Example 2: Program the ETM to export only trace information, if the instruction at a particular symbolic address is executed by a particular hardware thread.
- Example 3: Program the ETM to export only information about the instruction that writes to a particular variable.

Example 1

Program the ETM to export only trace information, if the instruction at the symbolic address *BLASTK_futex_wait* is executed (etm_filter1.cmm).

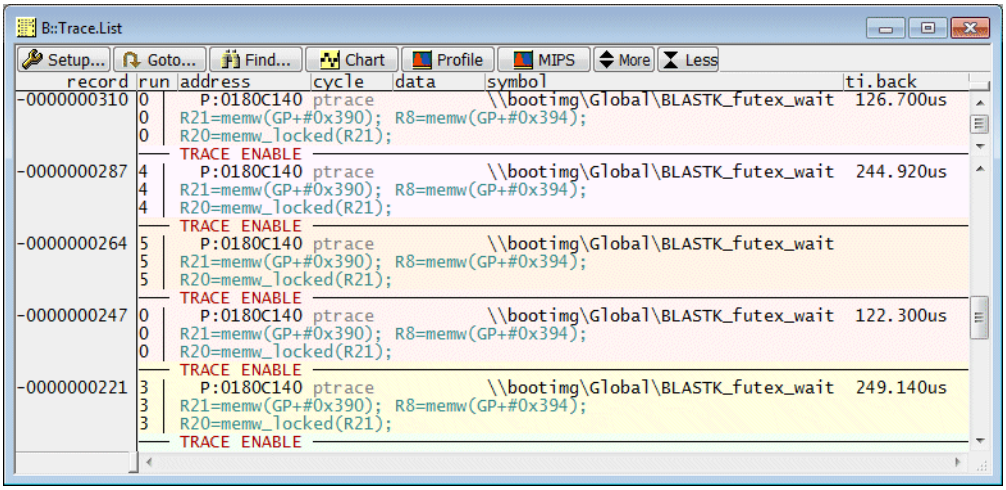
1. **Specify the event in the Break.Set dialog.**



   - Specify the program address in the **address / expression** field.
   - Specify the **type** Program (default).
   - Specify the **action** TraceEnable.

2. **Start and stop the program execution.**

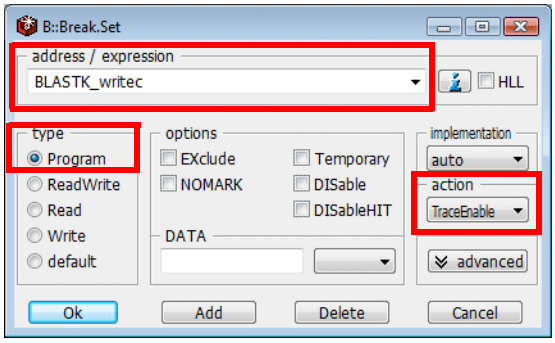
3. **Display the result.**


B::Trace.List
Setup...
Goto...
Find...
Chart
Profile
MIPS
More
Less
record run address cycle data symbol ti.back
-0000000310 0 P:0180C140 ptrace \\bootimg\Global\BLASTK_futex_wait 126.700us
0 R21=memw(GP+#0x390); R8=memw(GP+#0x394);
0 R20=memw_locked(R21);
TRACE ENABLE
-0000000287 4 P:0180C140 ptrace \\bootimg\Global\BLASTK_futex_wait 244.920us
4 R21=memw(GP+#0x390); R8=memw(GP+#0x394);
4 R20=memw_locked(R21);
TRACE ENABLE
-0000000264 5 P:0180C140 ptrace \\bootimg\Global\BLASTK_futex_wait
5 R21=memw(GP+#0x390); R8=memw(GP+#0x394);
5 R20=memw_locked(R21);
TRACE ENABLE
-0000000247 0 P:0180C140 ptrace \\bootimg\Global\BLASTK_futex_wait 122.300us
0 R21=memw(GP+#0x390); R8=memw(GP+#0x394);
0 R20=memw_locked(R21);
TRACE ENABLE
-0000000221 3 P:0180C140 ptrace \\bootimg\Global\BLASTK_futex_wait 249.140us
3 R21=memw(GP+#0x390); R8=memw(GP+#0x394);
3 R20=memw_locked(R21);
TRACE ENABLE

Example 2

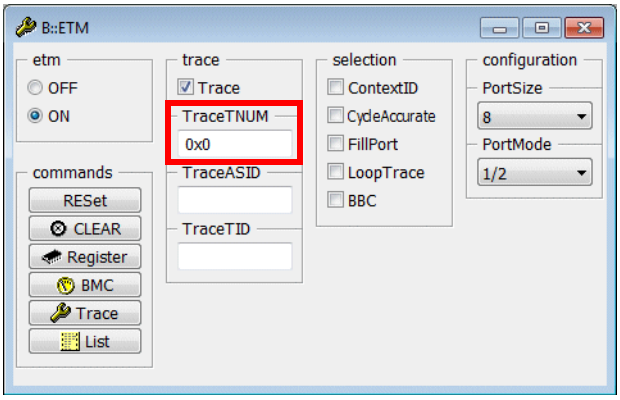
Program the ETM to export only trace information, if the instruction at the symbolic address *BLASTK_writec* is executed by hardware thread 0x0 (etm_filter2.cmm).

1. **Specify the event in the Break.Set dialog.**



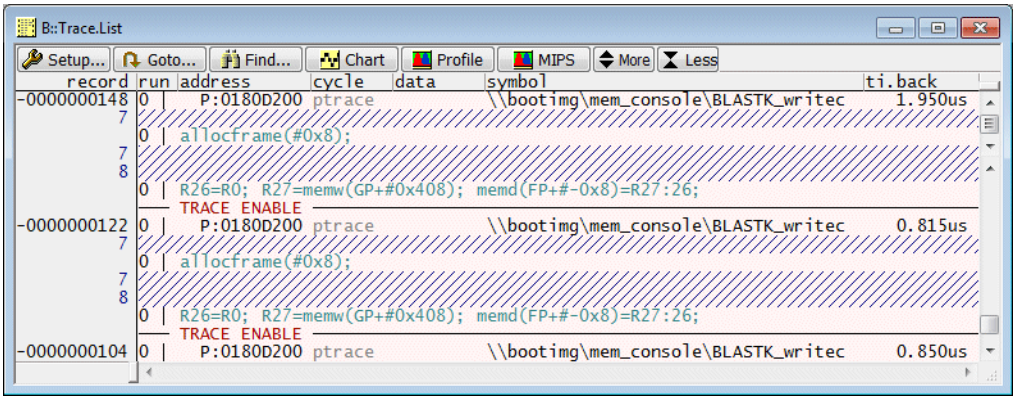
- Specify the program address in the **address / expression** field.
- Specify the **type** Program (default).
- Specify the **action** TraceEnable.

2. **Specify hardware thread 0x0 in the ETM configuration window.**



3. **Start and stop the program execution.**

4. Display the result.



Summary

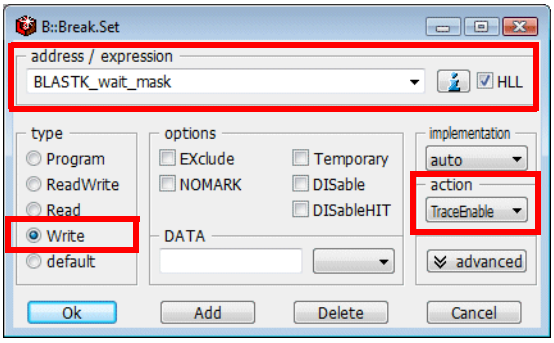
; Export only the execution of the specified instruction packets
; (up to 8 single instructions or up to 4 instruction ranges)

Break.Set *<address>* | *<range>* **/Program /TraceEnable**

Example 3

Program the ETM to export only information about the instruction that writes to the variable *BLASTK_wait_mask* (etm_filter3.cmm).

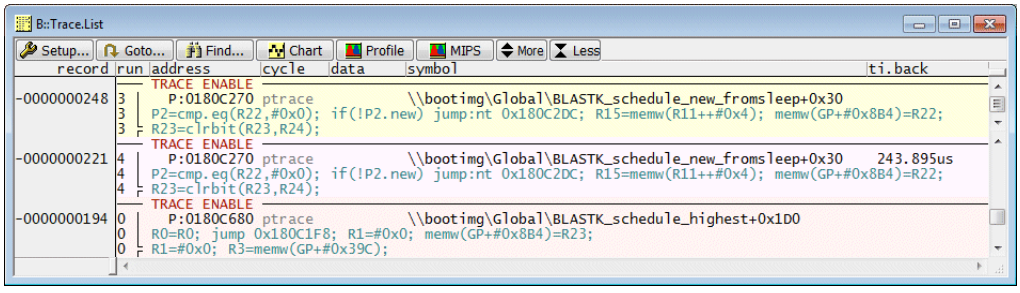
1. Specify the event in the Break.Set dialog.



- Specify the data address in the **address / expression** field. Activate the **HLL** check box to specify the breakpoint for the complete address range of the variable.
- Specify the **type** Write.
- Specify the **action** TraceEnable.

2. Start and stop the program execution

3. Display the result.



```
; Export only the instructions that perform the specified data access
; no data value allowed
; (up to 6 single address accesses or up to 3 access ranges)

Break.Set <address> | <range> /ReadWrite | /Read | /Write /TraceEnable
Var.Break.Set <hll_expression> /ReadWrite | /Read | /Write /TraceEnable
```

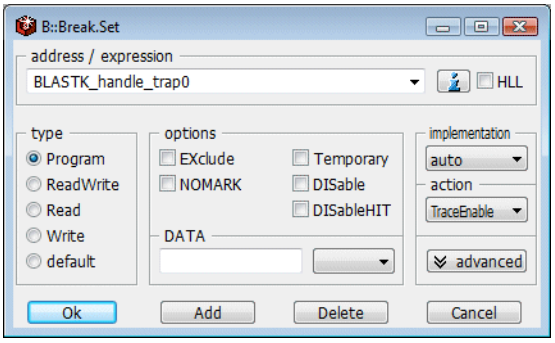
To illustrate statistical evaluations, the following examples are provided:

- Example 1: Analyze the intervals of a particular function.
- Example 2: Analyze the time between function A and function B.

Example 1: Time Interval of a Single Event

Analyze the intervals of *BLASTK_handle_trap0*.

1. Program the ETM to export only the entry to the function *BLASTK_handle_trap0*.

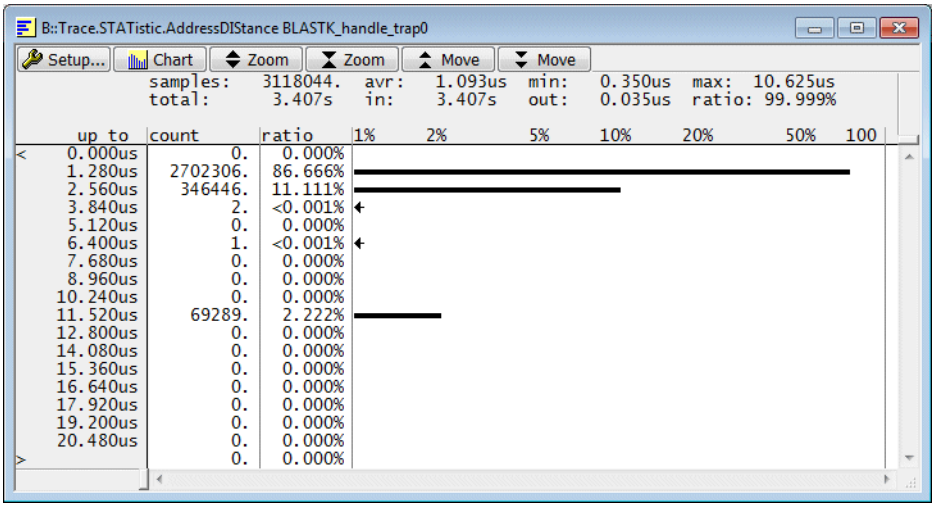


- Specify the program address in the **address / expression** field.
- Specify the **type** Program (default).
- Specify the **action** TraceEnable.

2. Start and stop the program execution.

3. Display the result.

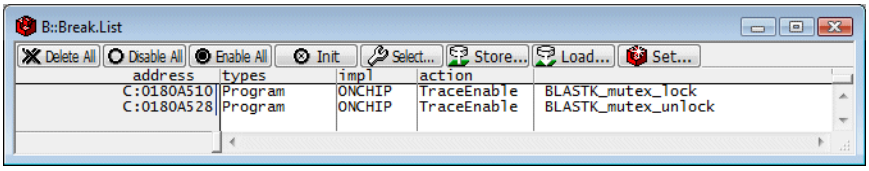
```
Trace.List
Trace.STATistic.AddressDIStance BLASTK_handle_trap0
```



Example 2: Time between Two Events

Analyze the time between *BLASTK_mutex_lock* and *BLASTK_mutex_unlock.*

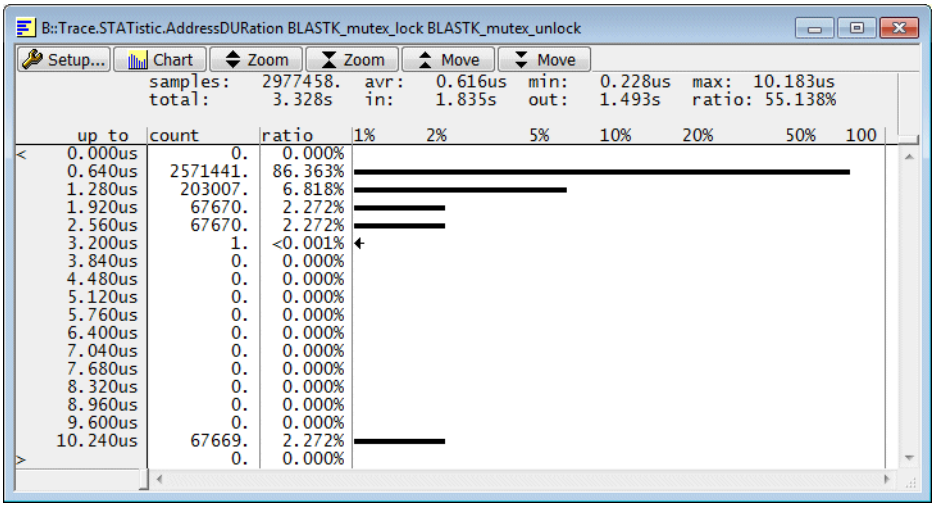
1. **Program the ETM to export only the entry to the functions *BLASTK_mutex_lock* and *BLASTK_mutex_unlock*.**



2. **Start and stop the program execution.**

3. **Display the result.**

```
Trace.List

Trace.STATistic.AddressDURation BLASTK_mutex_lock \
BLASTK_mutex_unlock
```

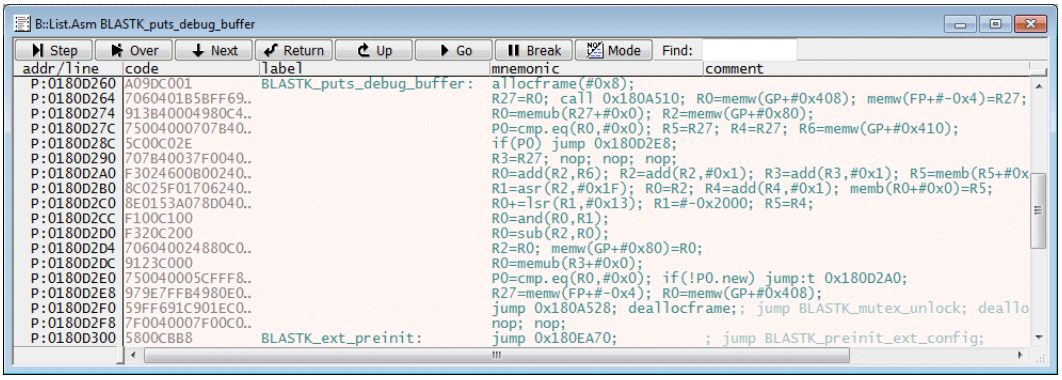


To illustrate the TraceON/OFF filter, the following example is provided:

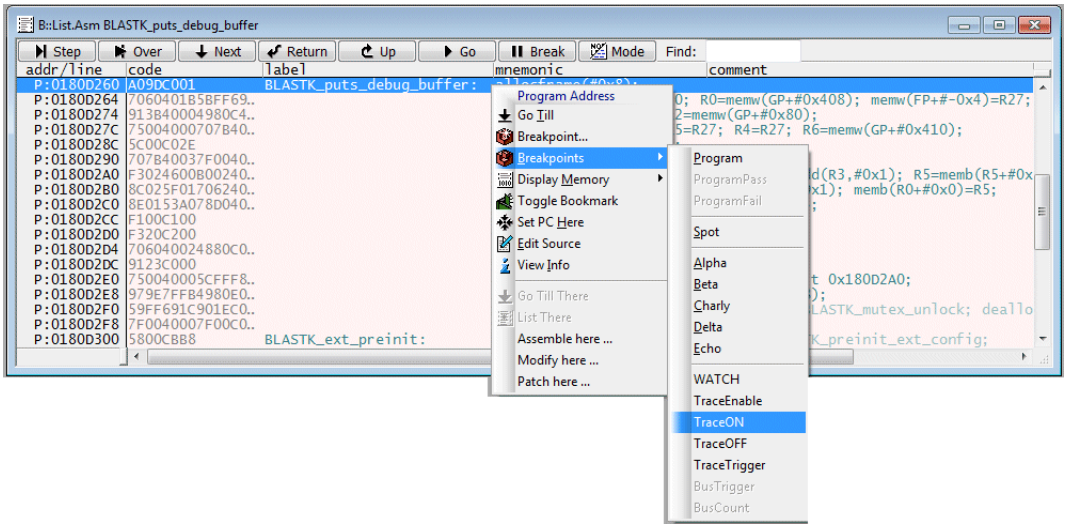
- Program the ETM to start the exporting of trace information, whenever the instruction at the address *BLASTK_puts_debug_buffer* was executed.
- Program the ETM to stop the exporting of trace information, whenever the instruction at the address *BLASTK_puts_debug_buffer+0x90* was executed (etm_filter4.cmm).

1. **Open a source listing at the label** *BLASTK_puts_debug_buffer.*

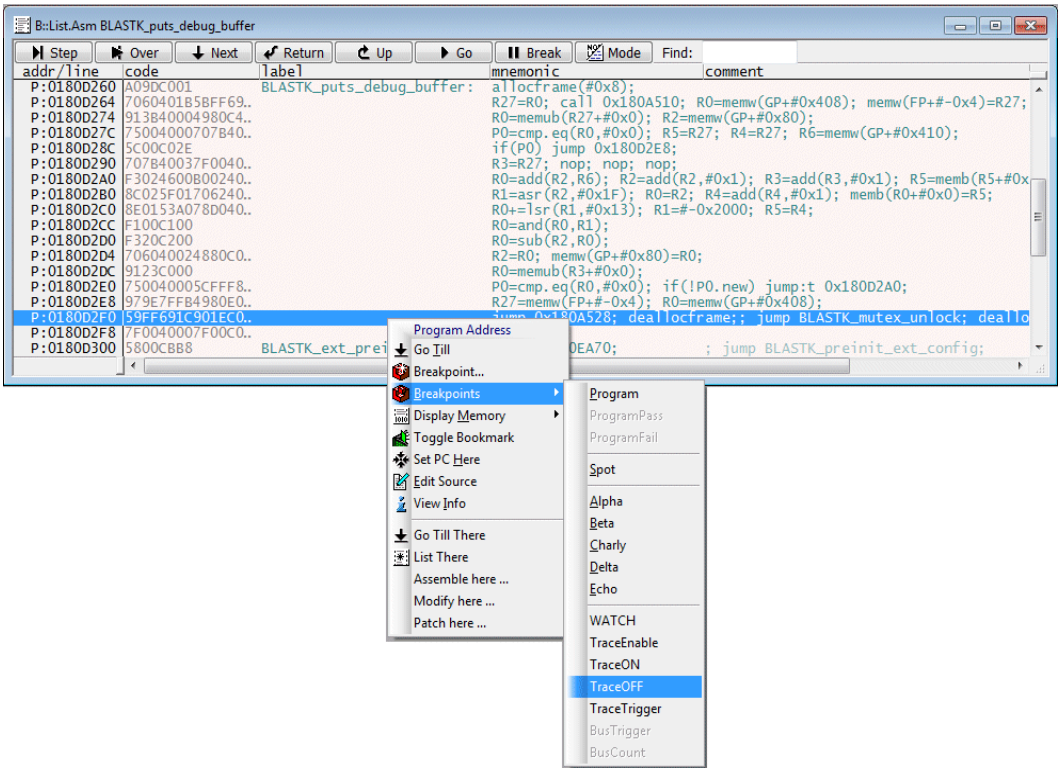
```
; List *
List.Asm BLASTK_puts_debug_buffer
```



2. **Set a TraceON breakpoint to the instruction packet at the label *BLASTK_puts_debug_buffer*.**



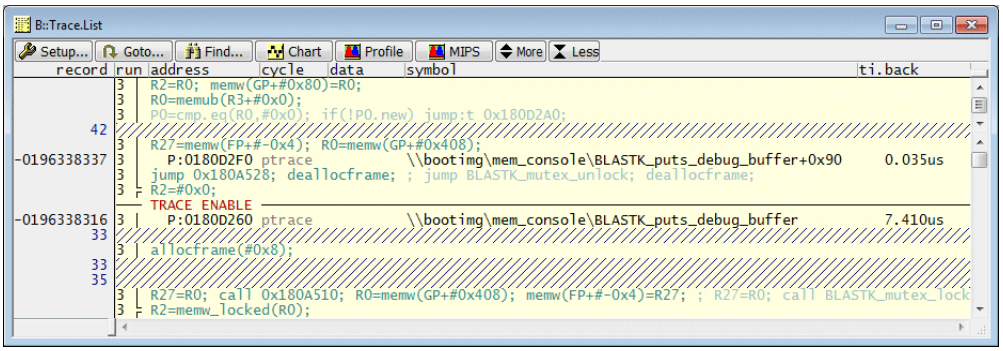
3. **Set a TraceOFF breakpoint to the instruction packet at the address *BLASTK_puts_debug_buffer+90*.**



4. **Start and stop the program execution.**

5. Display the result.

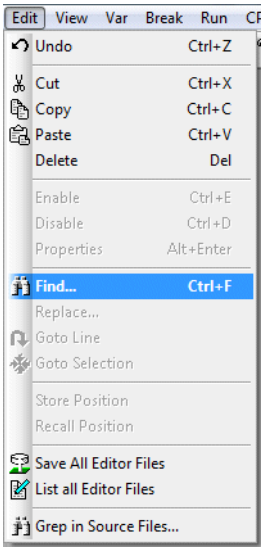
```
Trace.List
```

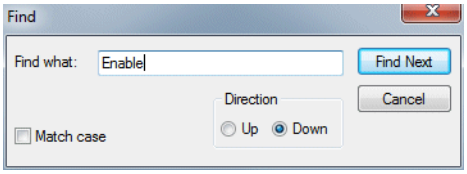


Proceed as follows, if you want to search for the ON/OFF transitions:

1. Select the Trace.List window as active window.

2. Specify *Enable* for the global TRACE32 Find.





```
; Export only the execution of the instructions between TraceON/TraceOFF
; (up to 2 pairs)

Break.Set <address> | <range>  /Program /TraceON
Break.Set <address> | <range> /ReadWrite | /Read | /Write /TraceON
Var.Break.Set <hll_expression> /ReadWrite | /Read | /Write /TraceON

Break.Set <address> | <range>  /Program /TraceOFF
Break.Set <address> | <range> /ReadWrite | /Read | /Write /TraceOFF
Var.Break.Set <hll_expression> /ReadWrite | /Read | /Write /TraceOFF
```

There are two use cases for TraceTrigger.

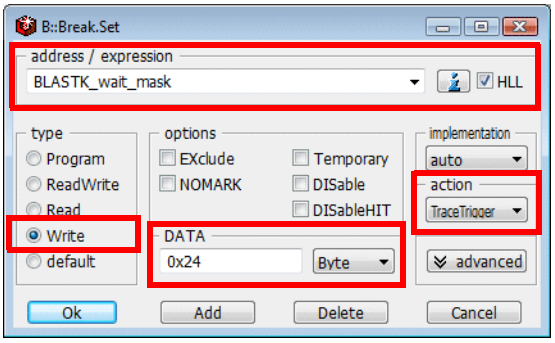
To illustrate the two use cases, the following examples are provided:

- Example 1: A TraceTrigger can be used instead of a breakpoint, if it is not allowed to stop the program execution.
- Example 2: A TraceTrigger can be used to get the prologue and the epilog of an event in the trace.

Example 1

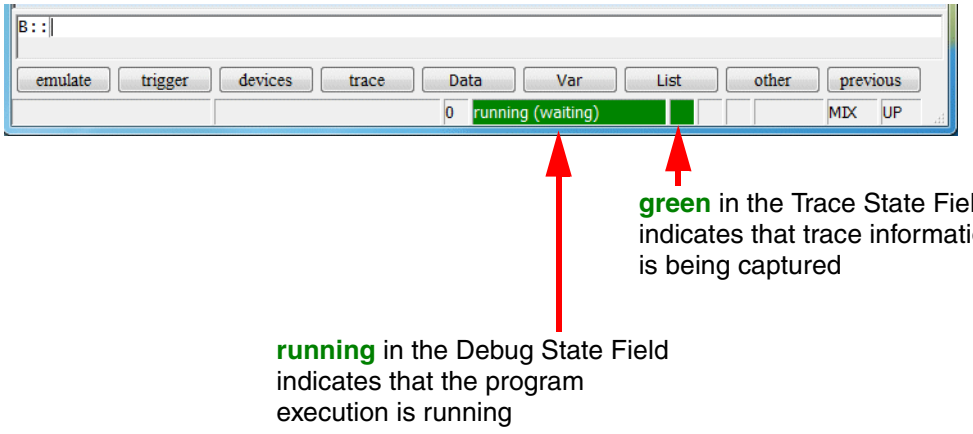
Stop the trace recording after 0x24 was written as a byte to the variable *BLASTK_wait_mask* (etm_trigger1.cmm).

1. Specify the event in the Break.Set dialog.



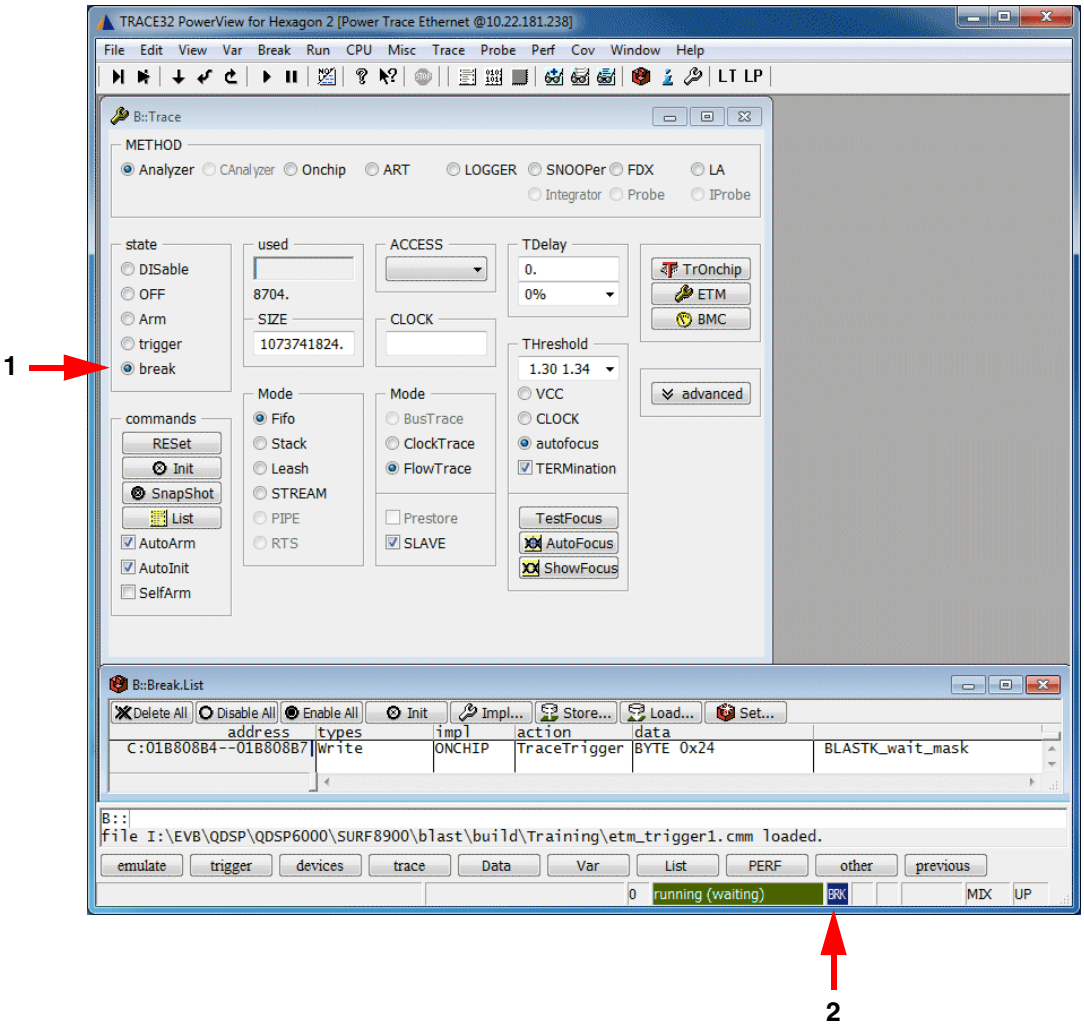
- Specify the data address in the **address / expression** field. Activate the **HLL** check box to specify the breakpoint for the complete address range of the variable.
- Specify the **type** Write.
- Specify **DATA** value and access width.
- Specify the **action** TraceTrigger.

2. Start the program execution.

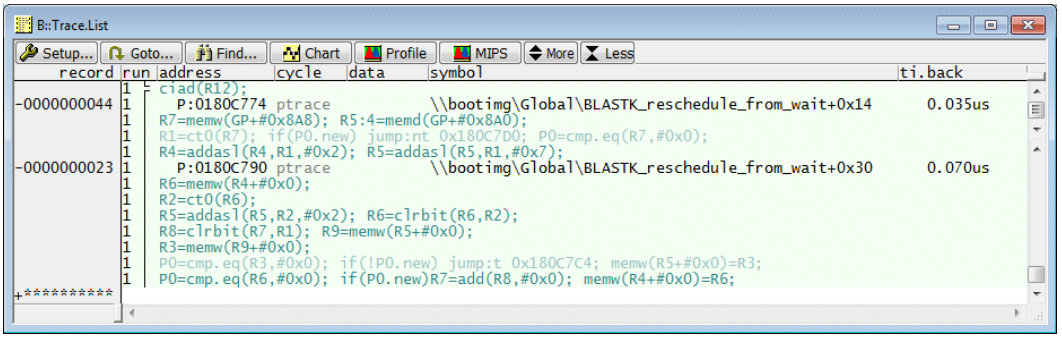


3. The recording to the trace repository is stopped soon after the event happened.

- The **state** field in the Trace Configuration window changes to **break** (1) to indicate that the recording to the trace repository is stopped.
- The Trace State field in the TRACE32 State Line changes to **BRK** accordingly (2).



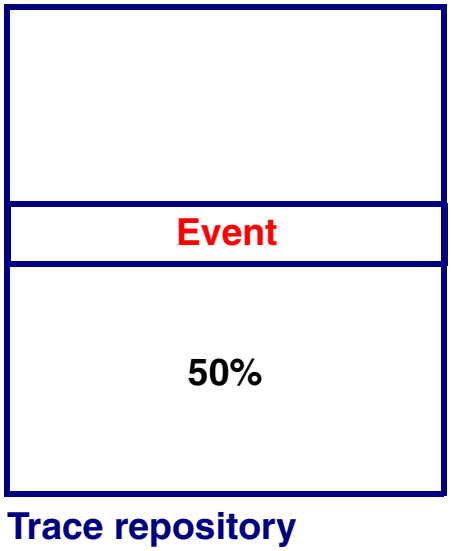
4. Display the result.



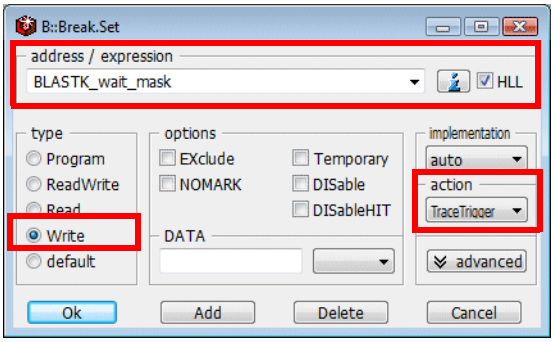
Please be aware that the result can only be displayed while the program execution is running if the program code was copied into the TRACE32 Virtual Memory before.

Example 2

Stop the trace recording when a write access to the variable *BLASTK_wait_mask* occurred and another 50% of the trace repository was filled.

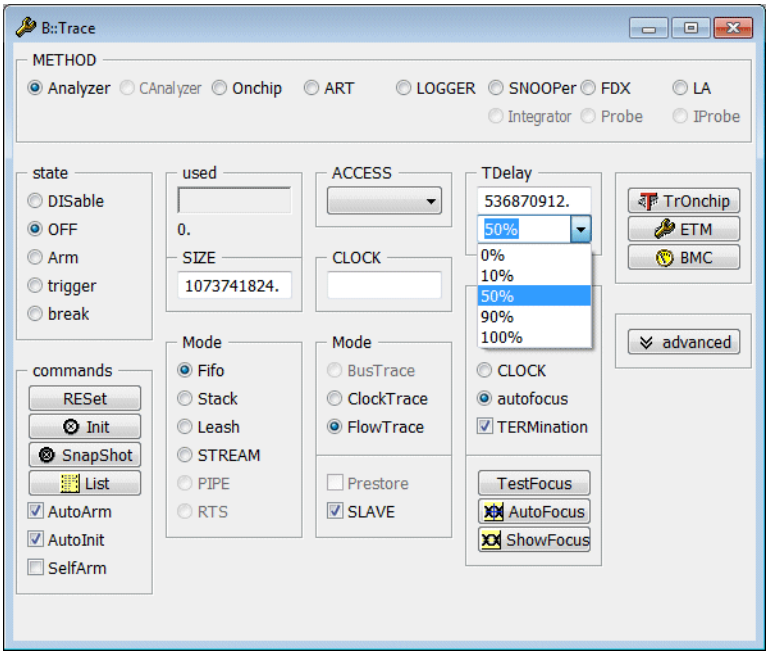


1. **Specify the event in the Break.Set dialog.**



- Specify the data address in the **address / expression** field. Activate the **HLL** check box to specify the breakpoint for the complete address range of the variable.
- Specify the **type** Write.
- Specify the **action** TraceTrigger.

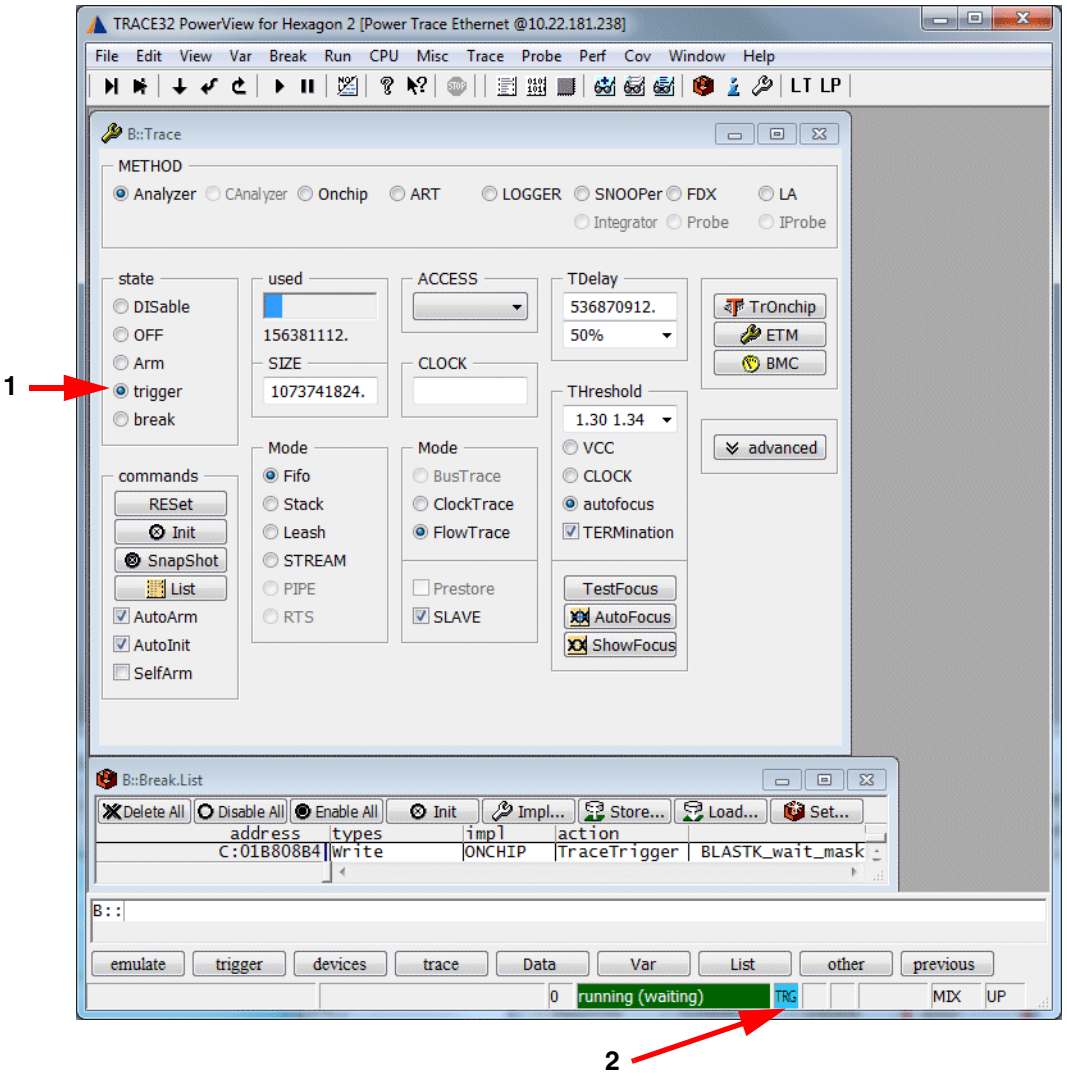
2. **Specify the fill of the trace repository after the event (TDelay counter).**



3. **Start the program execution.**

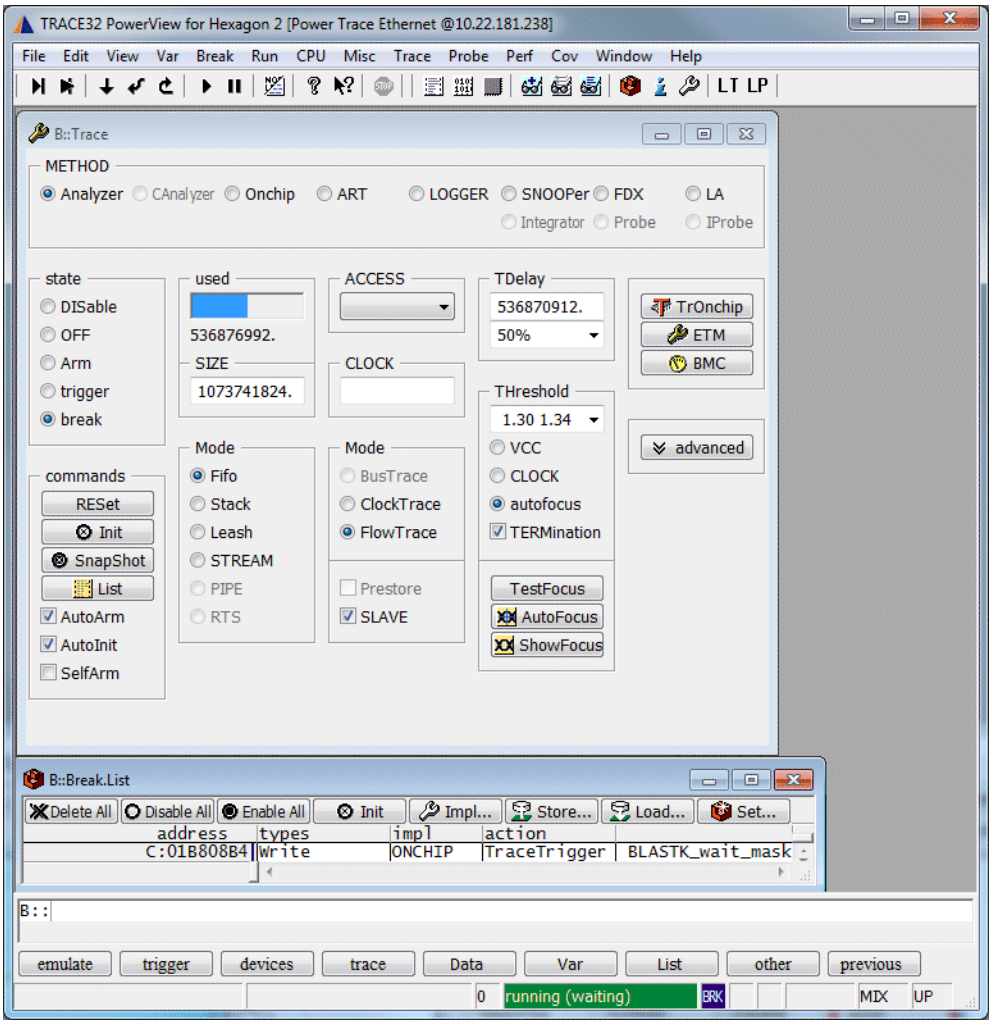
4. As soon as the event occurred

- The **state** field in the Trace Configuration window changes to **trigger** (1).
- The Trace State Field in the TRACE32 State Line changes to **TRG** accordingly (2).



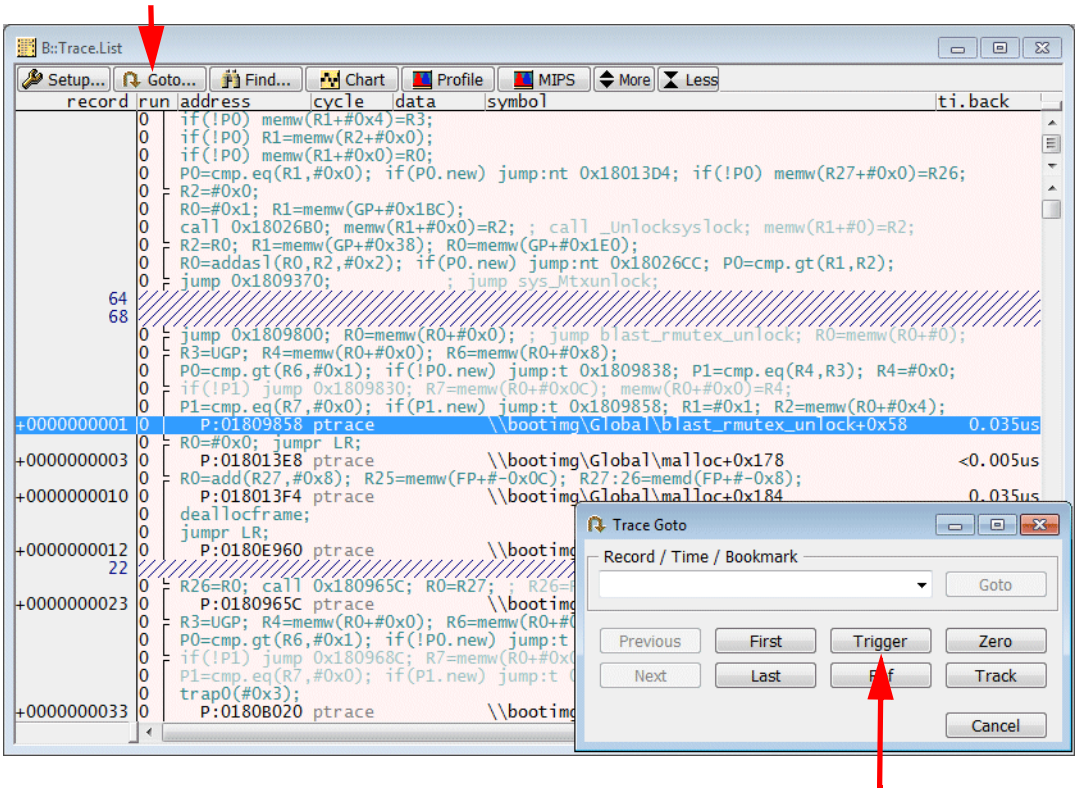
5. **As soon as the TDelay counter ran down**

- - The **state** field in the Trace Configuration window changes to **break**.
- - The Trace State field in the TRACE32 State Line changes to **BRK** accordingly.



6. After the TDelay counter elapsed the trace information can be displayed.

Push **Trigger** in the **Trace Goto** dialog for the display of the trigger point. All records recorded after the trigger event have a positive record number.



Summary

; Stop trace recording when the specified address is executed
; (up to 4 single instructions or up to 4 instruction ranges)

Break.Set *<address>* | *<range>* **/Program /TraceTrigger**

; Stop trace recording when the specified data access occurred
; (up 4 single data accesses or up to 2 data access ranges)

Break.Set *<address>* | *<range>* **/ReadWrite** | **/Read** | **/Write /TraceTrigger**
Var.Break.Set *<hll_expression>* **/ReadWrite** | **/Read** | **/Write /TraceTrigger**
Break.Set *<address>* | *<range>* **/***<access>* **/Data.auto** *<data>* | **/Data.Byte** *<data>* **/TraceTrigger**
Break.Set *<address>* | *<range>* **/***<access>* **/Data.Word** *<data>* | **/Data.Long** *<data>* **/TraceTrigger**
Var.Break.Set *<hll_expression>* **/ReadWrite** | **/Read** | **/Write /Data.auto** *<data>* **/TraceTrigger**

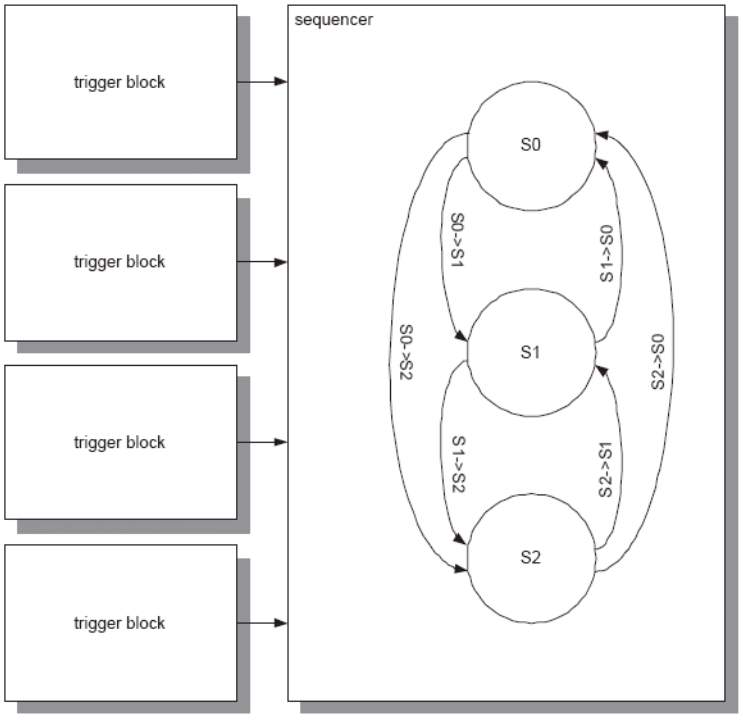
; Counter possible
Break.Set *<address>* | *<range>* **/***<access>* *<data_value>* **/TraceTrigger /COUNT** *<value>*
Var.Break.Set *<hll_expression>* **/***<access>* *<data_value>* **/TraceTrigger /COUNT** *<value>*

Filtering/Triggering via the ETM.Set

The **ETM.Set** commands allow a low-level programming of the triggering/filtering resources of the ETM.

The low-level programming of the ETM filters and trigger requires at least some basic knowledge about the so-called “event resources” provided by the Hexagon ETM. Please refer to your **ETM Architecture Specification**.

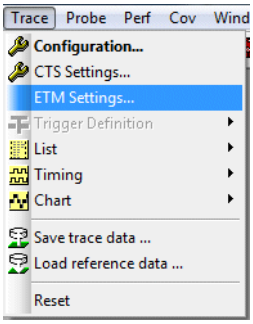
The event resources consist basically of 4 trigger blocks and a three state sequencer.



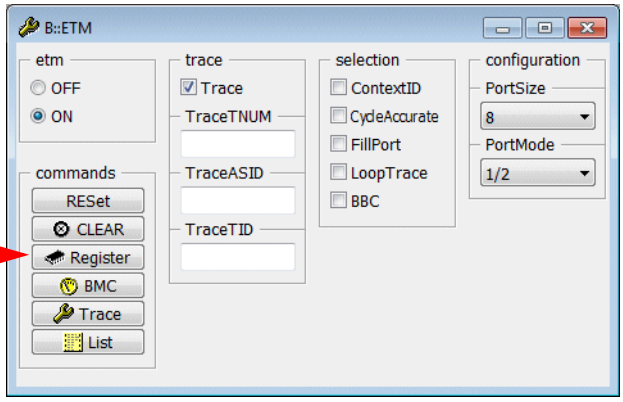
The low-level programming adds the following features:

- More sophisticated breakpoints than the **Break.Set** dialog.
- The sequencer allows to combine a series of events to form a breakpoint

The trigger block/sequencer configuration registers can be displayed as follows:

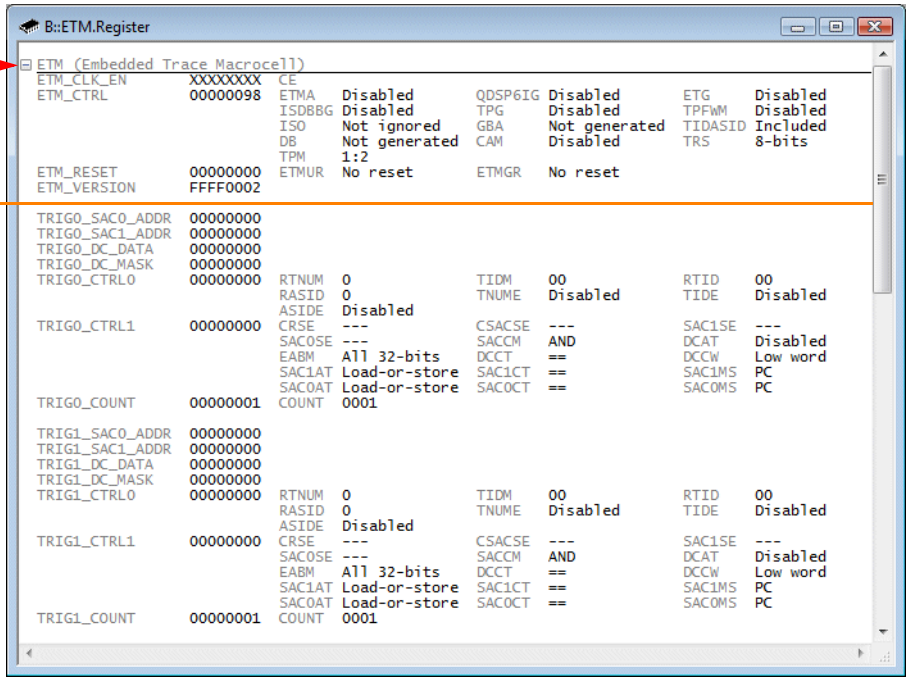


Click **Register** to display the ETM configuration registers

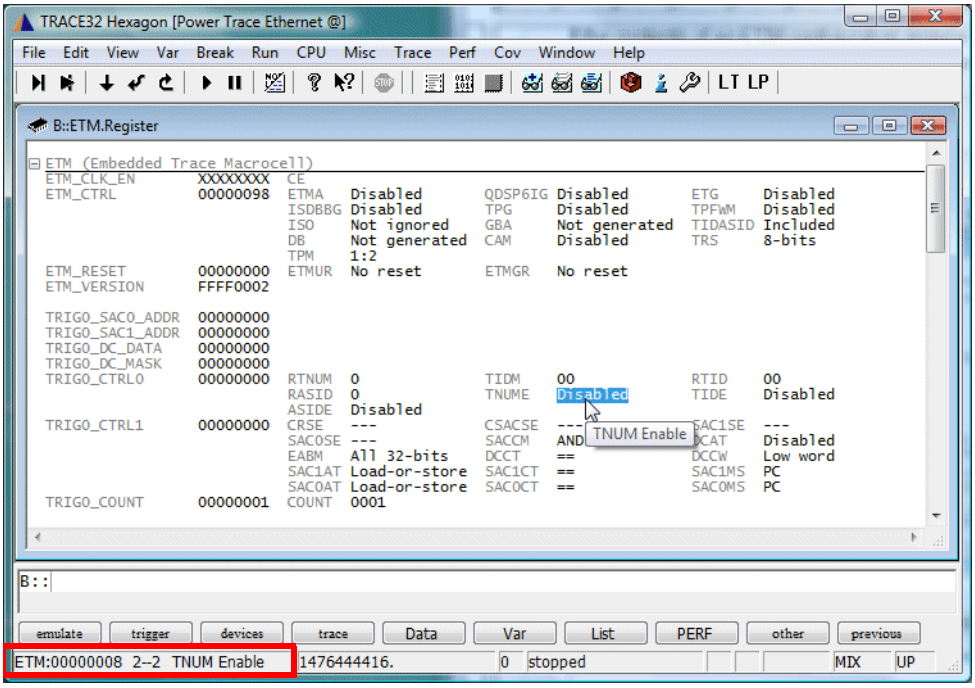


Click here to get details

Trigger block 0



If the contents of an ETM configuration register is selected, the address and a short description of the ETM register is displayed in the TRACE32 state line. For detailed information on the particular register, refer to the **ETM architecture specification**.

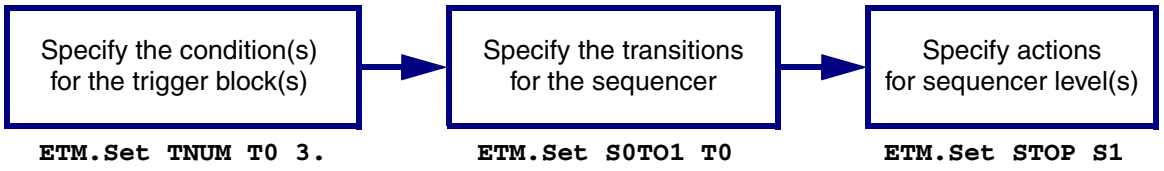


The ETM configuration registers can be read while the program execution is running. For an extensive usage of the ETM registers the following command is recommended:

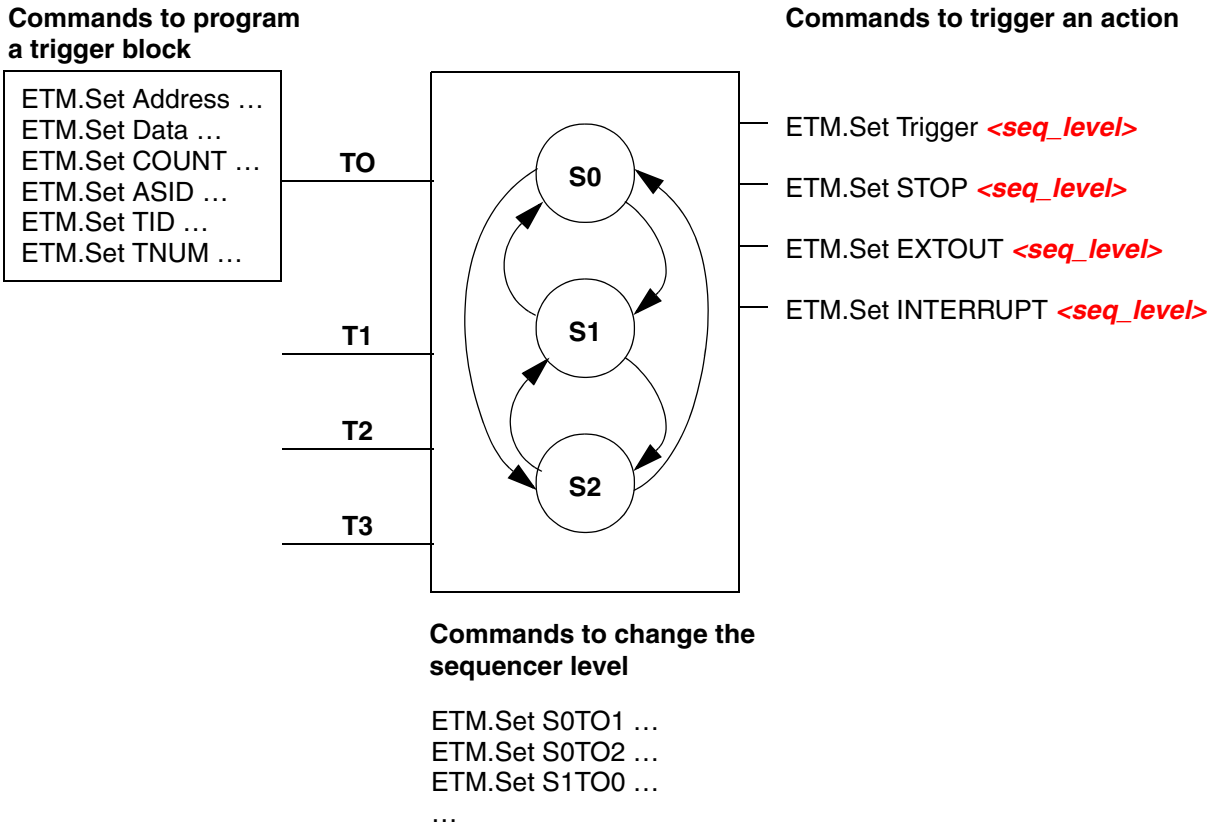
```
; Display the ETM configuration registers
; - mark changes by color (SpotLight)
; - update register display while program execution is running
;   (DualPort)
ETM.Register , /SpotLight /DualPort
```

Actions Based on Sequencer Level

Most trigger/filters are programmed as follows:



The following graphic shows the relevant **ETM.Set** commands:



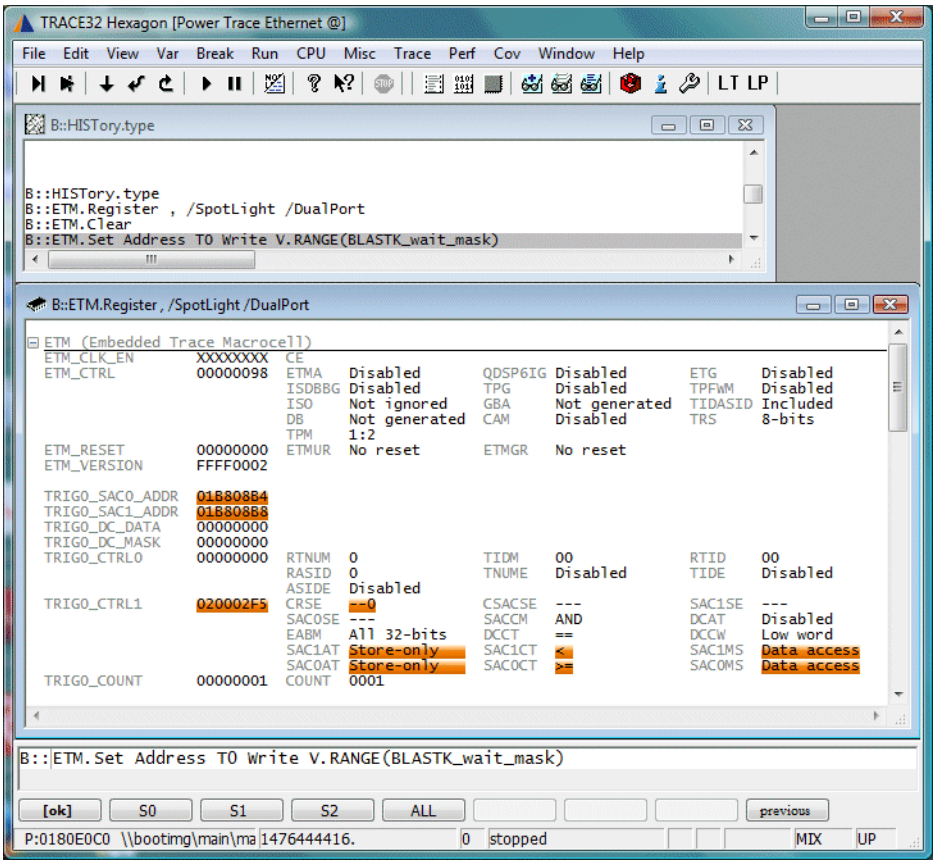
To illustrate actions based on sequencer level, the following examples are provided:

- Example 1: Stop the program execution if a value other than the specified one is written to the <variable X>.
- Example 2: Stop the program execution if a particular function was first executed by the hardware thread 1 and then by the hardware thread 3.

Example 1 - Actions based on Sequencer Level

Stop the program execution if a value other than 0x24 is written to the variable *BLASTK_wait_mask* (etm_set1.cmm).

```
; Display command history
HISTory.type

ETM.Register , /SpotLight /DualPort

; Reset all ETM registers
ETM.CLEAR
; Sequencer level 0 is active after ETM.Clear

; Program the address range of the variable mutex_lock into the
; address comparator of the trigger block 0, specify write access
ETM.Set Address T0 Write Var.RANGE(BLASTK_wait_mask)
```



```
; Program the data !0x24 into the data comparator of the trigger block 0
ETM.Set Data T0 != 0x24

; Change from sequencer level 0 to 1 if the event specified in trigger
; block 0 becomes true
ETM.Set S0TO1 T0

; Stop the program execution is sequencer level 1 is active
ETM.Set STOP S1
```

Please be aware, that this program stop is a one time stop. In order to stop the program execution for the same condition again, the same programming sequence needs to be reprogrammed.

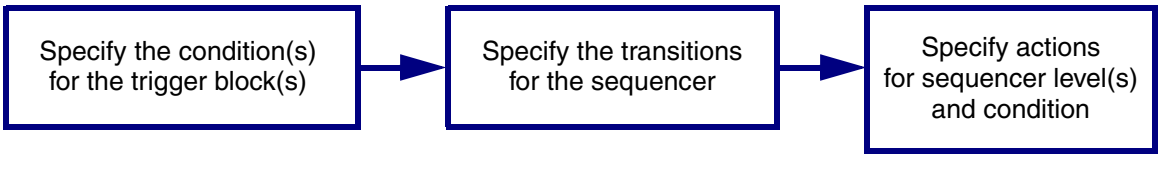
Example 2 - Actions based on Sequencer Level

Stop the program execution if the function *BLASTK_futex_wait* was first executed by the hardware thread 1 and then by the hardware thread 3 (etm_set2.cmm).

```
; Display command history
HISTory.type

; Reset all ETM registers
ETM.CLEAR
; sequencer level 0 is active after ETM.Clear

; Program the start address of the function BLASTK_writec into the
; address comparator of the trigger block 0
ETM.Set Address T0 Program BLASTK_futex_wait

; Program the hardware thread 0 into the TNUM comparator of the trigger
; block 0
ETM.Set TNUM T0 1.

; Change from sequencer level 0 to 1 if the event specified in trigger
; block 0 becomes true
ETM.Set S0TO1 T0

; Program the start address of the function BLASTK_writec into the
; address comparator of the trigger block 1
ETM.Set Address T1 Program BLASTK_futex_wait

; Program the hardware thread 3 into the TNUM comparator of the trigger
; block 1
ETM.Set TNUM T1 3.

; Change from sequencer level 1 to 2 if the event specified in trigger
; block 1 becomes true
ETM.Set S1TO2 T1

; Stop the program execution is sequencer level 2 is active
ETM.Set STOP S2
```

Actions Based on Sequencer Level and Condition

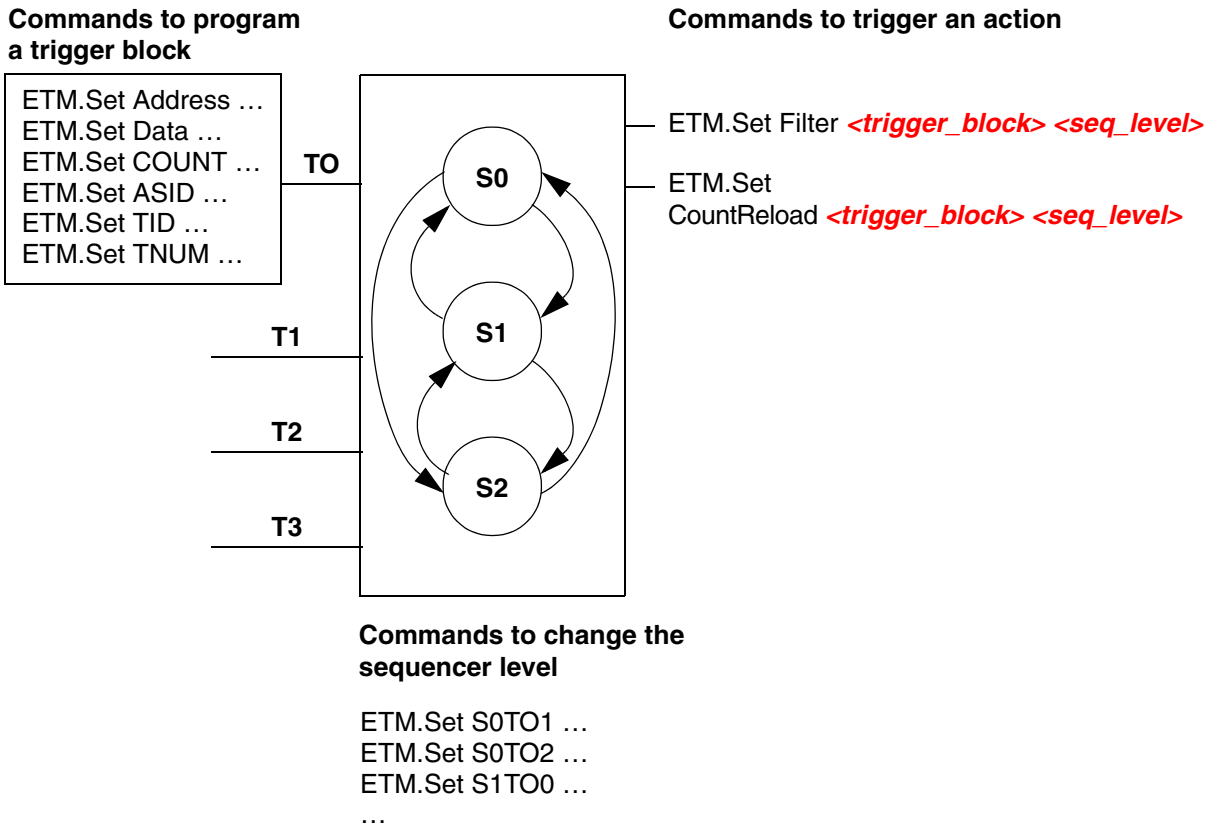
Some trigger/filters are programmed as follows:




```
ETM.Set TNUM T0 3.
```

```
ETM.Set Filter T0 S0
```

The following graphic shows the relevant **ETM.Set** commands:



To illustrate actions based on sequencer level and condition, the following examples are provided:

- Example 1: Program the ETM to export only trace information for *<hardware_thread_x>* and *<hardware_thread_y>*.
- Example 2: Program the ETM to export five times the entry to the *<function_x>* and one time the entry to the *<function_y>* repeatedly.
- Example 3: Stop the program execution after the *<function_x>* was called 10. times by hardware thread 0. Export only the function call.

Example 1 - Actions based on Sequencer Level and Condition

Program the ETM to export only trace information for hardware thread 0x0 and hardware thread 0x3 (etm_set3.cmm).

```
ETM.CLEAR                               ; Reset all ETM registers

ETM.Set TNUM T0 0x0                     ; Program the hardware thread 0x0
                                        ; into the TNUM comparator of the
                                        ; trigger block 0

ETM.Set TNUM T1 0x3                     ; Program the hardware thread 0x3
                                        ; into the TNUM comparator of the
                                        ; trigger block 1

ETM.Set Filter T0 ALL                   ; Export trace information in
                                        ; all sequencer levels if the
                                        ; condition specified for trigger
                                        ; block 0 is true

ETM.Set Filter T1 ALL                   ; Export trace information in
                                        ; all sequencer levels if the
                                        ; condition specified for trigger
                                        ; block 1 is true
```

Example 2 - Actions based on Sequencer Level and Condition

Program the ETM to export five times the entry to the function *blast_mutex_unlock* and one time the entry to the function *blast_mutex_lock* repeatedly (etm_set4.cmm).

```
; Reset all ETM registers
ETM.CLEAR
; sequencer level 0 is active after ETM.Clear

; Program the start address of the function blast_mutex_unlock into the
; address comparator of the trigger block 0

; Export the start address of the function blast_mutex_unlock if
; sequencer level 0 is active (alternative way to ETM.Set Filter …)
ETM.Set Address T0 Program blast_mutex_unlock S0

; Program the counter of trigger block 0 to 5.
ETM.Set Count T0 5.

; Change from sequencer level 0 to 1 if the event specified in trigger
; block 0 becomes true
ETM.Set S0TO1 T0

; Program the start address of the function blast_mutex_lock into the
; address comparator of the trigger block 1

; Export the start address of the function blast_mutex_lock if
; sequencer level 1 is active (alternative way to ETM.Set Filter …)
ETM.Set Address T1 Program blast_mutex_lock S1

; Change from sequencer level 1 to 0 if the event specified in trigger
; block 1 becomes true
ETM.Set S1TO0 T1

; Reload all counters if the event specified in trigger block 1 becomes
; true in the sequencer level 1
ETM.Set CountReload T1 S1
```

Example 3 - Actions based on Sequencer Level and Condition

Stop the program execution after the function *BLASTK_writec* was called 10. times by hardware thread 0. Export only the function call (etm_set5.cmm).

```
; Display command history
HISTory.type

; Reset all ETM registers
ETM.CLEAR
; sequencer level 0 is active after ETM.Clear

; Program the start address of the function BLASTK_writec into the
; address comparator of the trigger block 0

; Export this instruction as long as the sequencer level 0 is
; active
ETM.Set Address T0 Program BLASTK_writec S0

; Program the hardware thread 0 into the TNUM comparator of the trigger
; block 0
ETM.Set TNUM T0 0.

; Program the event counter of trigger block 0 with 10.
ETM.Set Count T0 10.

; Change from sequencer level 0 to 1 if the event specified in trigger
; block 0 becomes true
ETM.Set S0TO1 T0

; Stop the program execution is sequencer level 1 is active
ETM.Set STOP S1

; Display the result
Trace.List
```

Introduction

The ETM provides six 16-bit counters which can count one of the following events:

DCMISS	data cache misses
DCCONFLICT	data cache conflicts
ICMISS	instruction cache misses
ICSTALL	instruction cache stall-cycles
ITLBMISS	itlb misses
DTLBMISS	dtlb misses
STALLS	all stall cycles

TRACE32 PowerView enables you:

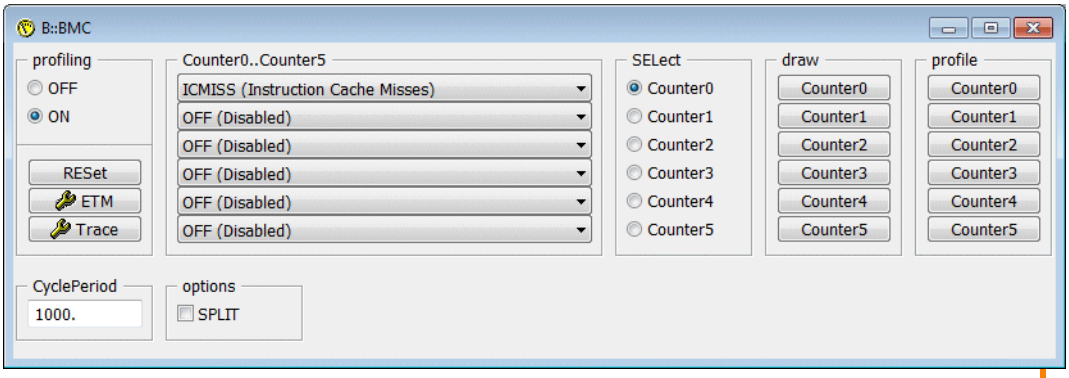
- to count the occurrence of up to six events summarized for all hardware threads (**BMC.SPLIT OFF**).
- to count the occurrence of a single event separately for each hardware thread (**BMC.SPLIT ON**).

The counters count their assigned event for a fixed number of clock cycles.

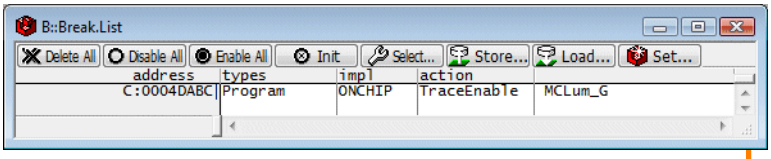
Profile packets containing the current counter values are exported by the ETM after this fixed number of cycles.

The benchmark counters, the filters provided by the ETM configuration window and the filter breakpoints can be combined.

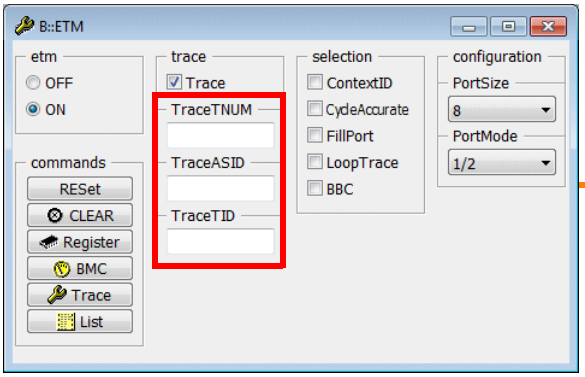
Benchmark counters



Filter breakpoints



ETM configuration



ETM trace packet generation

Trace repository*

* trace memory of PowerTrace or ETB

To illustrate the handling of benchmark counters, the following examples are provided:

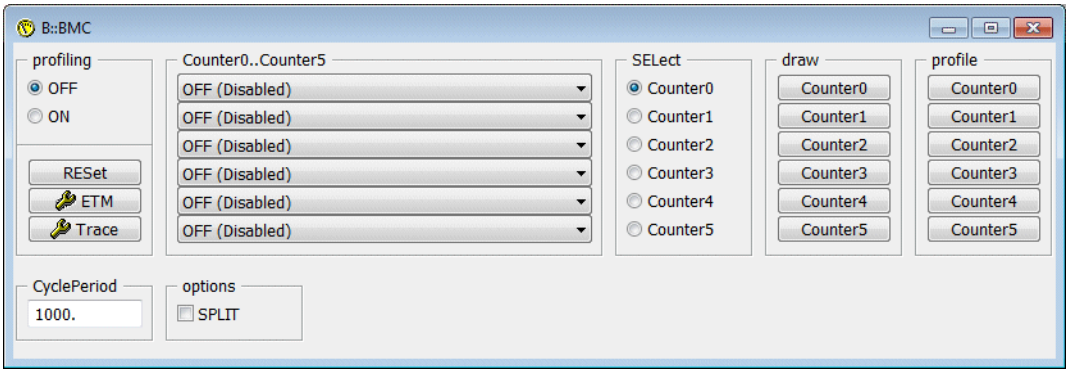
- Example 1: Count the total number of stall cycles and the number of instruction cache stall cycles summarized for all cores. Export this information every n clock cycles.
- Example 2: Count the total number of stall cycles separately for each hardware thread. Export this information every n clock cycles.
- Example 3: Count the instruction cache misses for hardware thread 0. Inspect the peak areas.
- Example 4: Count the total number of stalls between the entry to a particular function and the instruction at a particular address.

Example 1 - Benchmark Counters

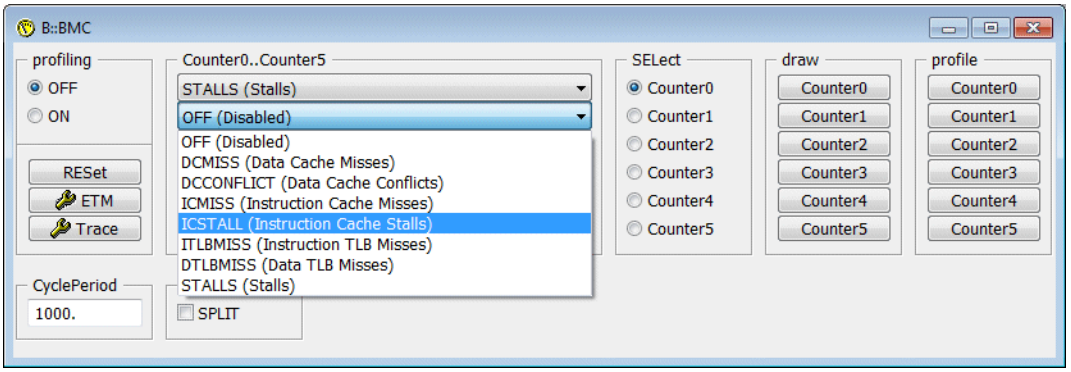
Count the total number of stall cycles and the number of instruction cache stall cycles summarized for all cores. Export this information every 500. clock cycles.

1. Open the benchmark counter configuration window.

```
BMC.state
```

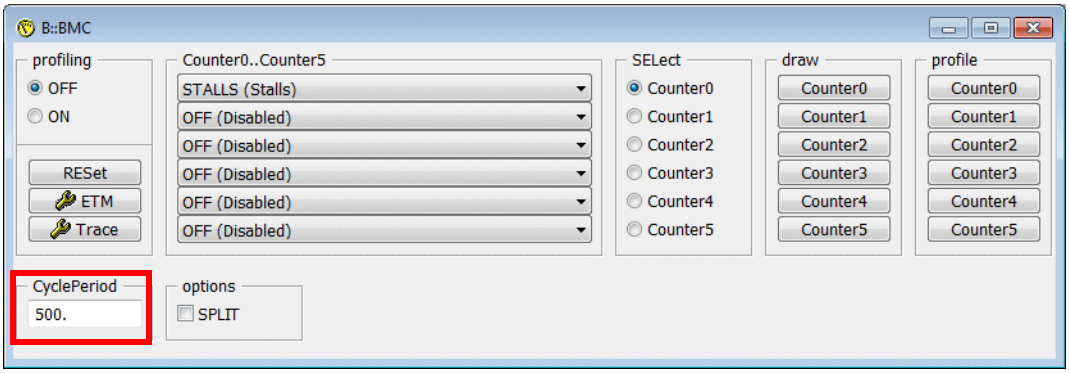


2. Configure the benchmark counters.



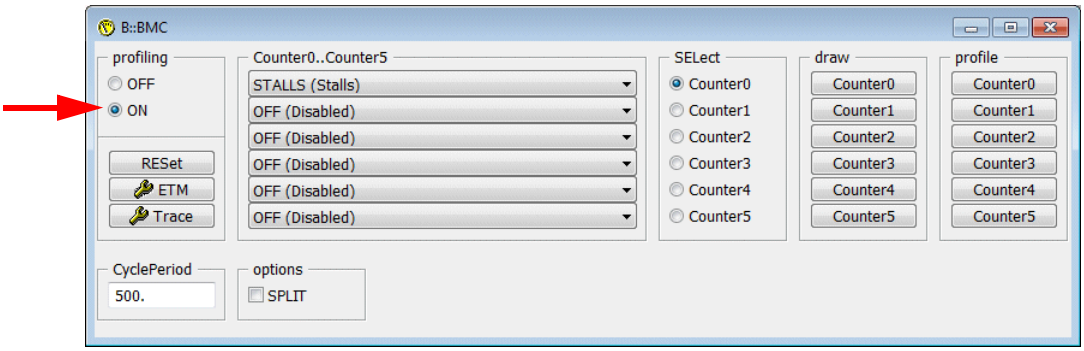
- **Counter0** counts the total number of stall cycles
- **Counter1** counts the number of instruction cache stall cycles

3. **Specify the exporting rate.**



- The counter contents are exported by the ETM all 500 clock cycles.

4. **Enable the TRACE32 BenchMark Counter functionality (BMC.ON).**

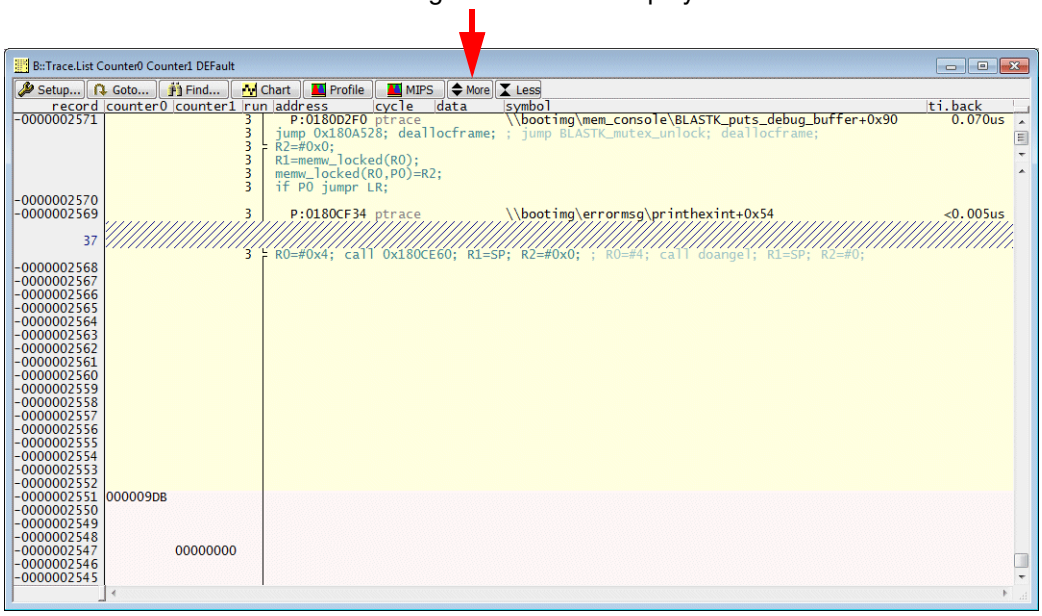


5. **Start and stop the program execution.**

6. Display the result.

```
Trace.List Counter0 Counter1 DEFault
```

Push the **More** button to get the counter display

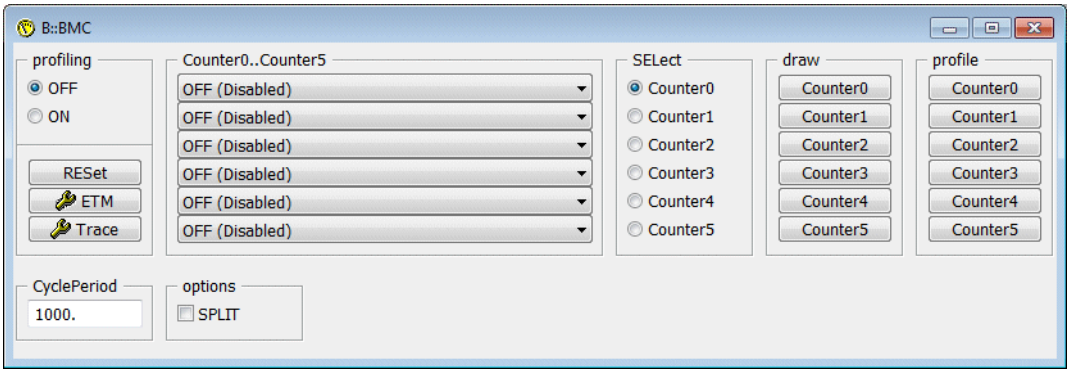


Example 2 - Benchmark Counters

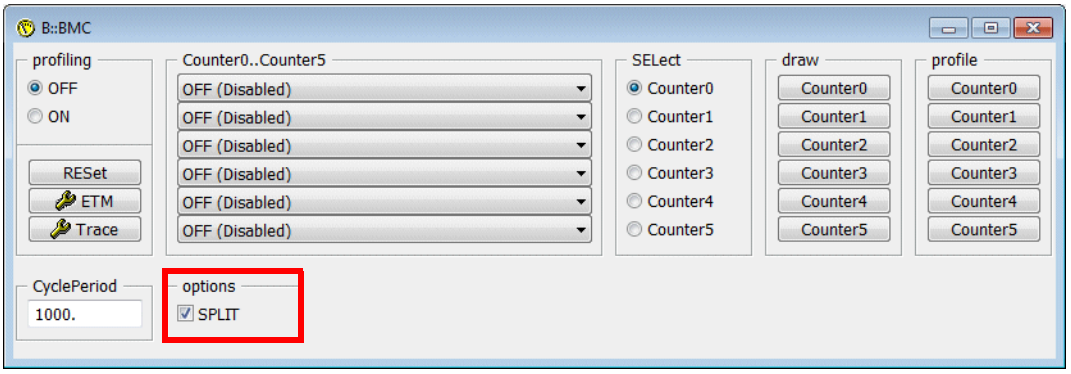
Count the total number of stall cycles separately for each hardware thread. Export this information all 500. clock cycles.

1. Open the benchmark counter configuration window.

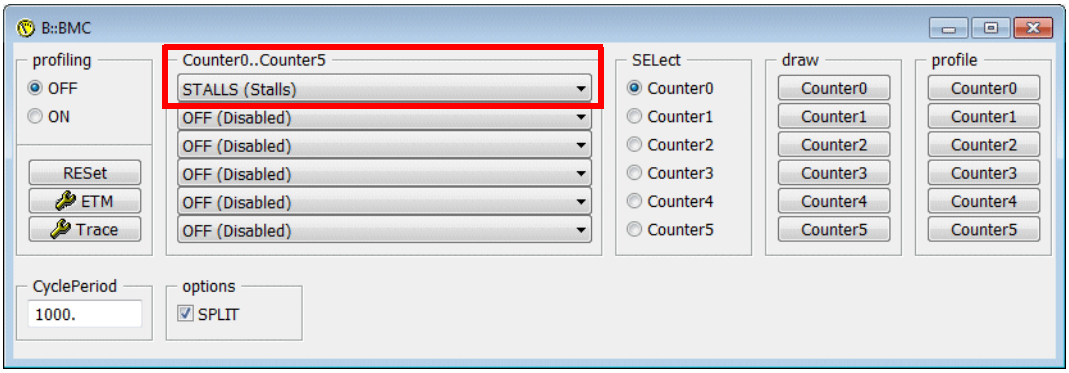
```
BMC.state
```



2. Activate the SPLIT option to program the ETM to count the specified event separately for each hardware thread.

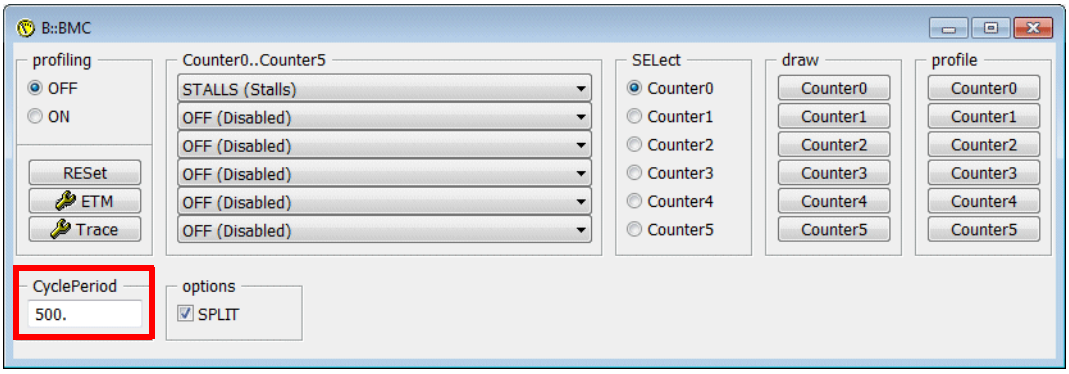


3. **Configure the benchmark counter Counter0.**



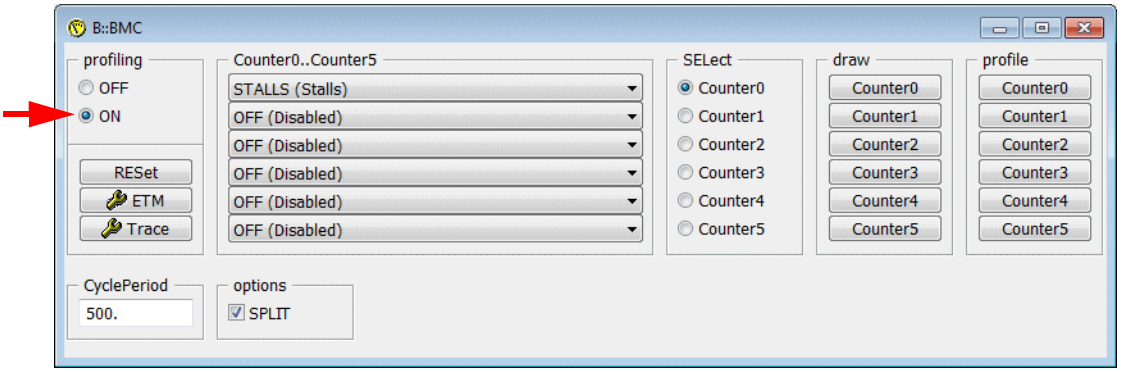
- Counter0 counts the total number of stall cycles

4. **Specify the exporting rate.**



- The counter contents are exported all 500 clock cycles.

5. **Enable the TRACE32 BenchMark Counter functionality (BMC.ON)**

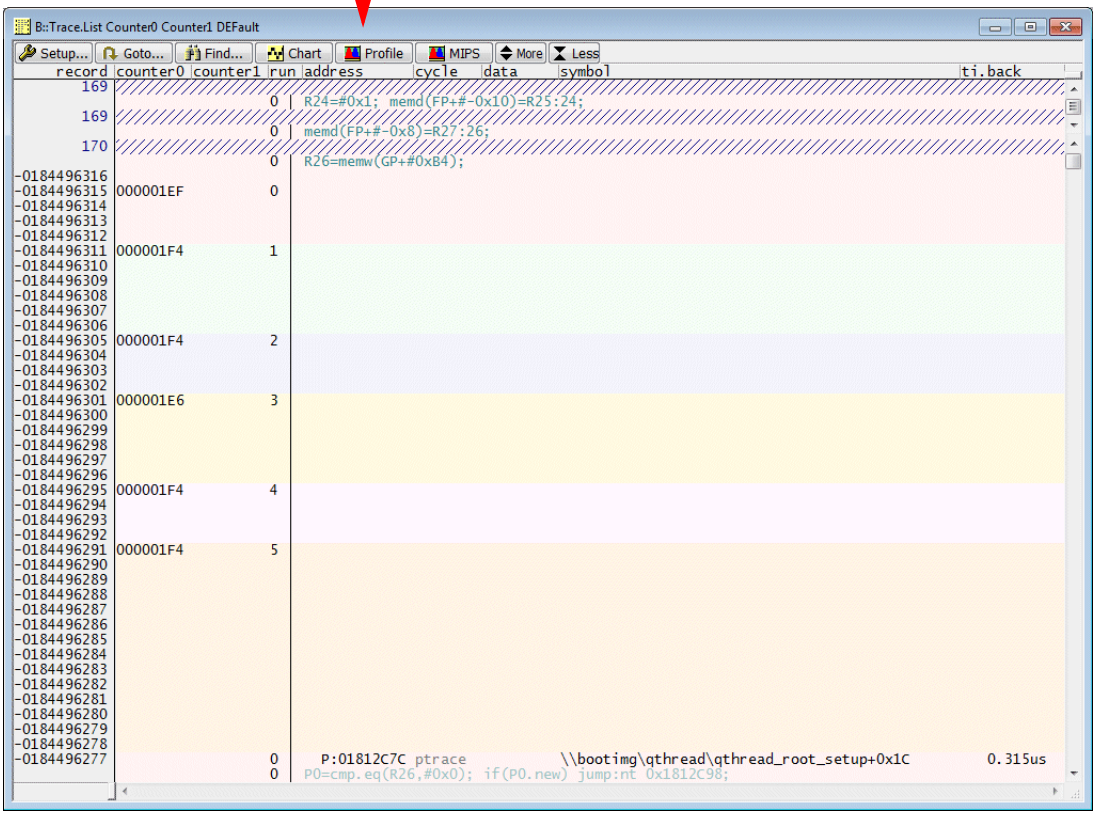


6. **Start and stop the program execution.**

7. **Display the result.**

```
Trace.List Counter0 Counter1 DEFault
```

Push the **More** button to get the counter display

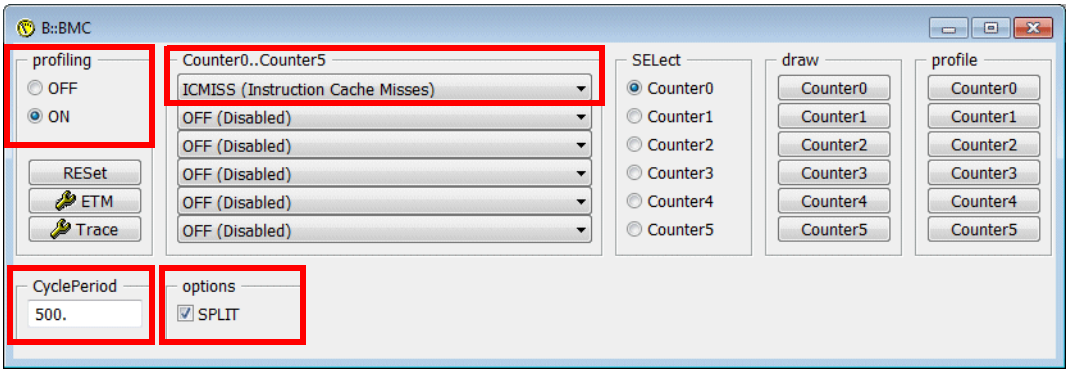


Example 3 - Benchmark Counters

Count the instruction cache misses for hardware thread 0. Inspect the peak areas.

1. **Configure the benchmark counter.**

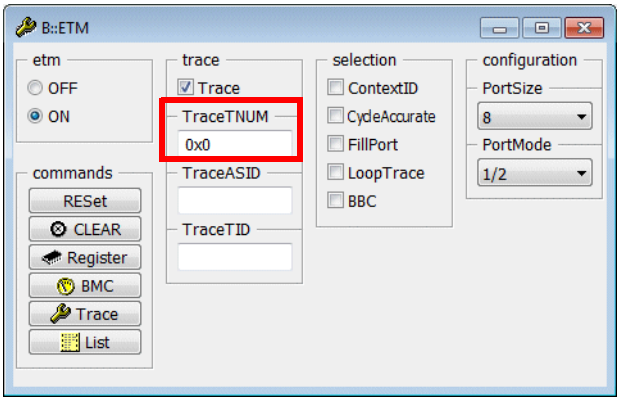
```
BMC.state
```



- Program the ETM to count the specified event for each hardware thread separately (**BMC.SPLIT ON**)
- Specify that Counter0 counts Instruction Cache Misses
- The counter contents is exported all 500. clock cycles
- Enable the TRACE32 BenchMark Counter functionality (**BMC.ON**)

2. **Program the ETM to export trace information only for hardware thread 0.**

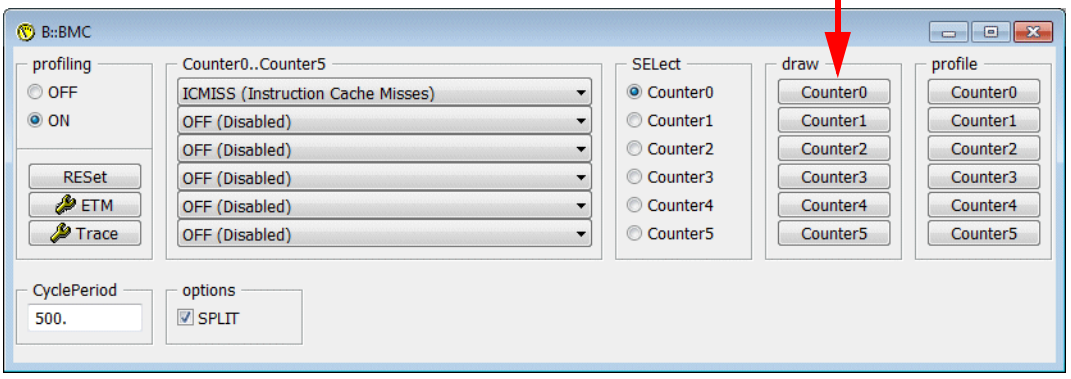
```
ETM.state
```



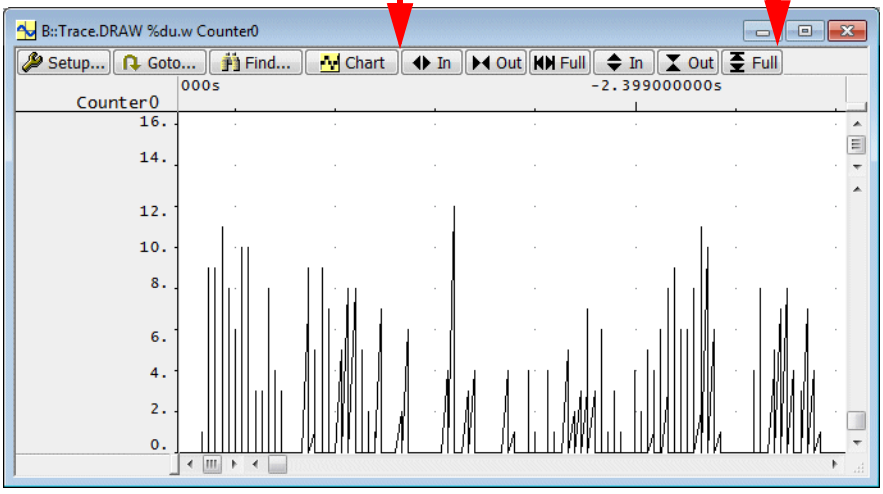
3. **Start and stop the program execution.**

4. Display the result.

Push **Counter0** in the draw field to get a graphical display of the counter values

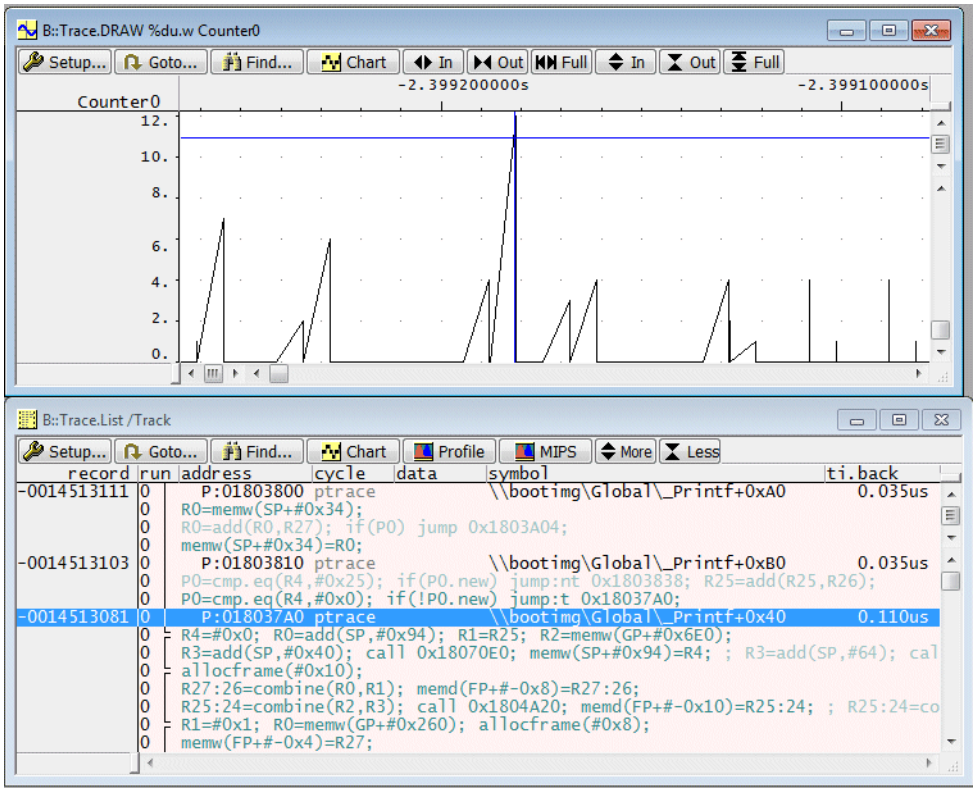


Use the zoom buttons in the display

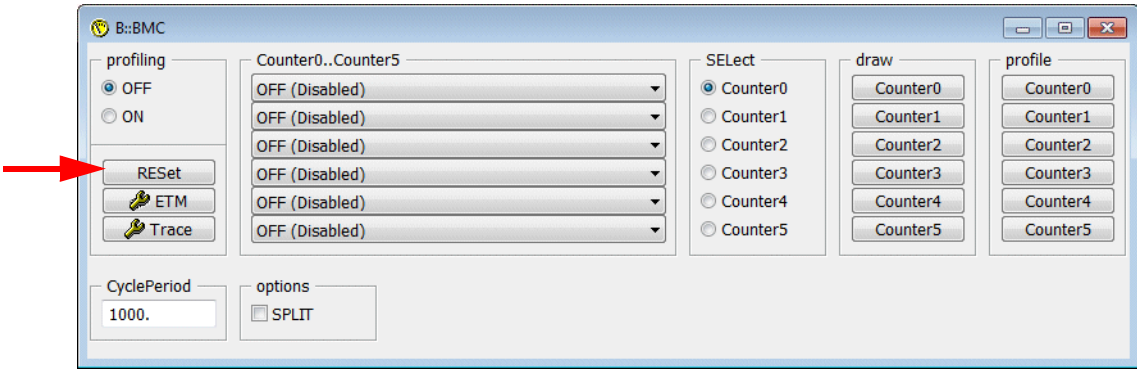


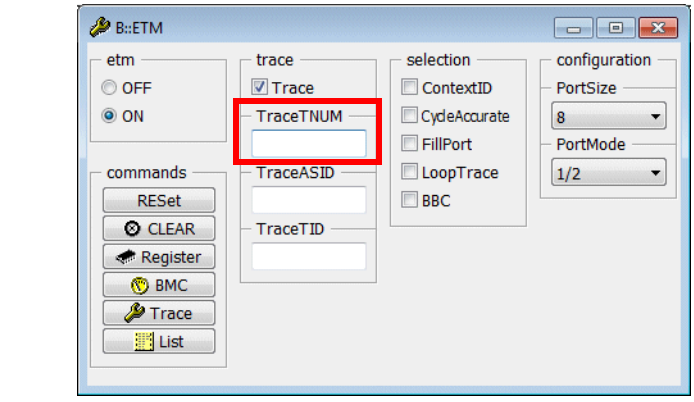
5. Open a trace listing to inspect peak areas.

```
Trace.List /Track
```



6. Reset all settings when you are done with your test.

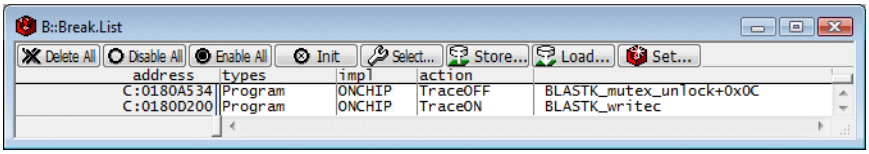




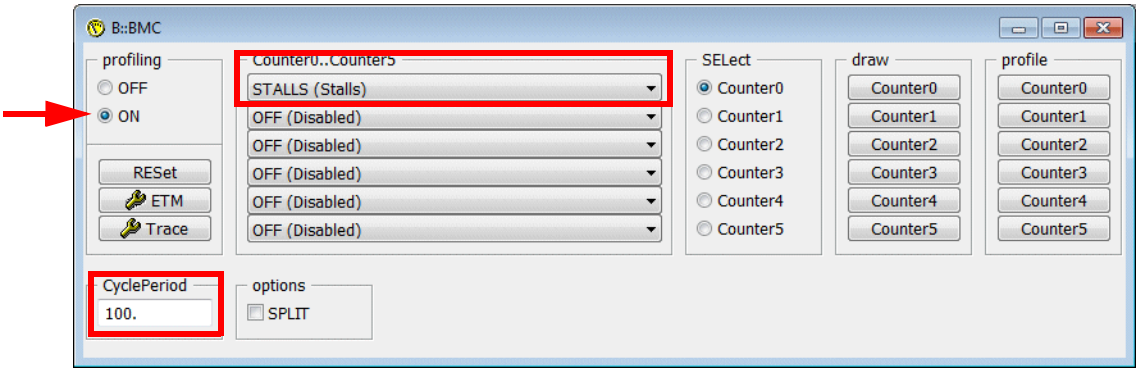
Example 4 - Benchmark Counters

Count the total number of stalls between the entry to the function *BLASTK_writec* and the instruction at address *BLASTK_mutex_unlock+0x0C*.

1. **Specify TraceON/TraceOFF breakpoints for the program range of interest.**



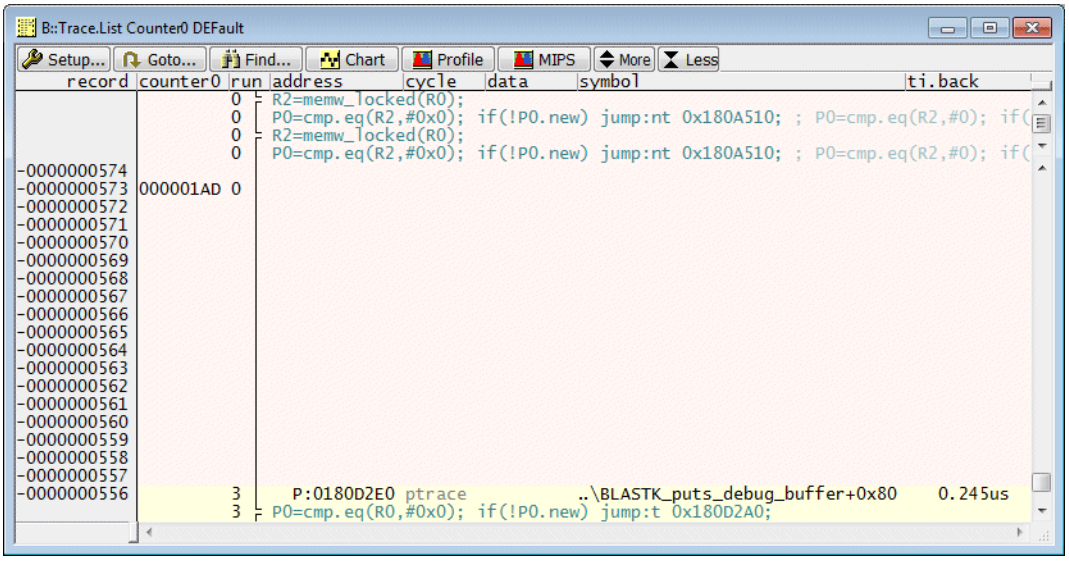
2. **Configure the benchmark counters.**



- Counter0 counts the total number of stalls
- The counter contents is exported all 100. clock cycles.
- Enable the TRACE32 BenchMark Counter functionality (**BMC.ON**)

3. **Start and stop the program execution.**

4. **Display the result.**



5. **Reset the benchmark counters and delete the breakpoints when you are done with your test.**

Function Run-time Analysis - Cache Misses/Stalls

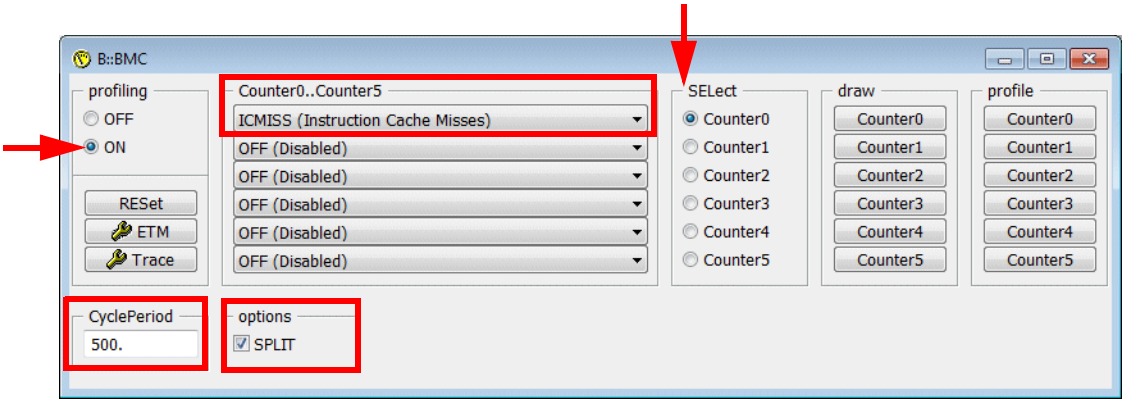
Function run-times increase with the number of stalls or/and cache misses. It makes sense to check such events.

Example

Analyze the number of Instruction Cache Misses for all function.

1. Configure the Benchmark Counter.

```
BMC.state                                   ; Open the benchmark counter
                                            ; configuration window
```

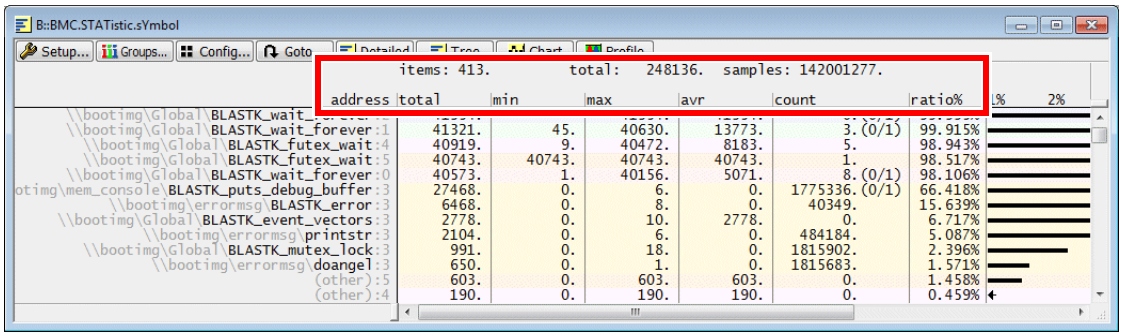


- Program the ETM to count the specified event for each hardware thread separately (**BMC.SPLIT ON**)
- Specify Instruction Cache Misses for **Counter0**.
- The counter contents is exported all 500 clock cycles.
- Enable the TRACE32 BenchMark Counter functionality (**BMC.ON**)
- **SELect** Counter0 as source for the benchmark counter statistic.

2. Start and stop the program execution.

3. **Display the result.**

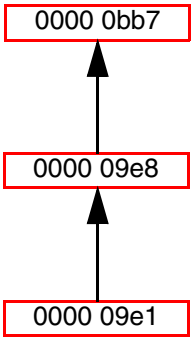
```
BMC.STATistic.sYmbol
```



For a description of the list summary and the columns, see tables below.

List Summary	
item	number of recorded functions/symbol regions
total	total number of stalls during measurement period
samples	number of recorded profiling packets

Columns with function details	
address	function name/name of symbol region (other) program sections that can not be assigned to a function
total	total number of stalls for the function during the recorded period
min	smallest number of stalls in a continuous address range of the function
max	largest number of stalls in a continuous address range of the function
avr	average number of stalls in a continuous address range of the function
count	number of new entries into the address range of the function/symbol region (start address executed)
ratio	ratio of stalls for the function with regards to the total number of stalls



Profiling packet

number of stalls is evenly split up on all instructions executed between 2 profiling packets

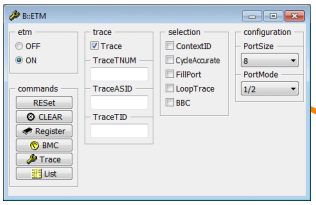
Profiling packet

Profiling packet

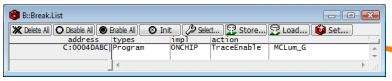
Summary: Trigger and Filter

A set of functions has an effect on the ETM trace packet generation. But at the end all these functions are using the same resources (the four trigger blocks and the sequencer provided by the ETM).

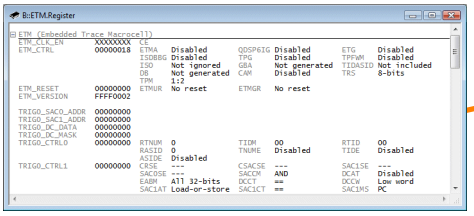
ETM configuration



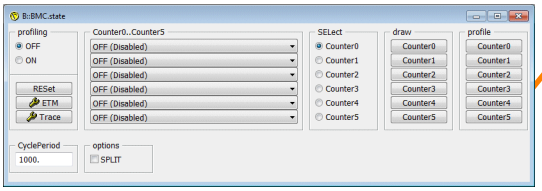
The filter and trigger breakpoints



The filter and trigger set via the ETM.Set command



The benchmark counters



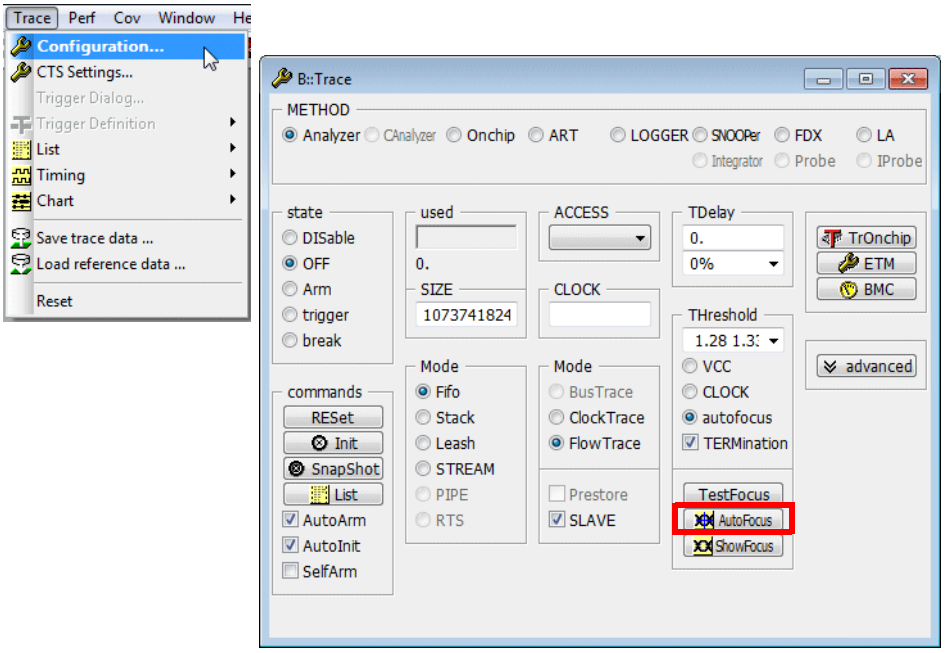
In the case of a resource conflict, prioritization is done as follows:

1. **ETM.Set commands**
2. **Break.Set commands**
3. **Benchmark counters**

	Please do not program the ETM resources via • Data.Set • PER.Set.simple TRACE32 may overwrite your settings.

The Calibration of the Recording Tool

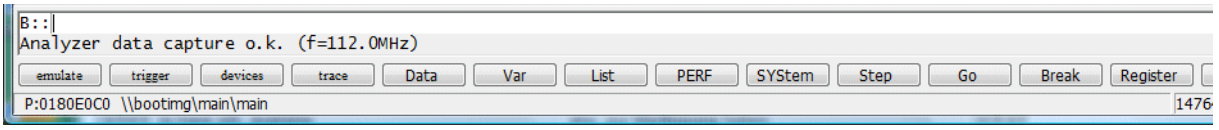
TRACE32 provide the **AutoFocus** button in the Trace configuration window to calibrate the recording tool.




```
Trace.AutoFocus
```

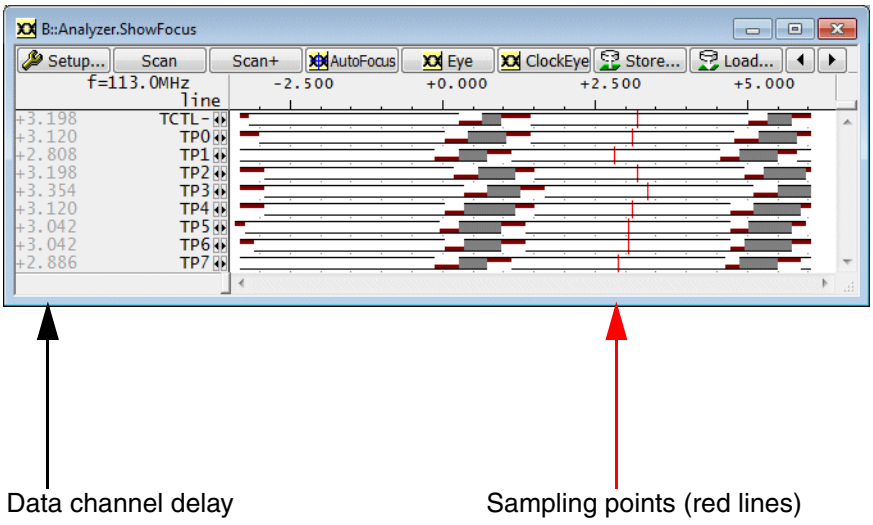
In order to perform the calibration TRACE32 loads a test program to the memory addressed by the PC or the stack pointer. It is also possible to define an *<address_range>* for the test program.

If the calibration is performed successfully, the following message will be displayed:



Frequencies smaller then 6 MHz result in f=0.0 MHz, since the frequency is maintained by TRACE32 as an integer.

The **ShowFocus** button in the Trace configuration window allows to inspect the result of the calibration.

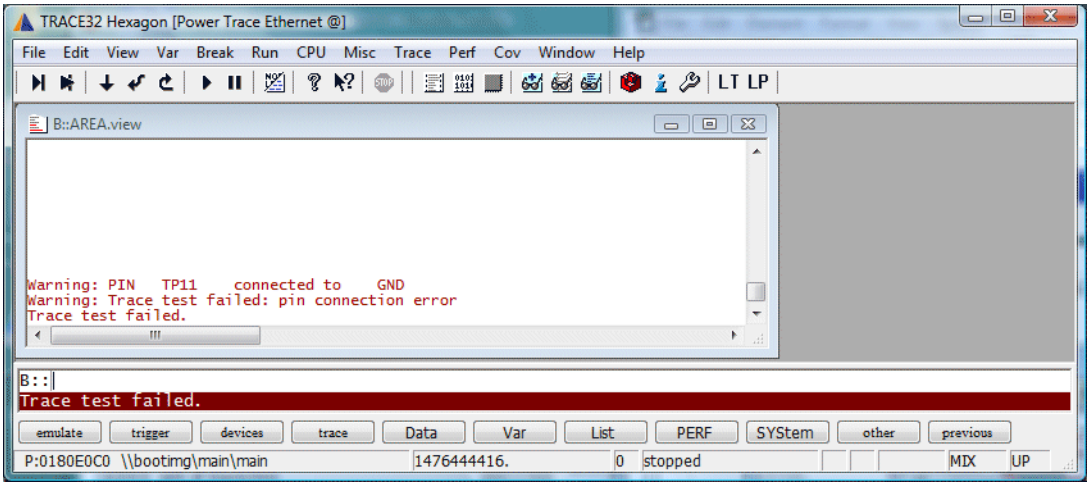


Data channel delay

Sampling points (red lines)

```
Trace.ShowFocus
```

If the calibration of the recording tool fails, the following error message is displayed:



The TRACE32 message area displays further diagnosis information.

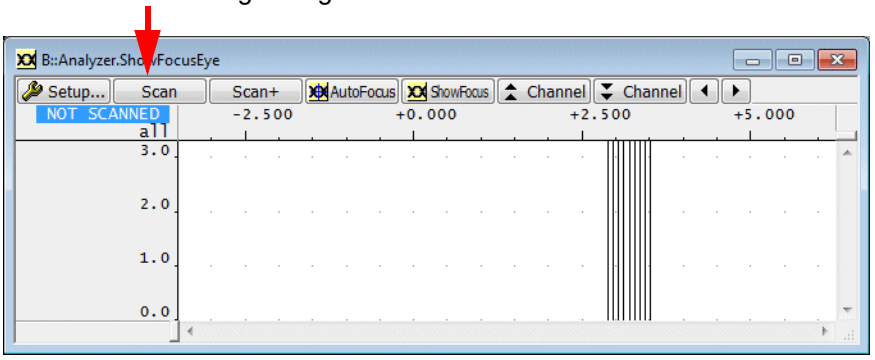
```
AREA.view
```

If the diagnosis information of TRACE32 is not sufficient to identify the problem, make sure that the following preconditions are fulfilled before you start a more detailed diagnosis:

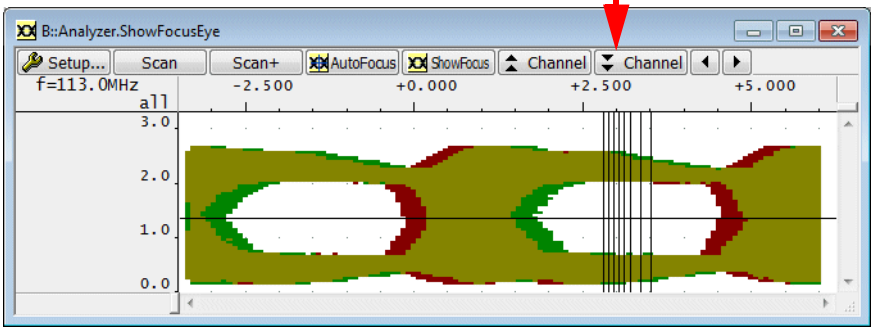
- The ETM is enabled on your target board.
- The ETM pins are enabled on your target board.

A helpful tool for further diagnosis can be the **Trace.ShowFocusEye** window.

Push **Scan** to get diagnosis data



Push **Channel** to check the data eyes of the trace channels



The recording tools can not dectect a data eye for TP11