LAUTERBACH
DEVELOPMENT TOOLS

Training Nexus Tracing

MANUAL

Training Nexus Tracing

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Training ..... 🗁

Training Nexus ..... 🗁

Training Nexus Tracing **1**

Basic Knowledge **8**

NEXUS Characteristics 8

Limited Bandwidth 9

Branch Trace Messages (All NEXUS Classes) 9

Indirect Branch History Messages (All NEXUS Classes) 10

Data Trace Messages (NEXUS Class 3 only) 12

Ownership Trace Messages (All NEXUS Modules) 14

Watchpoint Trace Messages (All NEXUS Classes) 15

Data Acquisition Messages (IEEE-ISTO 5001-2008/2012 and NEXUS Class 3 only) 16

Multicore Tracing 18

AMP Tracing 18

SMP Tracing 19

Source for the Recorded Trace Information 20

NEXUS Configuration by TRACE32 ..... **22**

Configuration of the Trace Interface 22

Parallel Interface 22

Serial Interface 28

Trace to Memory 30

Multicore Tracing 31

Configuration of the NEXUS Messages 33

Basic Messages 33

Additional Messages for IEEE-ISTO 5001-2008 and IEEE-ISTO 5001-2012 35

Add Timestamps to NEXUS Messages (MPC57xx/SPC57x only) 39

Multicore Tracing 43

NEXUS Trace Clients 44

Trace Client Types 44

Configuration 47

Target FIFO Overflow 48

Diagnosis 49

Stall Program Execution on Overflow Threat 50

Suppress Data Trace Messages on Overflow Threat 51

Further Countermeasures 52
FlowErrors 53
Displaying the Trace Content 54
Sources of Information for the Trace Display 54
Settings in the Trace Configuration Window 55
Recording Modes 55
States of the Trace 64
The AutoInit Command 65
Multicore Tracing 66
Basic Display Commands 67
Default Listing 67
Basic Formatting 69
Correlating the Trace Listing with the Source Listing 70
Browsing through the Trace Buffer 71
Display Items 72
Default Display Items 72
Further Display Items 76
Time Synchronization between TRACE32 Instances (AMP) 77
Setup 77
Utilization 78
Find a Specific Record 79
Belated Trace Analysis 80
Save the Trace Information to an ASCII File 81
Postprocessing with TRACE32 Instruction Set Simulator 82
Trace-based Debugging (CTS) 86
Re-Run the Program 86
Setup 86
Get Started 88
Forward and Backward Debugging 90
Re-Run the Program and Watch the Variables 91
Setup 91
Get Started 93
Forward and Backward Debugging 97
Details on HLL Instructions 98
CTS Technique 99
Filter and Trigger (Core) Overview 101
Resources 102
Filter and Trigger (Core) - Single Core 106
Examples for TraceEnable on Instructions 106
Example for TraceEnable on Instruction Range 111
Examples for TraceEnable on Read/Write Accesses 114
Example for TraceData 119

Examples for TraceON/TraceOFF 121
Global TraceON/Trace OFF 121
ProgramTraceON/Trace OFF 126
DataTraceON/Trace OFF 130
Example for TraceTrigger 134
Example for TraceTrigger with a Trigger Delay 137
Example for BusTrigger 140
Example for BusCount (Watchpoint) 142

Filter and Trigger (Core) - SMP Debugging 146
Examples for TraceEnable on Single Instruction 147
Examples for TraceEnable on Instruction Range 153
Examples for TraceEnable on Read/Write Accesses 156
Example for TraceData 161
Examples for TraceON/TraceOFF 163
Global TraceON/Trace OFF 163
ProgramTraceON/TraceOFF 168
DataTraceON/Trace OFF 173
Example for TraceTrigger 177
Example for TraceTrigger with a Trigger Delay 180
Example for BusTrigger 184
Example for BusCount (Watchpoint) 186

Filter and Trigger (Trace Clients) 190
Example for TraceEnableClient1 191

OS-Aware Tracing (ORTI File) 193
Activate the TRACE32 OS Awareness 193
Exporting Task Information (Overview) 195

OS-Aware Tracing - Single Core 196
Exporting all Types of Task Information (OTM) 196
Statistic Analysis of Task Switches 199
Statistic Analysis of OSEK Service Routines 201
Statistic Analysis of OSEK ISR2s 203
Statistic Analysis of Task-related OSEK ISR2s 204
Exporting all Types of Task Information and all Instructions (OTM) 205
Statistic Analysis of Interrupts 205
Statistic Analysis of Interrupts and Tasks 206
Statistic Analysis of Interrupts in Tasks 207
Exporting Task Information (Write Access) 208
Task Switches 208
OSEK Service Routines 211
OSEK ISR2s 214
Task-related OSEK ISR2s 217
Exporting Task Switches and all Instructions (Write Access) 220

Statistic Analysis of Interrupts 220
Statistic Analysis of Interrupts and Tasks 221
Statistic Analysis of Interrupts in Tasks 222
Belated Trace Analysis (OS) 223

OS-Aware Tracing - SMP Systems 224
Exporting all Types of Task Information (OTM) 224
Statistic Analysis of Task Switches 227
Statistic Analysis of OSEK Service Routines 229
Statistic Analysis of OSEK ISR2s 231
Statistic Analysis of Task-related OSEK ISR2s 233
Exporting all Types of Task Information and all Instructions (OTM) 235
Statistic Analysis of Interrupts 235
Statistic Analysis of Interrupts and Tasks 236
Statistic Analysis of Interrupts in Tasks 237
Exporting Task Information (Write Access) 238
Task Switches 238
OSEK Service Routines 243
OSEK ISR2s 248
Task-related OSEK ISR2s 253
Exporting Task Switches and all Instructions (Write Access) 257
Statistic Analysis of Interrupts 257
Statistic Analysis of Interrupts and Tasks 258
Statistic Analysis of Interrupts in Tasks 259
Belated Trace Analysis (OS) 260

Function Run-Times Analysis (Overview) 261
Software under Analysis (no OS or OS) 261
Flat vs. Nesting Analysis 261
Basic Knowledge about Flat Analysis 262
Basic Knowledge about Nesting Analysis 263
Summary 265

Function Run-Times Analysis - Single 266
Flat Analysis 266
Optimum NEXUS Configuration (No OS) 266
Optimum NEXUS Configuration (OS) 267
Function Time Chart 268
Nesting Analysis 276
Restrictions 276
Optimum NEXUS Configuration (No OS) 276
Optimum NEXUS Configuration (OS) 277
Items under Analysis 278
Numerical Nested Function Run-time Analysis for all Software 281
Additional Statistics Items for OS 289

Timing Improvements for OS 292
Problems and Workarounds for OS 293
More Nesting Analysis Commands 297
Third-party Timing Tools 305
Function Run-Times Analysis - SMP Instance 306
Flat Analysis 306
Optimum NEXUS Configuration (No OS) 306
Optimum NEXUS Configuration (OS) 307
Function Timing Diagram 308
Function Timing Diagram (Including Task Information) 310
Nesting Analysis 316
Restrictions 316
Optimum NEXUS Configuration (OS) 317
Numerical Nested Function Run-time Analysis for all Software 318
Timing Improvements for OS 326
More Nesting Analysis Commands 327
Third-party Timing Tools 334
Structure the Trace Evaluation 335
GROUP Creation 335
Working with GROUPs 339
GROUP Status ENable 339
GROUP Status ENable + Merge 341
GROUP Status ENable + HIDE 342
Trace-based Code Coverage 343

Training Nexus Tracing

Version 06-Jun-2024

NEXUS Characteristics

NEXUS is a message-based trace protocol. A NEXUS hardware module generates the trace messages. Trace messages can be generated for activities of core(s), eTPU(s), GTM(s), for activities of DMA controller(s), of FlexRay controller(s), of SRAM port sniffers and other units. The **Source Processor Identifier** in the NEXUS messages identifies the trace source.

NEXUS hardware modules are available in two versions:

- **NEXUS Class 2 + Modules** provide the visibility of the instruction flow and task switches.
- **NEXUS Class 3 + Modules** provide the visibility of the instruction flow, load/store operations, task switches and trace information generated by code instrumentation.

Trace messages generated by a NEXUS module:

- can be exported off-chip via a **parallel trace interface**.
- can be exported off-chip via a **serial (Aurora) trace interface**.
- can be stored to an on-chip trace memory (**trace to memory**).

NEXUS hardware modules are compliant to one of the following standards:

- **IEEE-ISTO 5001™-2012**

  Serial (Aurora) trace interfaces are always compliant to this standard.
- **IEEE-ISTO 5001™-2008**
- **IEEE-ISTO 5001™-2003**

Before you continue with this training, refer to your processor manual and check:

- Which class is supported by your NEXUS module?
- Are trace messages exported off-chip via a parallel or serial trace interface?
- Are trace messages stored to an on-chip trace memory?
- Which NEXUS standard is supported by your NEXUS module?

Limited Bandwidth

Regardless of the implementation of your NEXUS module (off-chip export or on-chip trace memory) it may happen while testing that more trace messages are generated than the trace interface/memory interface can convey. This may disturb your tests.

For a better understanding of this issue and its counter-measures, a short introduction into the NEXUS protocol is given. The following example configuration is used: MPC5775K with parallel trace interface consisting of 16 pins (MDO) for the export of NEXUS messages. The term *trace beat* is used for the trace information that is transferred per trace clock.

Branch Trace Messages (All NEXUS Classes)

Branch trace messages provide a standard protocol for instruction flow visibility.

Direct Branch Messages

TCODE number = 3 (6 bits)
Source processor identifier (4 bits)
Number of sequential instructions executed since the last taken branch (1 to 8 bits)
Timestamp (optional) (0 to 30 bits)

11 to 48 bits
in 1 to 4 trace beats

Indirect Branch Messages

TCODE number = 4 (6 bits)
Source processor identifier (4 bits)
Address space indicator (1 bit)
Number of sequential instructions executed since the last taken branch (1 to 8 bits)
Branch destination address (1 to 32 bits)
Timestamp (optional) (0 to 30 bits)

13 to 81 bits
in 1 to 6 trace beats

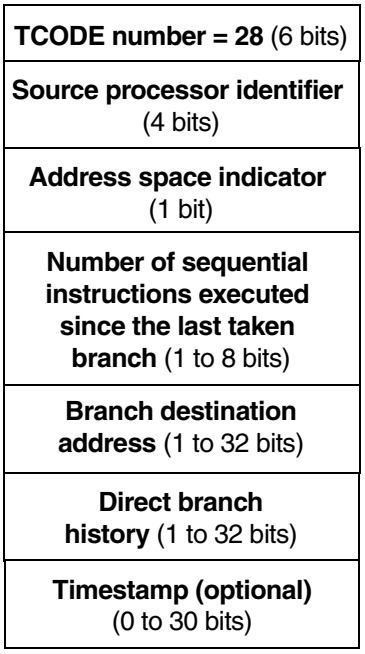
Indirect Branch History Messages (All NEXUS Classes)

Indirect Branch History Messages can be used to save bandwidth, since only indirect branches cause messages. Information on direct branches is stored in the Direct Branch History.

Indirect Branch History Messages are recommended for:

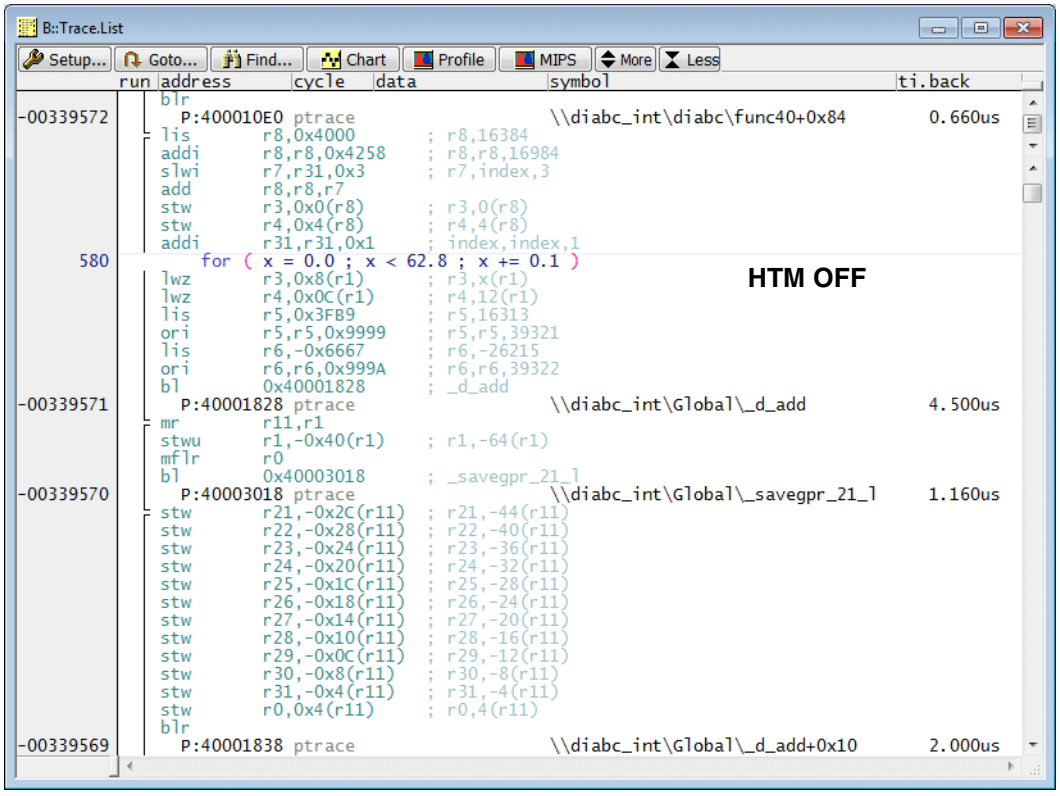
- small trace ports if they have bandwidth problems
- long instruction flow traces
- TRACE32 Trace Mode STREAM
- multi-source traces

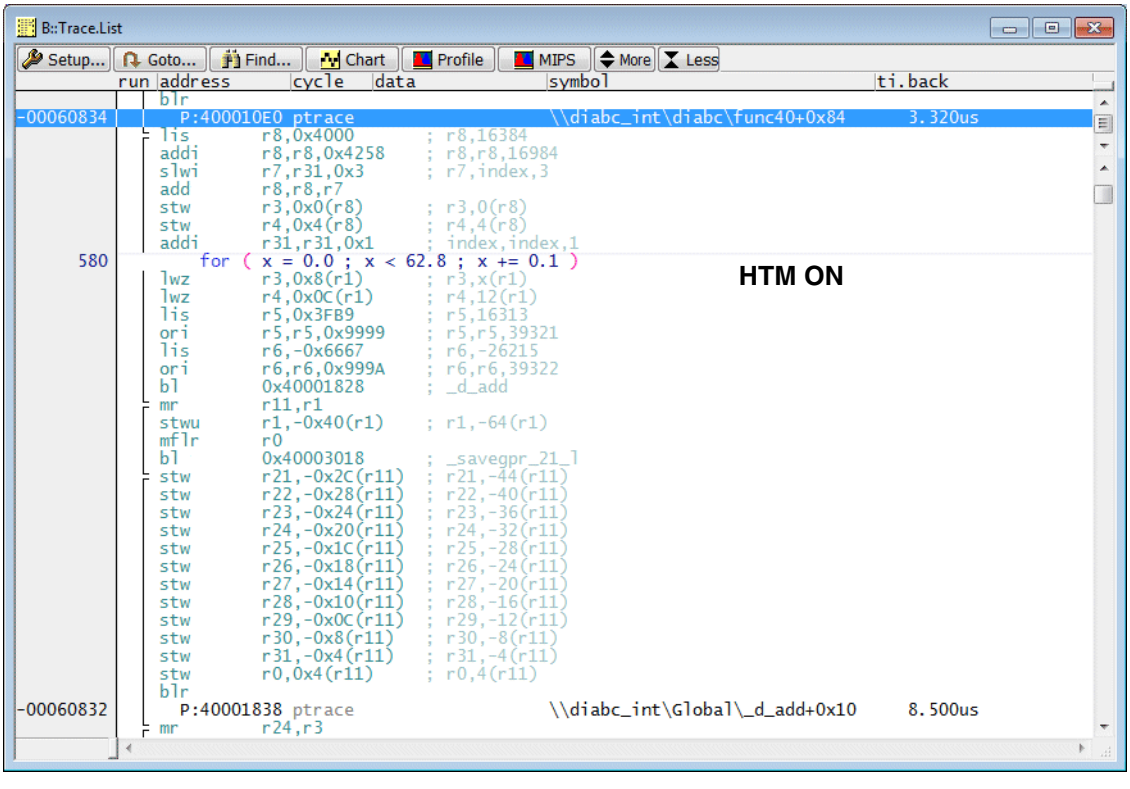
Indirect Branch History Messages



14 to 113 bits
in 1 to 8 trace beats

- The caveat of the use of Indirect Branch History Messages is a less accurate timestamp, since less NEXUS messages are generated and timestamped.





Data Trace Messages (NEXUS Class 3 only)

Data trace messages are used to export information on the load/store operations.

Data write messages

TCODE number = 5 (6 bits)
Source processor identifier (4 bits)
Address space Indicator (1 bit)
Data size (4 bits)
Data write address (1 to 32 bits)
Data write value (1 to 64 bits)
Timestamp (optional) (0 to 30 bits)

17 to 141 bits
in 2 to 9 trace beats

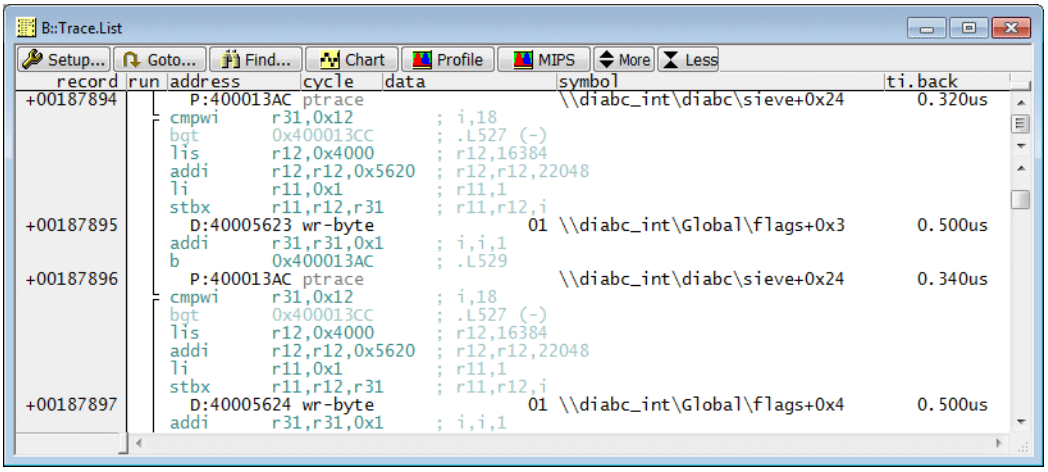
Data read messages

TCODE number = 6 (6 bits)
Source processor identifier (4 bits)
Address space Indicator (1 bits)
Data size (4 bits)
Data read address (1 to 32 bits)
Data read value (1 to 64 bits)
Timestamp (optional) (0 to 30 bits)

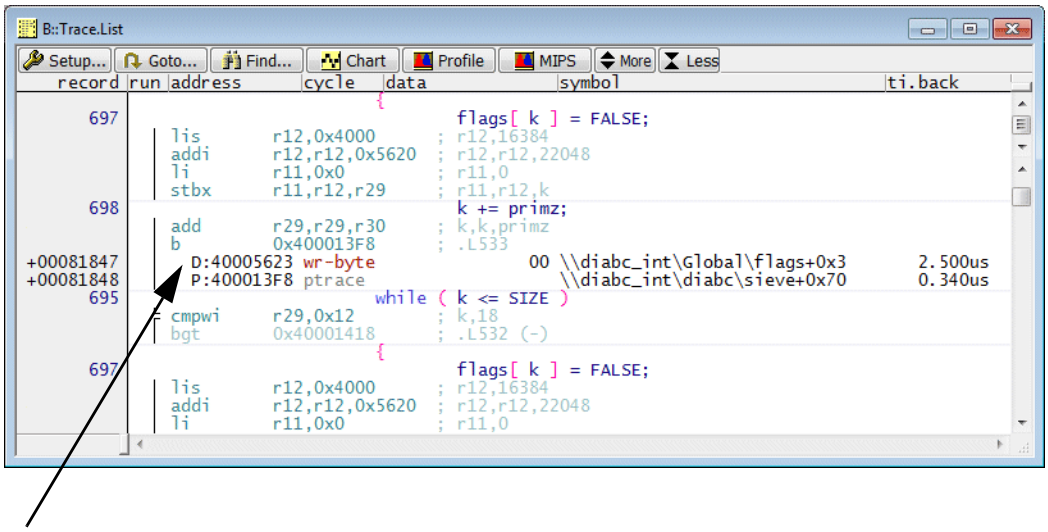
17 to 141 bits
in 2 to 9 trace beats

Exporting information on load/store operations may easily generate more trace messages than the interface in use can convey. This is most likely to occur when several data accesses are carried out in quick succession.

If information on all load/store operations is exported, each data access can be correlated to its instruction (data cycle assignment).



If a trace filter is used to export only some load/store operations, the correlation to the instruction is not always possible.



It was not possible to correlate the load/store operation to its instruction. For this reason the data access cycle is printed in red and is displayed preceding the next Branch Trace Message.

Ownership Trace Messages (All NEXUS Modules)

Ownership trace messages are trace messages that are generated when a write access to the Process ID register PID0 (8 bit) occurs.

Ownership trace messages can be used to export OS-related information e.g. task switch information for NEXUS Class 2 Modules.

Ownership Trace Message

TCODE number = 2 (6 bits)
Source processor identifier (4 bits)
Task/Process ID tag (32 bits)

42 bits
in 3 trace beats

Alternative for IEEE-ISTO 5001™-2012

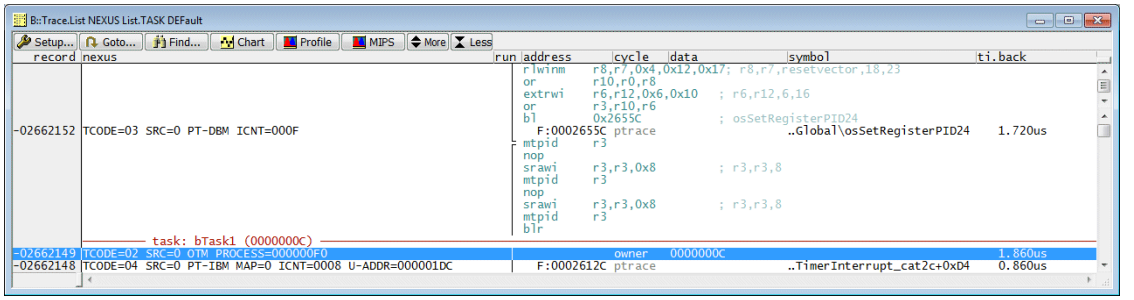
Since 8 bits are often not sufficient to encode OS-related information, the 32-bit NEXUS PID Register (NPIDR) can be used as an alternative. Ownership Trace Messages have also a slightly different format for IEEE-ISTO 5001™-2012.

Ownership Trace Message

TCODE number = 2 (6 bits)
Source processor identifier (4 bits)
Task/Process ID tag (1 to 32 bits)
Timestamp (optional) (0 to 30 bits)

11 to 72 bits
in 1 to 5 trace beats

The Ownership Trace Messages can not clearly be assigned to an instruction. Similar to the filtered Data Trace Messages they are printed in red and displayed preceding the next Branch Trace Message.



Watchpoint Trace Messages (All NEXUS Classes)

The Onchip breakpoints of the MPC5xxx/SPC5xxx can be used:

- to stop the program execution at a specific event.
- to generate a pulse on EVTO at a specific event.

  - Not available for AMP systems if synchronous break is activated.

  - Not available for SMP systems.
- to export Watchpoint Hit Messages.

Data Acquisition Messages (IEEE-ISTO 5001-2008/2012 and NEXUS Class 3 only)

Data Acquisition Messaging (DQM) allows code to be instrumented to export customized trace information.

Data Acquisition Messages are trace messages that are generated when a write access to the Debug Data Acquisition Message register DDAM (32 bit) occurs. DQTAG (8 bit) is sampled from the DEVENT register when a write to DDAM is performed.

The DQTAG field can be used to attribute the information written to DDAM. E.g. the DQTAG field can be interpreted by the trace tool as a channel ID.

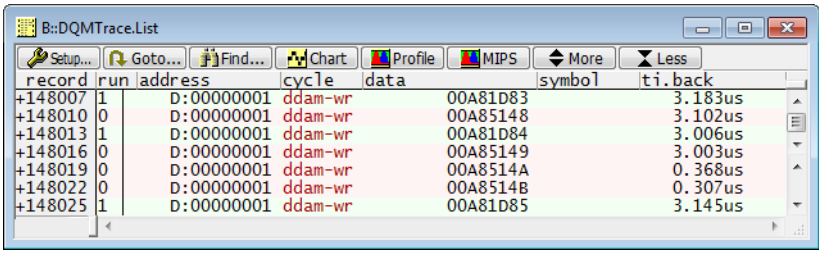
Data acquisition message

TCODE number = 7 (6 bits)
Source processor identifier (4 bits)
Identification tag from DQTAG (8 bits)
Data from DDAM (1 to 32 bits)
Timestamp (optional) (0 to 30 bits)

19 to 80 bits
in 2 to 6 trace beats

The command group **DQMTrace** is used to display and analyze the Data Acquisition Messages.

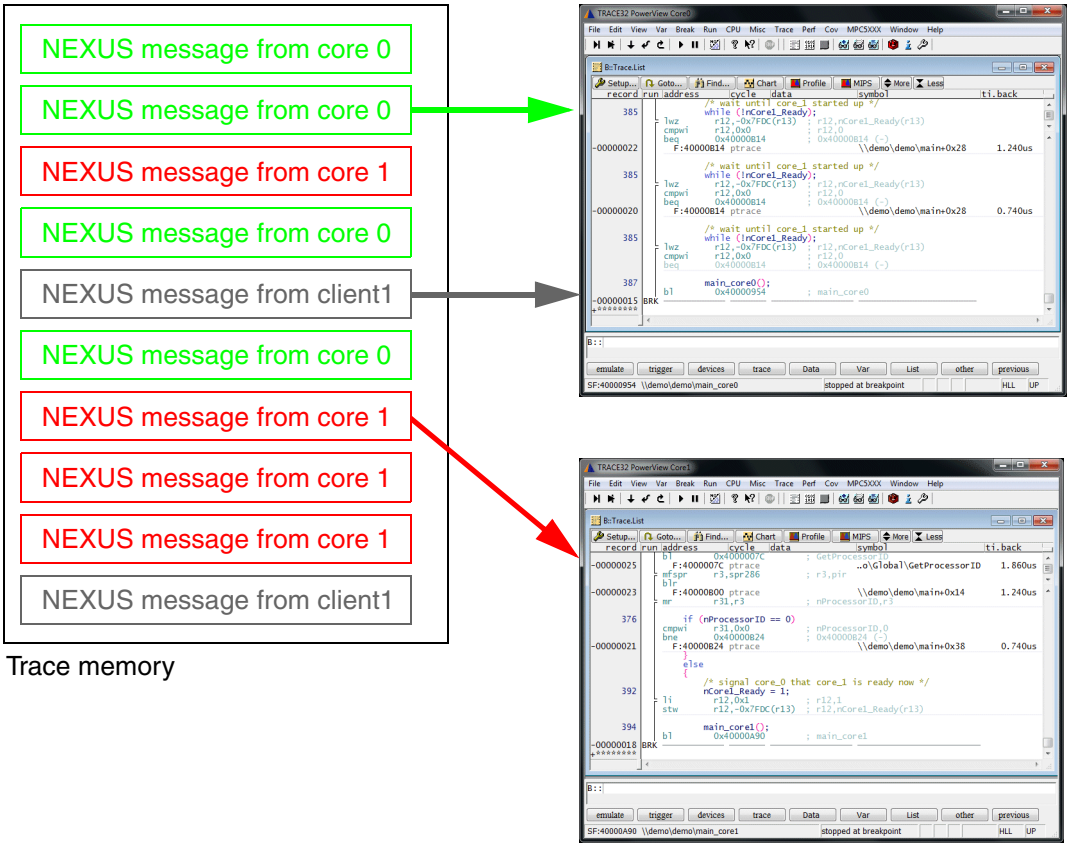
```
DQMTrace.List
```



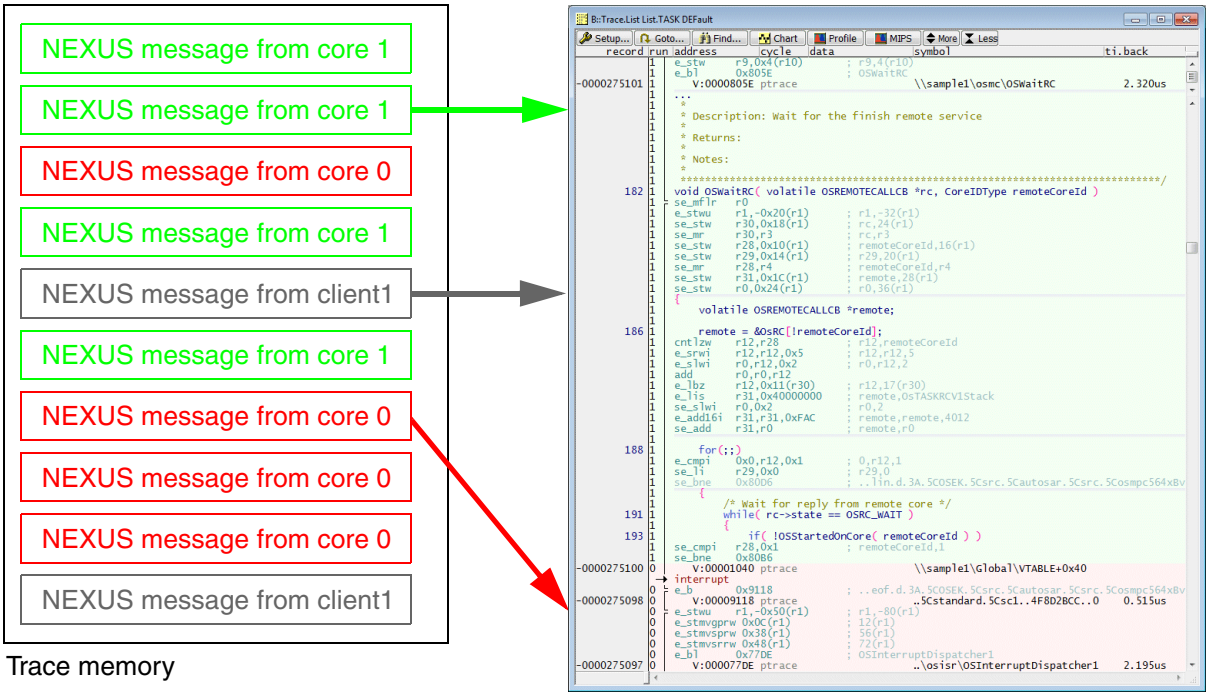
column layout	
address	Identification tag
cycle	Always “write access to DDAM”
data	Exported data
ti.back	Timestamp

AMP Tracing

Trace Display	Each TRACE32 instance analyzes and displays the trace information generated by the core(s) it controls. Trace messages generated by clients (DMA, FlexRay etc.) are assigned to the TRACE32 instance that enabled the messaging.

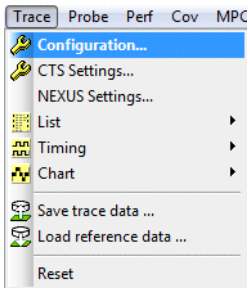


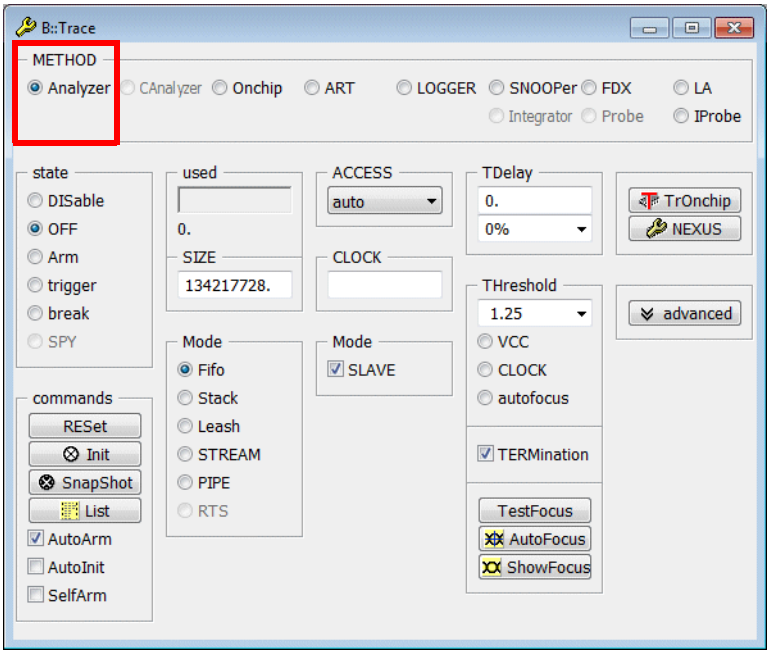
Trace Display	Trace information from all trace sources in the SMP system is displayed together.



Source for the Recorded Trace Information

If TRACE32 is started when a PowerTrace hardware and a NEXUS ADAPTER / PREPROCESSOR SERIAL is connected, the source for the trace information is the so-called **Analyzer** (**Trace.METHOD Analyzer**).



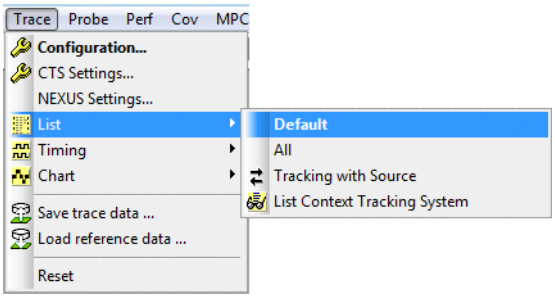


The setting **Trace.METHOD Analyzer** has the following impacts:

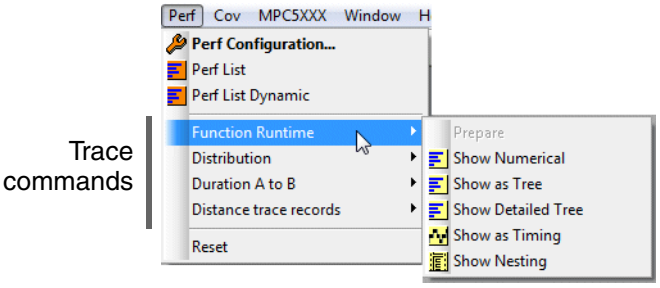
1. **Trace** is an alias for **Analyzer**.

```
Trace.List                          ; Trace.List means
                                    ; Analyzer.List

Trace.Mode Fifo                     ; Trace.Mode Fifo means
                                    ; Analyzer.Mode Fifo
```

2. All commands from the Trace menu apply to the Analyzer.



3. All Trace commands from the Perf menu apply to Analyzer.



4. TRACE32 is advised to use the trace information recorded to the Analyzer as source for the trace evaluations of the following command groups:

CTS.*<sub_cmd>*	Trace-based debugging
COVerage.*<sub_cmd>*	Trace-based code coverage
ISTAT.*<sub_cmd>*	Detailed instruction analysis
MIPS.*<sub_cmd>*	MIPS analysis
BMC.*<sub_cmd>*	Synthesize instruction flow with recorded benchmark counter information

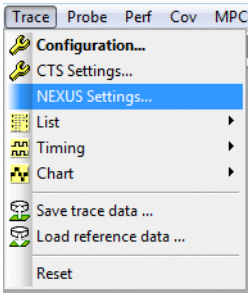
This NEXUS Training uses always the command group **Trace**. If your trace information is stored to an **on-chip trace memory**, just select the trace method Onchip and nearly all features will work as demonstrated for the trace method Analyzer

```
Trace.METHOD Onchip
```

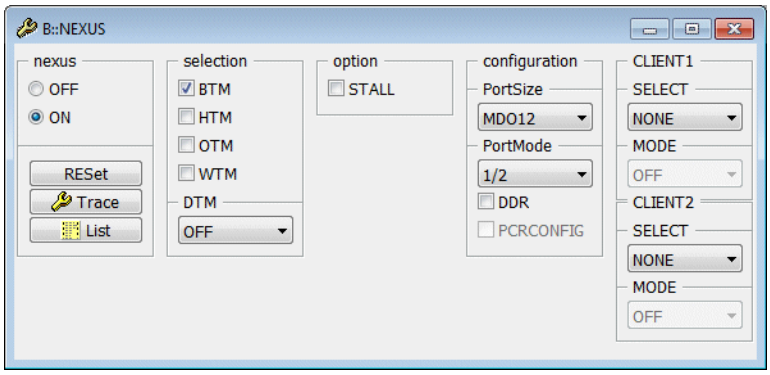
Configuration of the Trace Interface

Parallel Interface

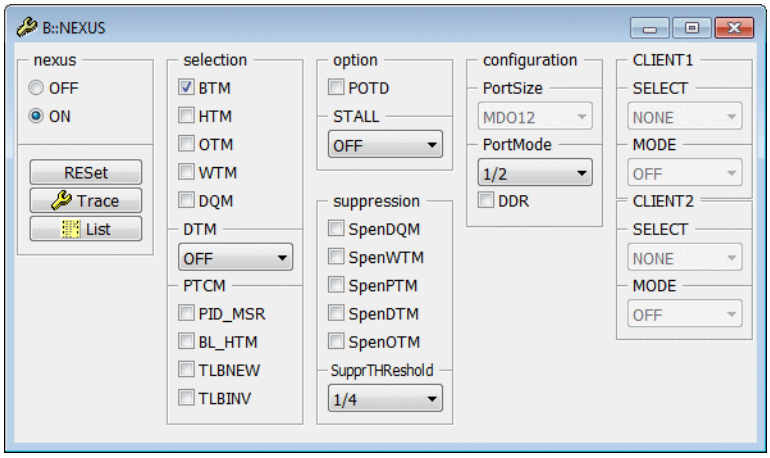
The interface configuration is done via the NEXUS window. The TRACE32 NEXUS window has a different look for IEEE-ISTO 5001™-2003, IEEE-ISTO 5001™-2008 and IEEE-ISTO 5001™-2012.



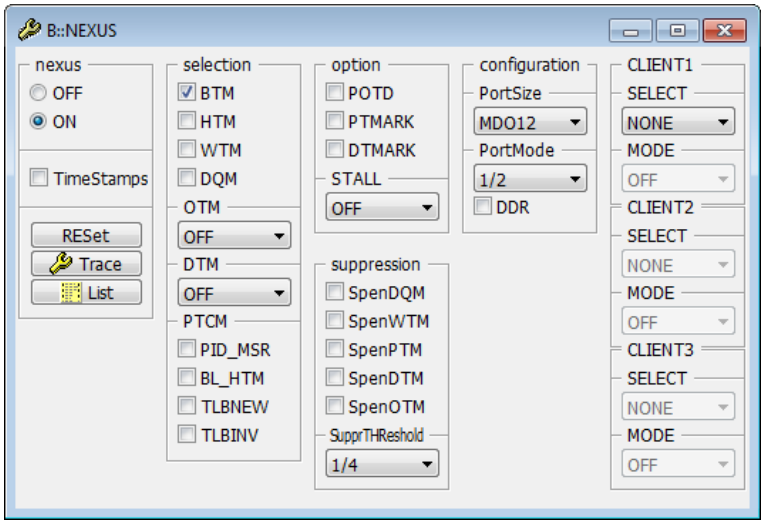
NEXUS window for IEEE-ISTO 5001™-2003



NEXUS window for IEEE-ISTO 5001™-2008



NEXUS window for IEEE-ISTO 5001™-2012

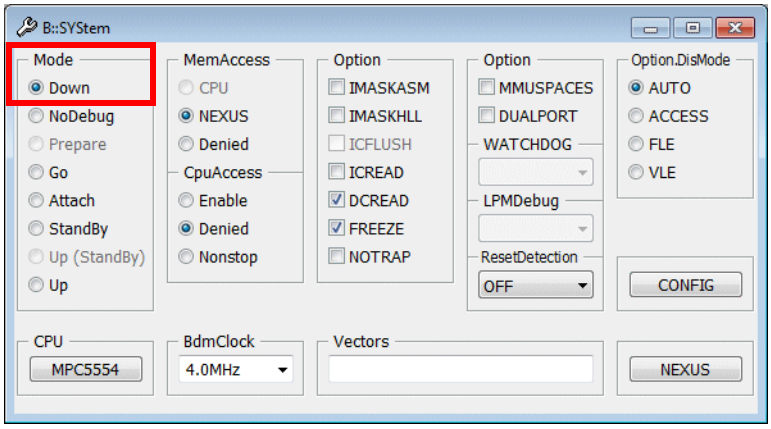


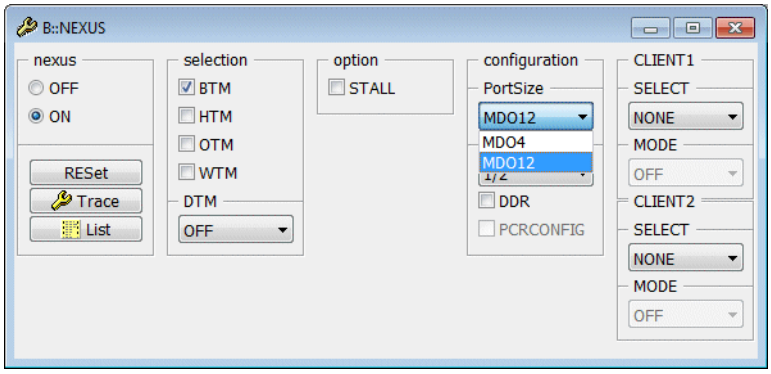

```
NEXUS.state                              ; display NEXUS window
```

The configuration for a parallel NEXUS interface is identical for all compliant standards. This is why only the simpler IEEE-ISTO 5001™-2003 is shown in the configuration examples.

Select NEXUS Port Size

Selecting the NEXUS port size is only possible if ***SYStem Mode Down*** is selected.

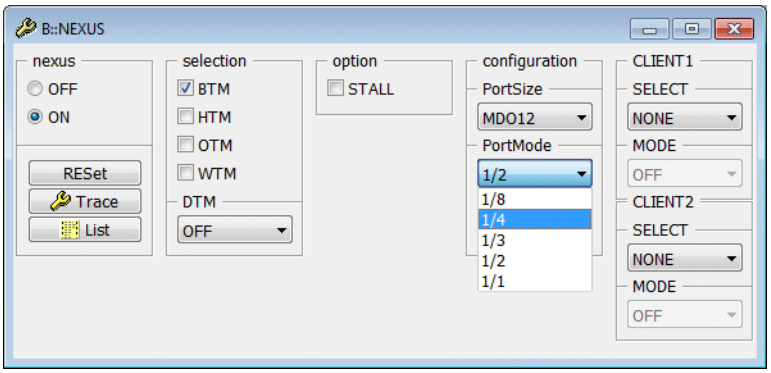




NEXUS port size	
MOD2 | MOD4 | MDO8 | MDO12 | MDO16	Specify MDO[f:0] (number of Message Data Out pins available)

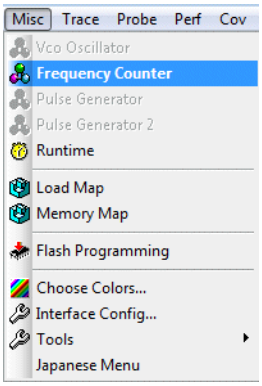
```
NEXUS.PortSize MDO12                    ; specify a trace port width of
                                        ; 12 MDOs
```

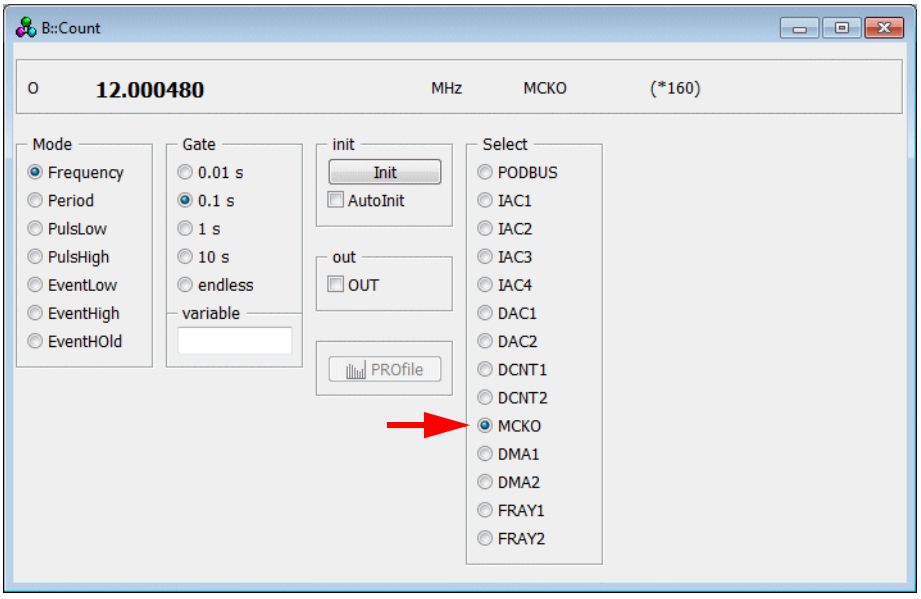
The PortMode determines the frequency of MCKO (Message ClocK Out) relative to the system clock (SYS_CLK). Max. MCKO is usually 80 MHz, please refer to the **Nexus characteristics** in the data sheet of your chip for details.

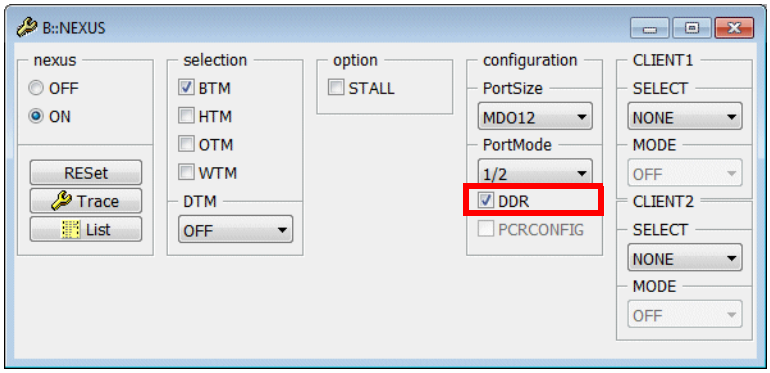



```
NEXUS.PortMode 1/4
```

To measure the MCKO frequency with TRACE32 proceed as follows:







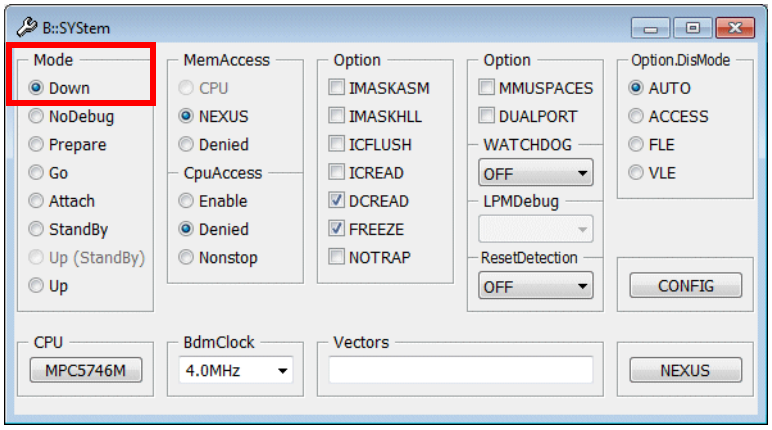
Advise the NEXUS module to export trace information on the rising and falling edge of MCKO (not supported by all chips/cores).

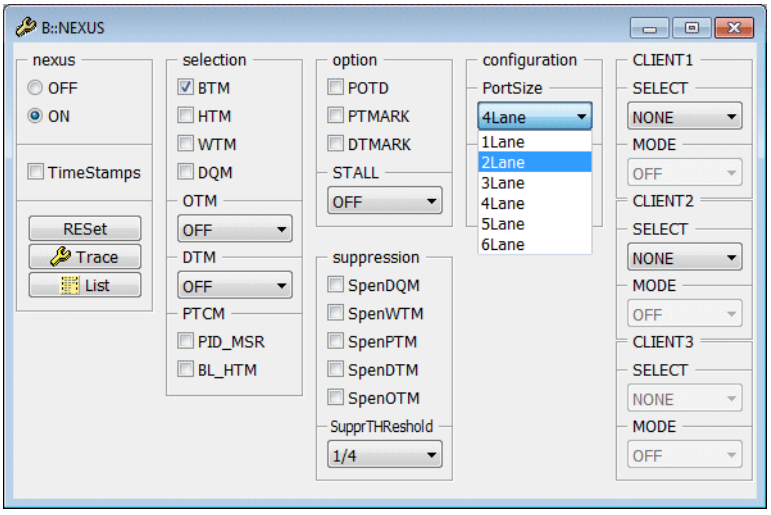
```
NEXUS.DDR ON
```

Chips with serial interface provide a Nexus module compliant to the IEEE-ISTO 5001™-2012 standard.

Select NEXUS PortSize

Selecting the NEXUS PortSize is only possible if ***SYStem Mode Down*** is selected.

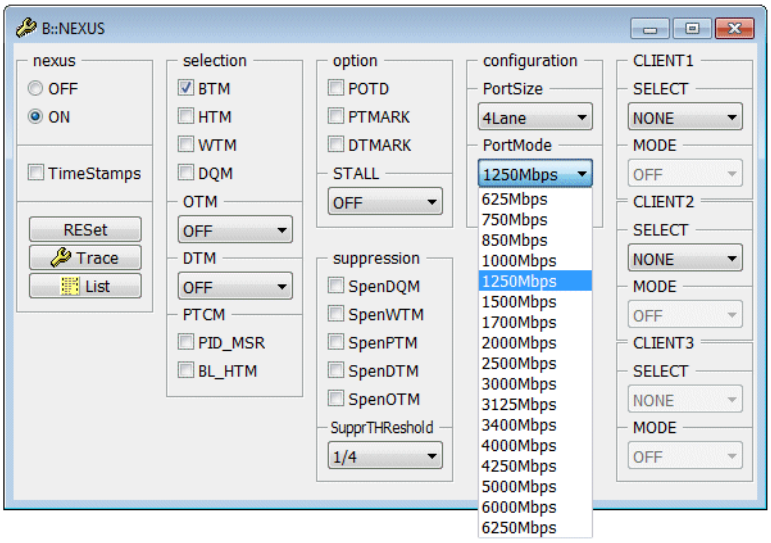




NEXUS port size	
2Lane | 4Lane	Specify number of (Aurora) lanes

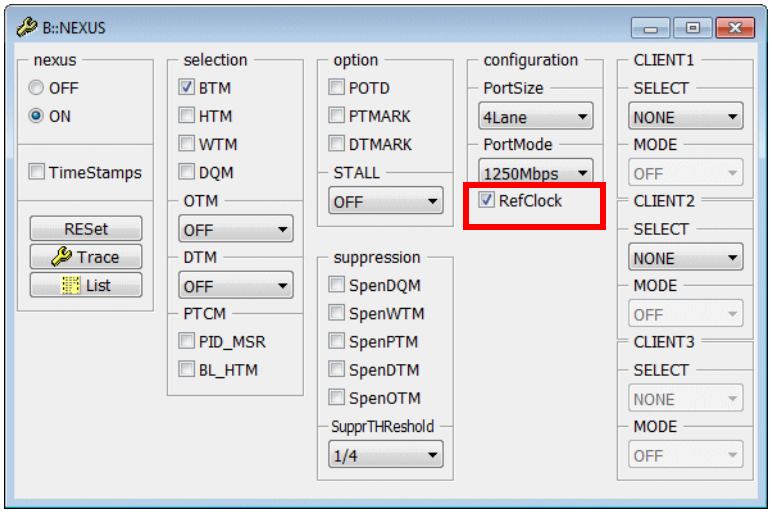
```
NEXUS.PortSize 2Lane                    ; specify a trace port with 2 lanes
```

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



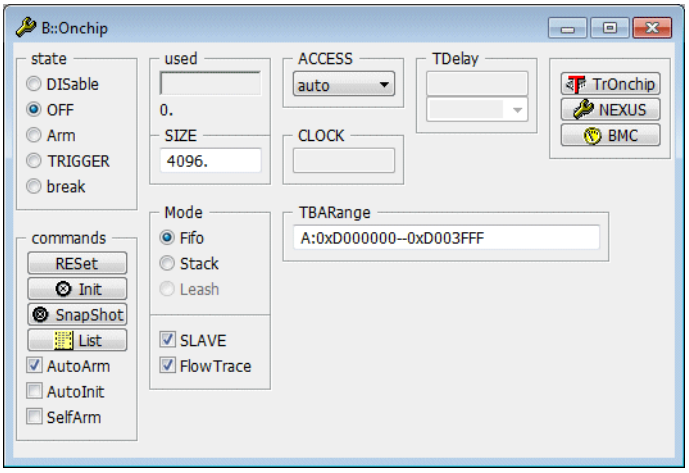

```
NEXUS.PortMode 1250Mbps
```

Automotive processors usually need an external reference clock for Aurora operation. Lauterbach´s PREPROCESSOR SERIAL can provide this clock signal. It is enabled using **NEXUS.RefClock ON.**



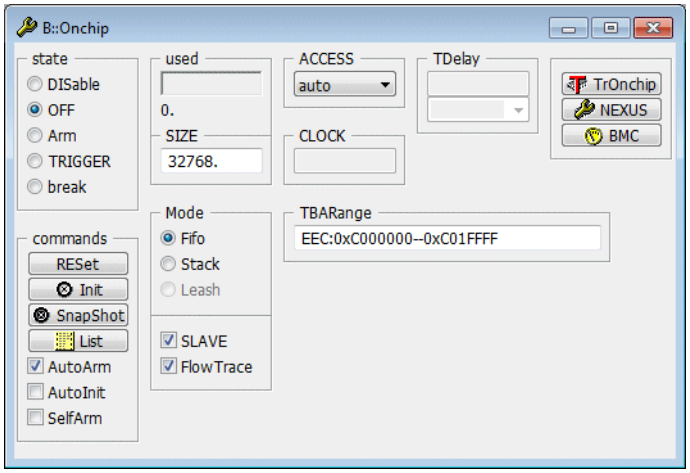
The usage of the onchip trace memory requires that trace memory is allocated.

```
; allocate trace memory for 4K NEXUS packets (packet size = 32 bit)
; A: stands for physical memory
Onchip.TBARange A:0xD000000--0xD003fff
```



Emulation devices may provide more trace memory.

```
; allocate trace memory for 32K NEXUS packets (packet size = 32 bit)
; EEC: stands for emulation device memory
Onchip.TBARange EEC:0xC000000--0xC01FFFF
```



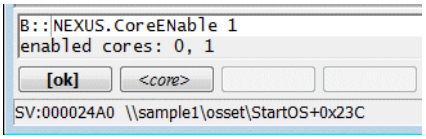
Trace information generated for multiple cores is:

- Exported via a single off-chip trace interface.
- Stored in a joint on-chip trace memory.

SMP Systems

Due to the fact that one TRACE32 instance is used to control multiple cores in an **SMP system** there is only one NEXUS configuration window, and thus no problem to keep the Nexus interface setting consistent.

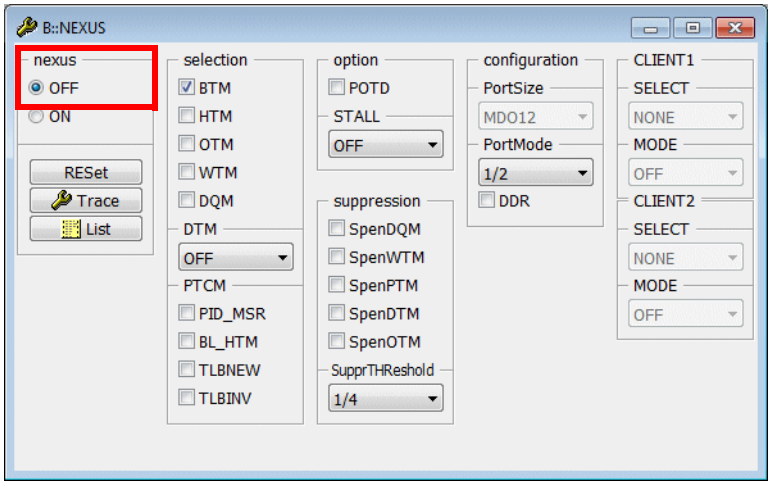
Since trace messaging from more than one core may easily generate more trace messages than the interface in use can convey, it is possible to enable the message generation only for the cores that are in the focus of the analysis.



NEXUS.CoreENable {*<logical_core>*}	Enable core tracing for listed logical cores.

The situation is different for multiple cores in an **AMP system**. Here each core is controlled by its own TRACE32 instance, each with its own NEXUS configuration window. Since the TRACE32 Resource Management does not keep the Nexus interface settings in multiple TRACE32 instances consistent, this is the job of the user.

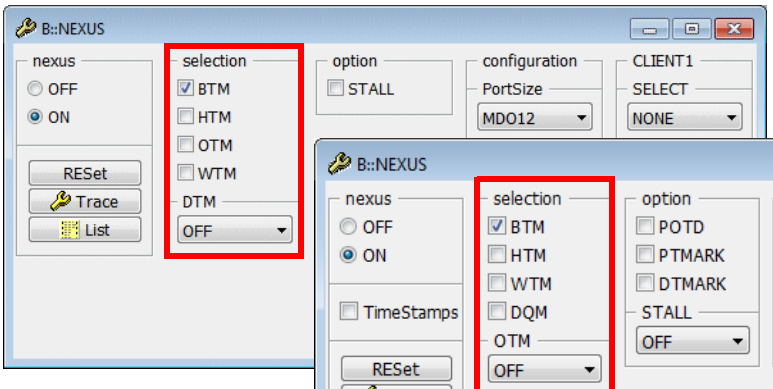
Since trace messaging from more than one core may easily generate more trace messages than the interface in use can convey, it is recommended to disable the message generation for core(s) that are not in the focus of the analysis.

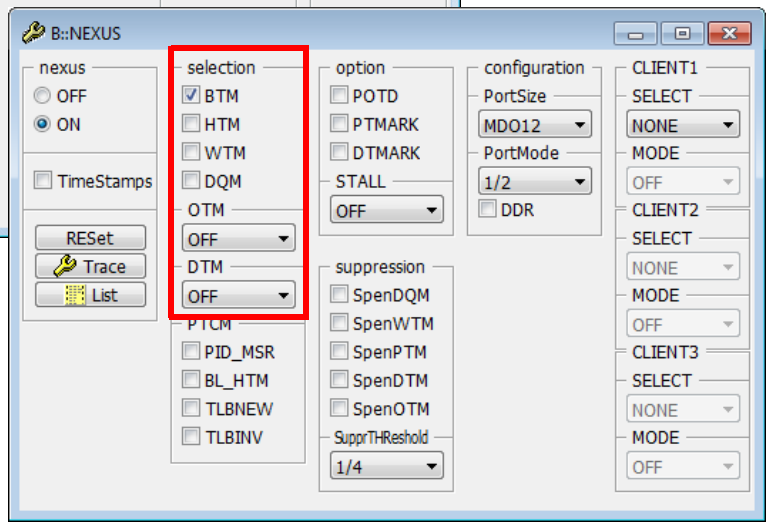



```
NEXUS.OFF
```

Basic Messages

NEXUS window for IEEE-ISTO 5001™-2003



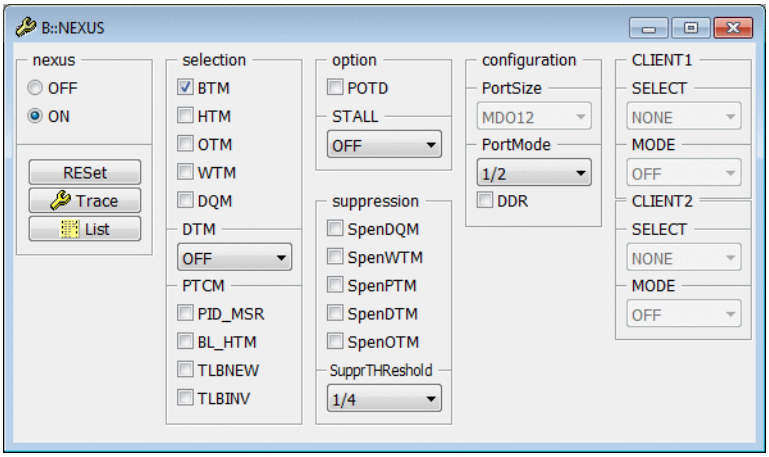


NEXUS window for IEEE-ISTO 5001™-2012

Messages	
BTM ON	Enable Branch Trace Messages.
BTM ON + HTM ON	Enable Indirect Branch History Messages.
OTM ON (2003/2008 Standard) **OTM PID0** (2012 Standard) **OTM NPIDR** (2012 Standard)	Enable Ownership Trace Messages via 8-bit PID0. Enable Ownership Trace Messages via 8-bit PID0. Enable Ownership Trace Messages via 32-bit NPIDR.
WTM ON	Enable Watchpoint Hit Messages. Watchpoint Hit Messages are usually not used by the user. TRACE32 enables them automatically, if they are needed.

DTM Read **DTM Write** **DTM ReadWrite** **DTM IFETCH**	Enable Data Read Messages. Enable Data Write Messages. Enable Data Read and Write Messages. Instruction fetches are exported as Data Read Messages.
	The basic idea of the **Limited** settings is to exclude stack read/writes from the message generation and thus avoid bandwidth problems.
DTM ReadLimited (2012 Standard)	Enable Data Read Messages, but exclude read accesses using GPR R1 in effective address computations.
DTM WriteLimited (2012 Standard)	Enable Data Write Messages, but exclude write accesses using GPR R1 in effective address computations.
DTM ReadWriteLimited (2012 Standard)	Enable Data Read and Write Messages, but exclude read/write accesses using GPR R1 in effective address computations.

```
NEXUS.BTM ON                          ; enable Branch Trace Messages

NEXUS.DTM ReadWrite                   ; enable Data Trace Messages for
                                      ; both read and write operations

NEXUS.OTM NPIDR                       ; enable Ownership Trace Messages
                                      ; via NPIDR register
```



Data Acquisition Messages

Messages	
DQM ON	Enable Data Acquisition Messages.

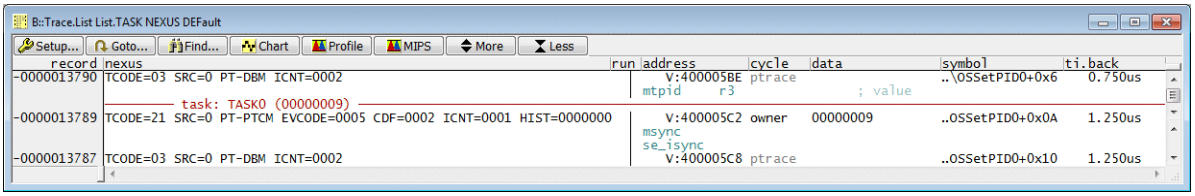
When a write to the PID or NPIDR register occurs, a Program Trace Correlation Message can be generated instead of an Ownership Trace Message.

The Program Trace Correlation Message contains the address of the instruction that wrote the OS-related information to the PID or NPIDR register and the OS-related information itself.

This has the following advantages:

- **Trace.List:** the OS-related information can be directly assigned to instruction that wrote to the PID or NPIDR register.
- **Trace.STATistic.Func:** the accuracy of all task-aware function run-time measurements is improved.

OTM ON **PID_MSR ON** (2008 Standard) or **OTM PID0** **PID_MSR ON** (2012 Standard) or **OTM NPIDR** **PID_MSR ON** (2012 Standard)	Enable Program Trace Correlation Messages for Ownership tracing.
POTD	Periodic Ownership Trace message Disable. OFF: Periodic Ownership Trace Message is enabled (default). ON: Periodic Ownership Trace Message is disabled. Recommended if PID0 register is used.

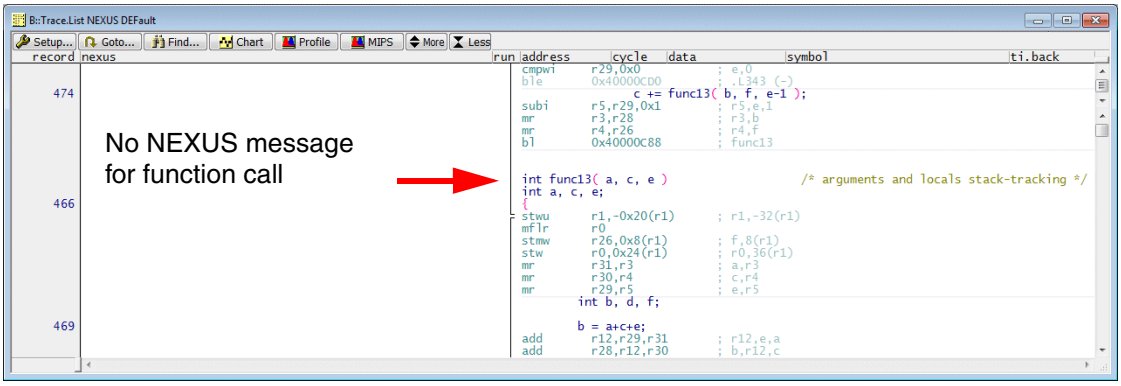



```
NEXUS.OTM NPIDR

NEXUS.PID_MSR ON

NEXUS.POTD ON
```

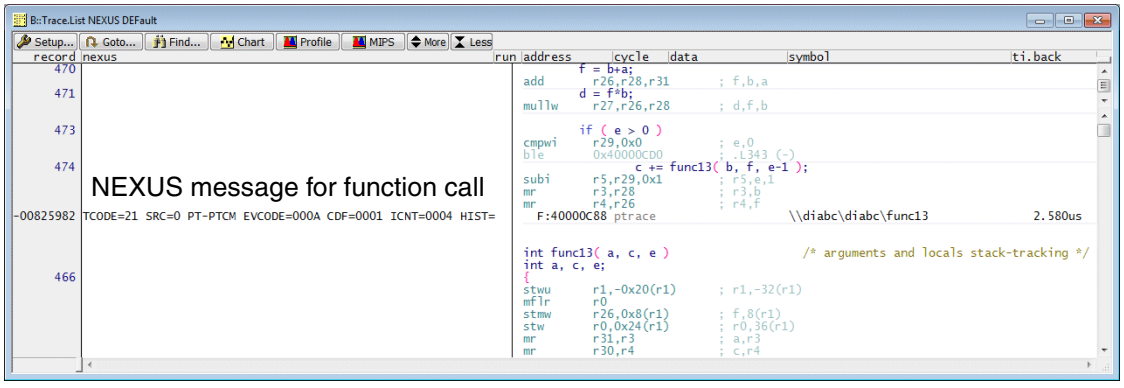
A Program Trace Correlation Message is generated when a direct branch function call (bl/bcl/bla/bcla) occurred while Indirect Branch History messaging is used.

BL_HTM	Program Trace Correlation Message is generated on a direct branch function call (for NEXUS.HTM ON only).

```
NEXUS.BTM ON

NEXUS.HTM ON
```




```
NEXUS.PTCM BL_HTM ON
```

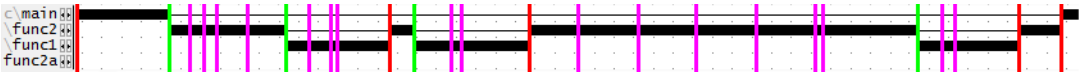


Enabling Program Trace Correlation Messages for direct branch function calls allows the optimum message generation for function run-time measurements. The screenshots below show this for the TRACE32 command **Trace.Chart.Func**.

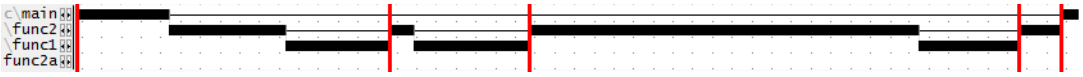
Legend:

- I: Indirect Branch Message generated for function exits (“BLR”), function pointers, interrupts etc.
- I: Direct Branch Message generated for function calls (opcode “BL”)
- I: Direct Branch Message generated for conditional branches

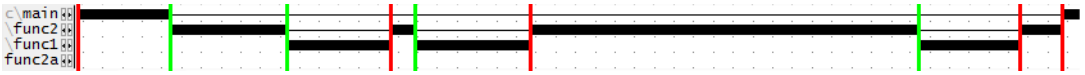
BTM ON
more trace messages are generated than required.



BTM ON + HTM OM
too little trace messages are generated for an accurate run-time measurement.



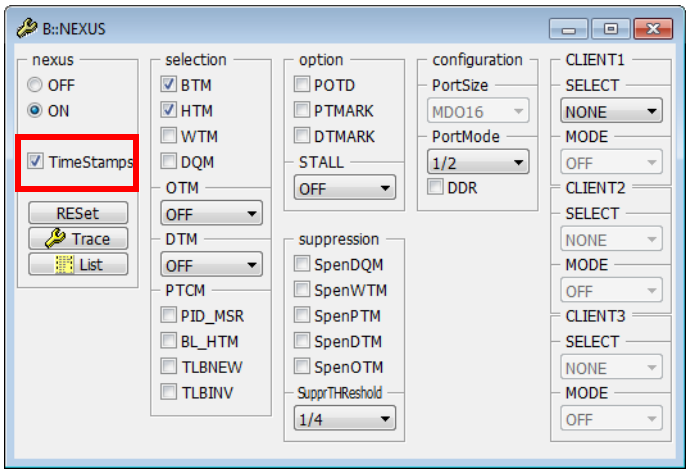
BTM ON + HTM ON + BL_HTM ON
an optimum number of trace messages is generated for an accurate run-time measurement.



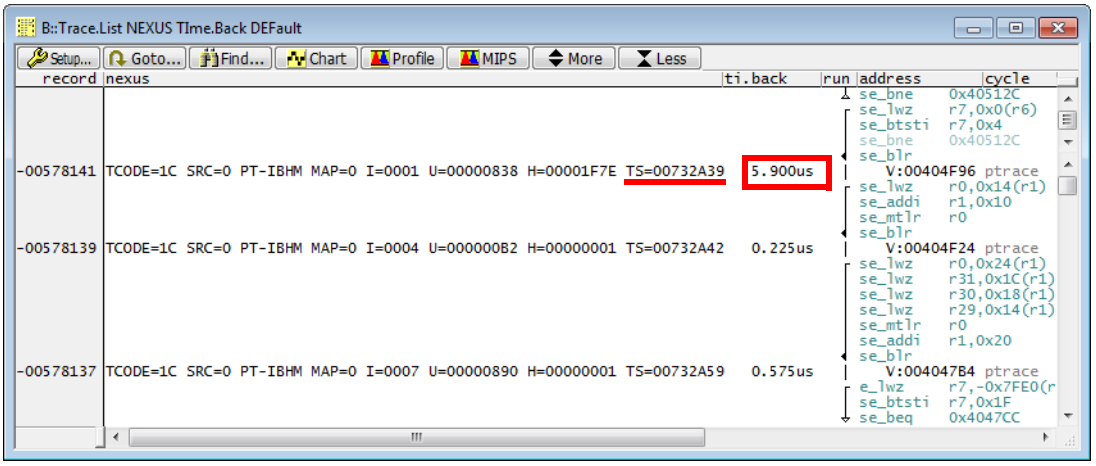
The Nexus Module implemented on the MPC57xx/SPC57x is able to add a timestamp field to the Nexus messages. The timestamp value is applied to the messages as they enter the Nexus message queues.

To use this feature proceed as follows:

1. Check TimeStamps in the NEXUS configuration window.

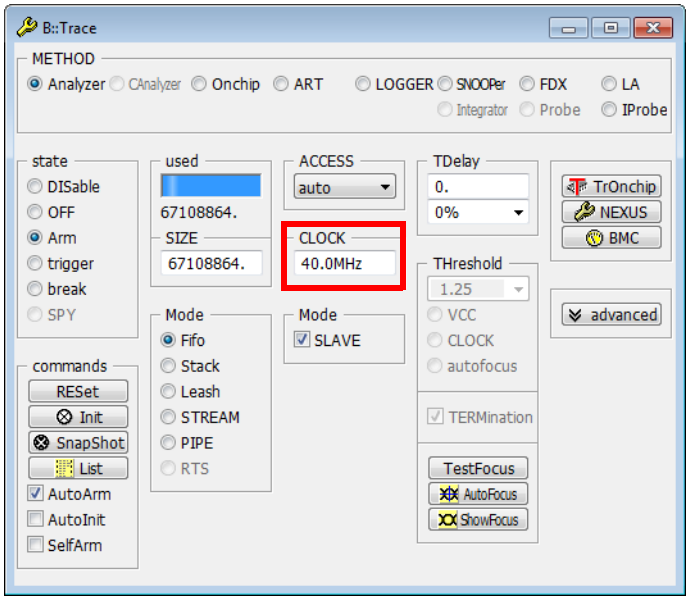



```
NEXUS.TimeStamps ON
```



TRACE32 calculates its trace time information (ti.back) out of the values of the Nexus timestamp field (TS=). To calculate the time information TRACE32 needs to know the core clock frequency.

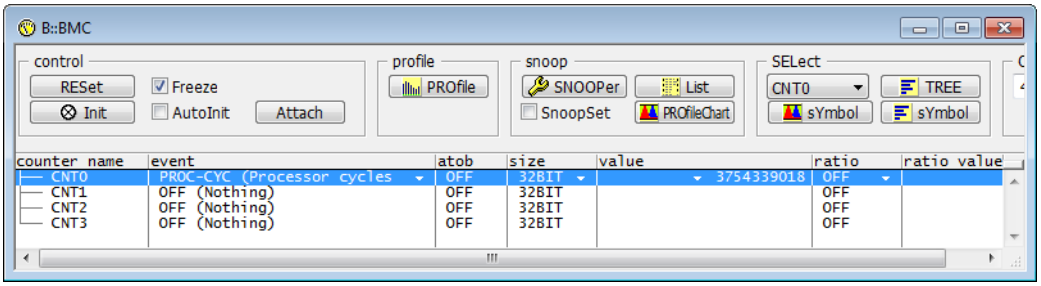
2. Inform TRACE32 about the core clock frequency.

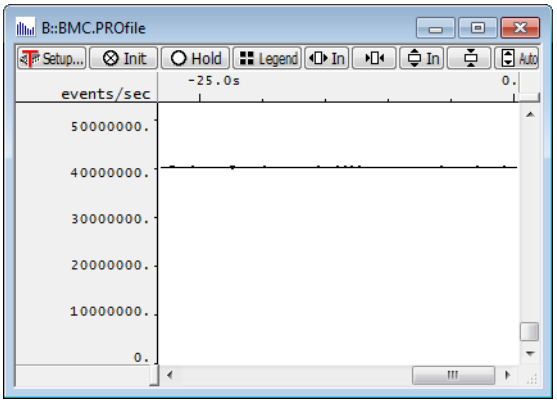



```
Trace.CLOCK 40.MHz
```

Trace.CLOCK *<freq>*	Specify core clock frequency.
Trace.CLOCK {*<freq>*}	The core clock frequency can be set per core in an SMP system.

If you do not know your core clock frequency you can measure it as follows:

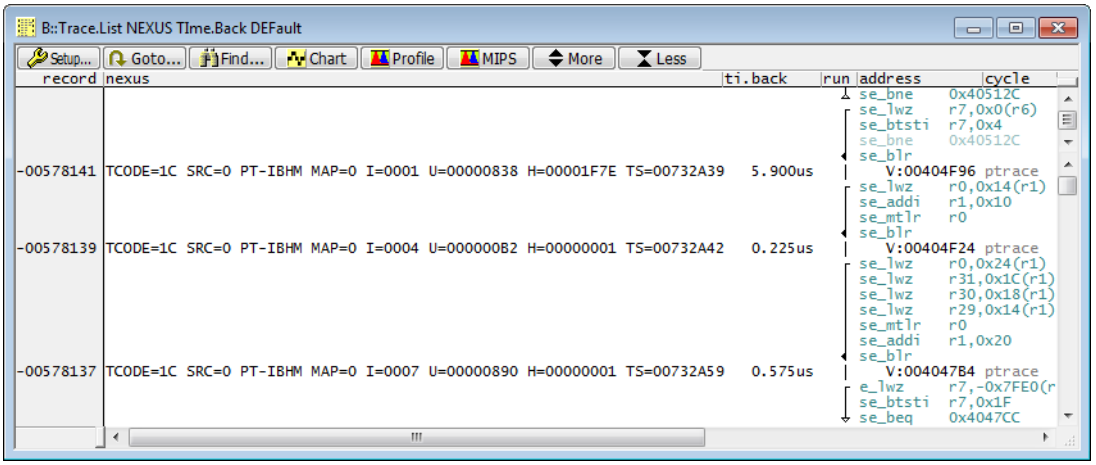





```
BMC.CNT0.EVENT PROC-CYC          ; configure CNT0 to count processor
                                 ; cycles

BMC.PROfile                      ; display the frequency of CNT0
```

3. If all configurations are done start and stop the program execution.

4. Display the result.



Adding timestamp information to Nexus messages has the following advantages:

- The time is more precise, because it is added at the trace source. Parallel execution is clearly visible.
- Nexus timestamps are the only way to get time information for trace-to-memory (onchip trace).
- Nexus timestamps solve some issues of the serial trace recording.

But Nexus timestamps have also disadvantages:

- They need additional bandwidth (approx. 20%).
- The TRACE32 trace decoding becomes slower, since the time information has to be calculated for the complete recording before it can be displayed (Tracking).
- Since TRACE32 uses a fixed core frequency to calculate trace time information out of the Nexus timestamps, this calculation is not possible for variable clock frequencies.

  To display only the Nexus timestamp information in the trace display, use the following command:

  ```
  Trace.List Nexus CLOCKS.Back DEFault TIme.Back.OFF
  ```

- It may happen, that not all cores in a chip provide the ability to generate Nexus timestamps.

If NEXUS TimeStamps is unchecked, the TRACE32 tool timestamp mechanism is used. This means a Nexus message is timestamped after it is completely received and stored into the trace memory of the TRACE32 tool. The TRACE32 tool timestamp has a resolution of 20ns for POWERTRACE/ETHERNET or POWERTRACE PX and 5 ns for POWERTRACE II / POWERTRACE III. The time is less precise, because it is added at the trace sink. The merging of the parallel trace streams to a single serial trace stream and the TRACE32 recording logic are the main reasons that make TRACE32 tool timestamp less precise.

SMP systems: Due to the fact that one TRACE32 instance is used to control all cores of the SMP system, the message setup is identical for all controlled cores.

AMP systems: Due to the fact that one TRACE32 instance is used per core, an individual message setup per core is possible.

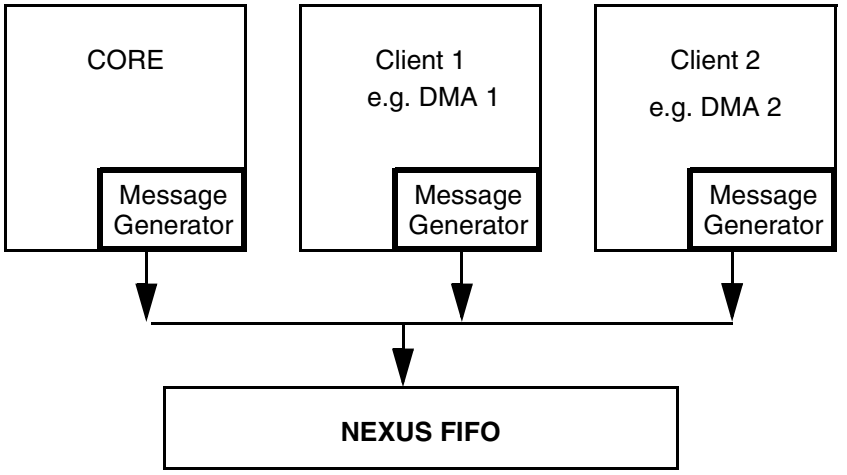
NEXUS Trace Clients

A MPC5xxx/SPC5xxx core can provide several models for the trace clients.

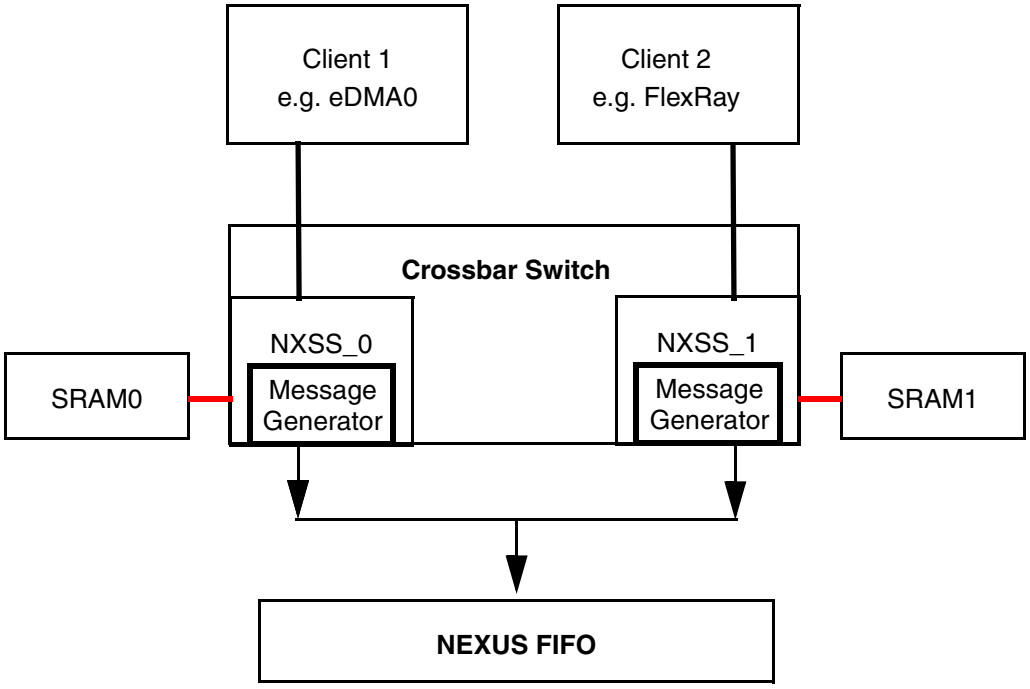
Trace Client Types

Dedicated Trace Clients

Each client that can generate NEXUS messages has its own Message Generator.

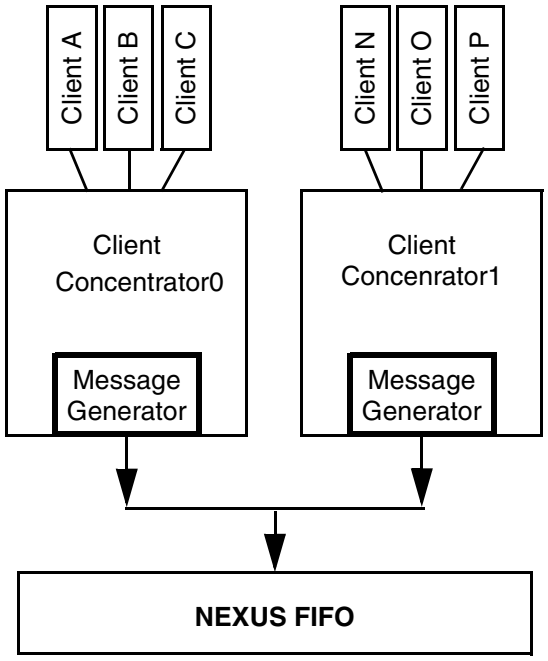


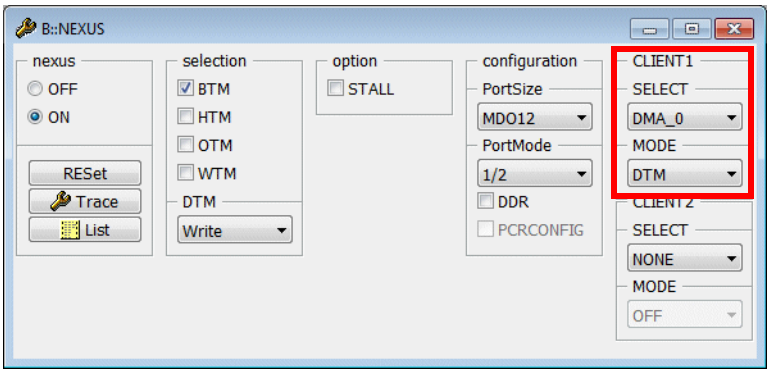
A port sniffers is used to generate the NEXUS messages for the selected clients.



Please be aware, that the NXSS (Nexus Crossbar Slave Port Data Trace Module) can only snoop read/write accesses from the selected trace client to the connected SRAM.

The Message Generator of the Client Concentrator generates NEXUS messages for the connected clients. Clients can be enabled independently.






```
NEXUS.CLIENT1 SELECT DMA              ; specify DMA for CLIENT1

NEXUS.CLIENT1 MODE DTM                ; NEXUS messages are generated
                                      ; according to the DTM settings
                                      ; (here write)
```

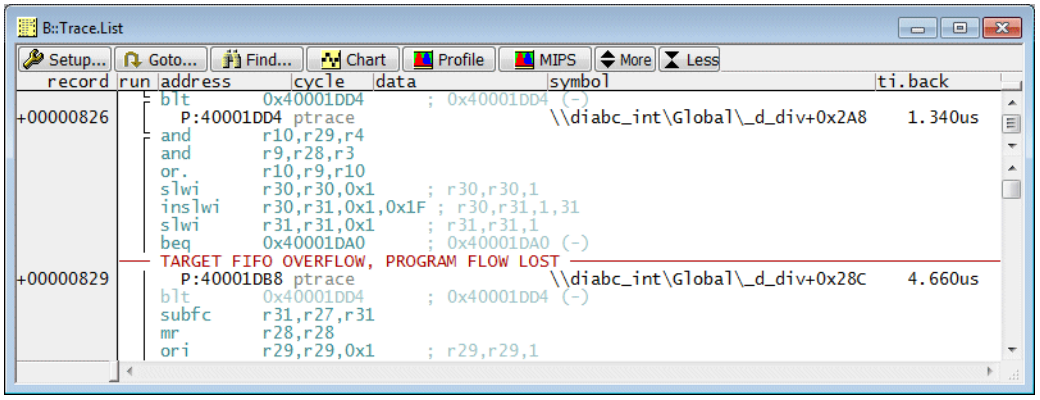
Possible clients:

DMA/DMA2	DMA controller
ETHERNET	Ethernet controller
FLEXRAY	FlexRay controller
PDI	Parallel Digital Interface controller
…	

Possible modes:

OFF	No NEXUS messages are generated.
Read	Generate NEXUS messages for all read accesses.
Write	Generate NEXUS messages for all write accesses.
ReadWrite	Generate NEXUS messages for all read and write accesses.
DTM	The client is using the DTM settings.

Target FIFO Overflow

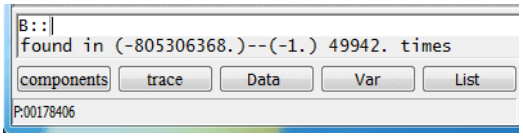


TARGET FIFO OVERFLOW, PROGRAM FLOW LOST occurs, when so much trace information is generated that it can not be buffered in the NEXUS FIFO.

In order to get an immediate display of the trace contents TRACE32 uploads only the currently displayed section from the physical trace memory to the host. To check if there are FIFOFULLs it is recommended to upload the complete trace contents to the host by the command: **Trace.FLOWPROCESS.**

The number of FIFOFULL is printed to the TRACE32 state line as result of the following command:

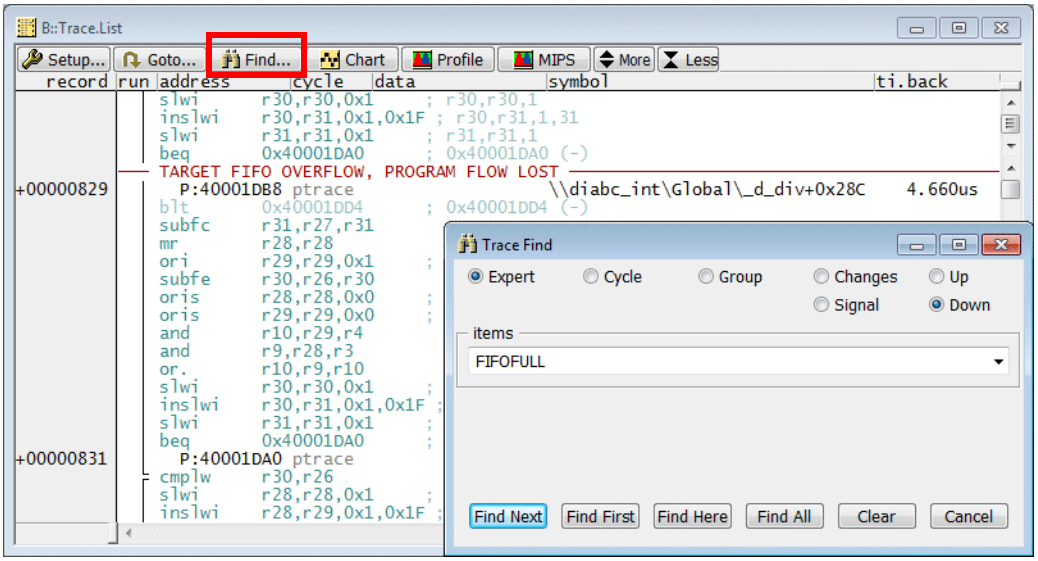
```
Trace.Find FIFOFULL /ALL
```



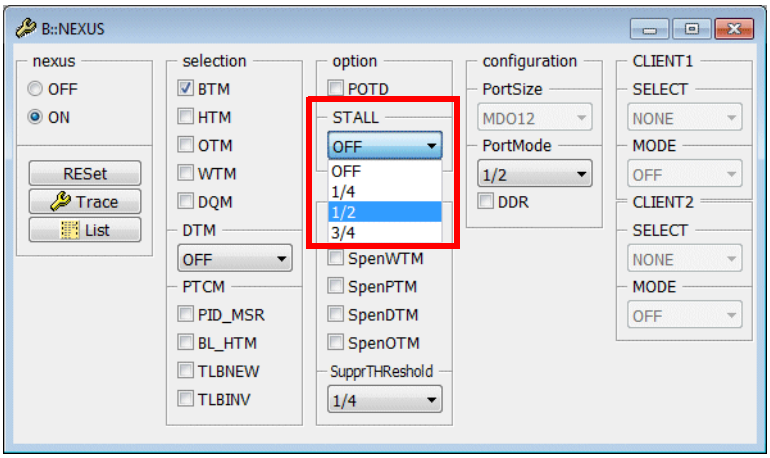
The following TRACE32 functions allows you to process the result in a script:

```
FOUND.COUNT()
```

The single FIFOFULLs can be found in the trace:



FIFOFULLs may occur during your tests, they are not errors. But FIFOFULLs may disturb your trace analysis. There are various strategies to avoid FIFOFULLs.

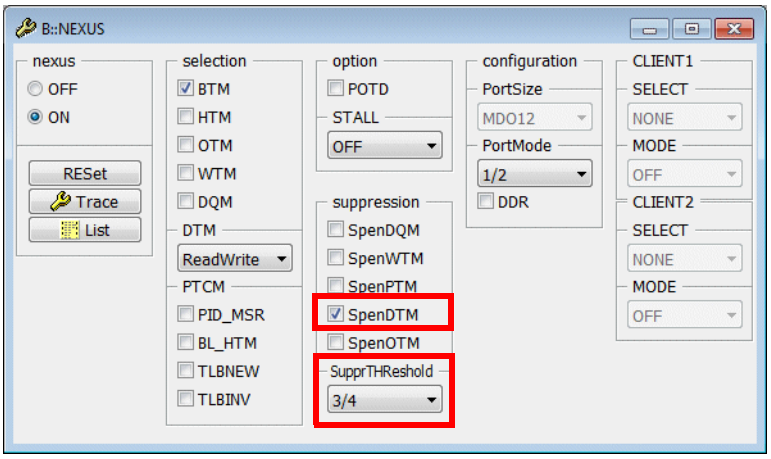


Overrun control	
STALL OFF	Generate overrun message when a new message can not be queued due to the NEXUS FIFO being full. No new message is queued to the NEXUS FIFO until it is completely empty.
STALL ON (2003 Standard)	Stall the program execution whenever the on-chip NEXUS FIFO threatens to overflow.
STALL 1/4 (2008/2012 Standard)	Stall the program execution when 1/4 of the on-chip NEXUS FIFO is filled.
STALL 1/2 (2008/2012 Standard)	Stall the program execution when 1/2 of the on-chip NEXUS FIFO is filled.
STALL 3/4 (2008/2012 Standard)	Stall the program execution when 3/4 of the on-chip NEXUS FIFO is filled.

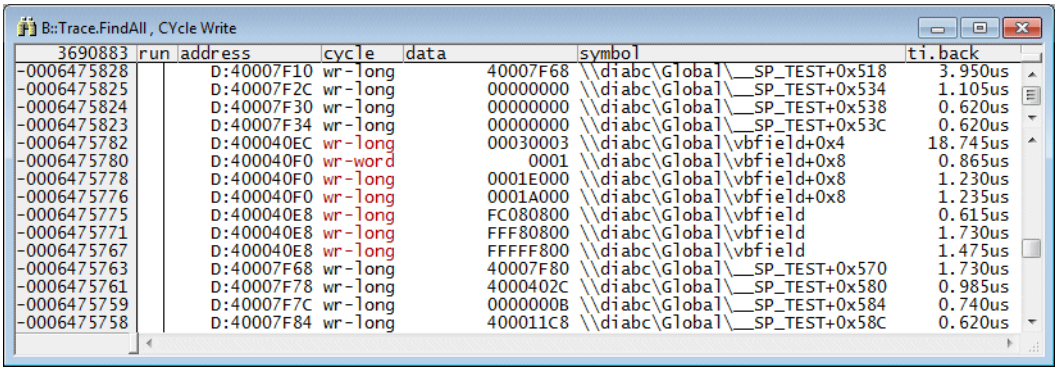
```
NEXUS.STALL 3/4
```

Suppress Data Trace Messages on Overflow Threat

Since Data Trace Messages are high-risk for NEXUS FIFOs getting full, it may be helpful to suppress these messages when the NEXUS FIFO reaches a certain fill level.




```
NEXUS.DTM ReadWrite

NEXUS.SupprTHReshold 3/4     ; Sets the NEXUS FIFO fill level, at which
                             ; messages will be suppressed

NEXUS.SpenDTM ON             ; Suppress Data Trace Messages when the
                             ; NEXUS FIFO reaches the specified filling
                             ; level

…                            ; Start and stop the program execution

Trace.FindAll,CYcle Write    ; Search for all write accesses
```



3690883	run	address	cycle	data	symbol	ti.back
-0006475828		D:40007F10	wr-long	40007F68	\\diabc\Global__SP_TEST+0x518	3.950us
-0006475825		D:40007F2C	wr-long	00000000	\\diabc\Global__SP_TEST+0x534	1.105us
-0006475824		D:40007F30	wr-long	00000000	\\diabc\Global__SP_TEST+0x538	0.620us
-0006475823		D:40007F34	wr-long	00000000	\\diabc\Global__SP_TEST+0x53C	0.620us
-0006475782		D:400040EC	wr-long	00030003	\\diabc\Global\vbfield+0x4	18.745us
-0006475780		D:400040F0	wr-word	0001	\\diabc\Global\vbfield+0x8	0.865us
-0006475778		D:400040F0	wr-long	0001E000	\\diabc\Global\vbfield+0x8	1.230us
-0006475776		D:400040F0	wr-long	0001A000	\\diabc\Global\vbfield+0x8	1.235us
-0006475775		D:400040E8	wr-long	FC080800	\\diabc\Global\vbfield	0.615us
-0006475771		D:400040E8	wr-long	FFF80800	\\diabc\Global\vbfield	1.730us
-0006475767		D:400040E8	wr-long	FFFFF800	\\diabc\Global\vbfield	1.475us
-0006475763		D:40007F68	wr-long	40007F80	\\diabc\Global__SP_TEST+0x570	1.730us
-0006475761		D:40007F78	wr-long	4000402C	\\diabc\Global__SP_TEST+0x580	0.985us
-0006475759		D:40007F7C	wr-long	0000000B	\\diabc\Global__SP_TEST+0x584	0.740us
-0006475758		D:40007F84	wr-long	400011C8	\\diabc\Global__SP_TEST+0x58C	0.620us

The NEXUS protocol does not indicate the message suppression. But read/write cycles that can not be assigned to its instruction (displayed in red) are a good indicator, the a message suppression occurred.

If you do not want to stall the program execution or suppress messages, just reduce the number of the generated trace messages:

- Enable HTM (**NEXUS.BTM** ON and **NEXUS.HTM** ON)
- Switch DTM to off when possible (**NEXUS.DTM** OFF).
- Disable the NEXUS message generation for cores you are not interested in for you current analysis.
- Filter the DTMs. Refer to **"Filter and Trigger (Core) Overview"**, page 101 for details.

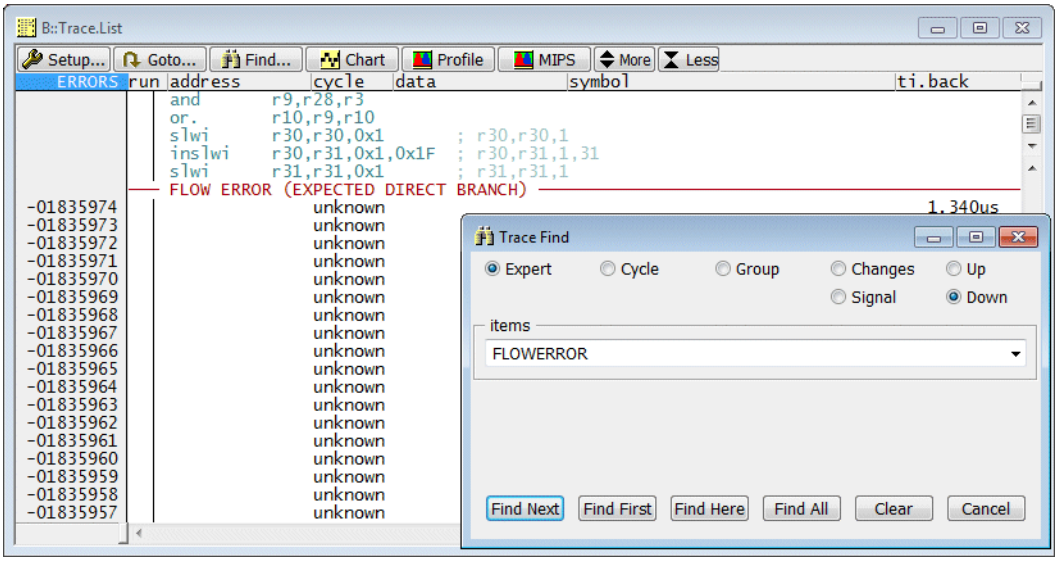
TRACE32 indicates FLOWERRORS:

- If the instruction flow information generated by NEXUS does not match with the code image in the target.
- If invalid NEXUS messages are generated.

The complete number of FLOWERRORS is printed to the TRACE32 state line as result of the following command:

```
Trace.Find FLOWERROR /ALL
```

The single FLOWERROR can be found in the trace:

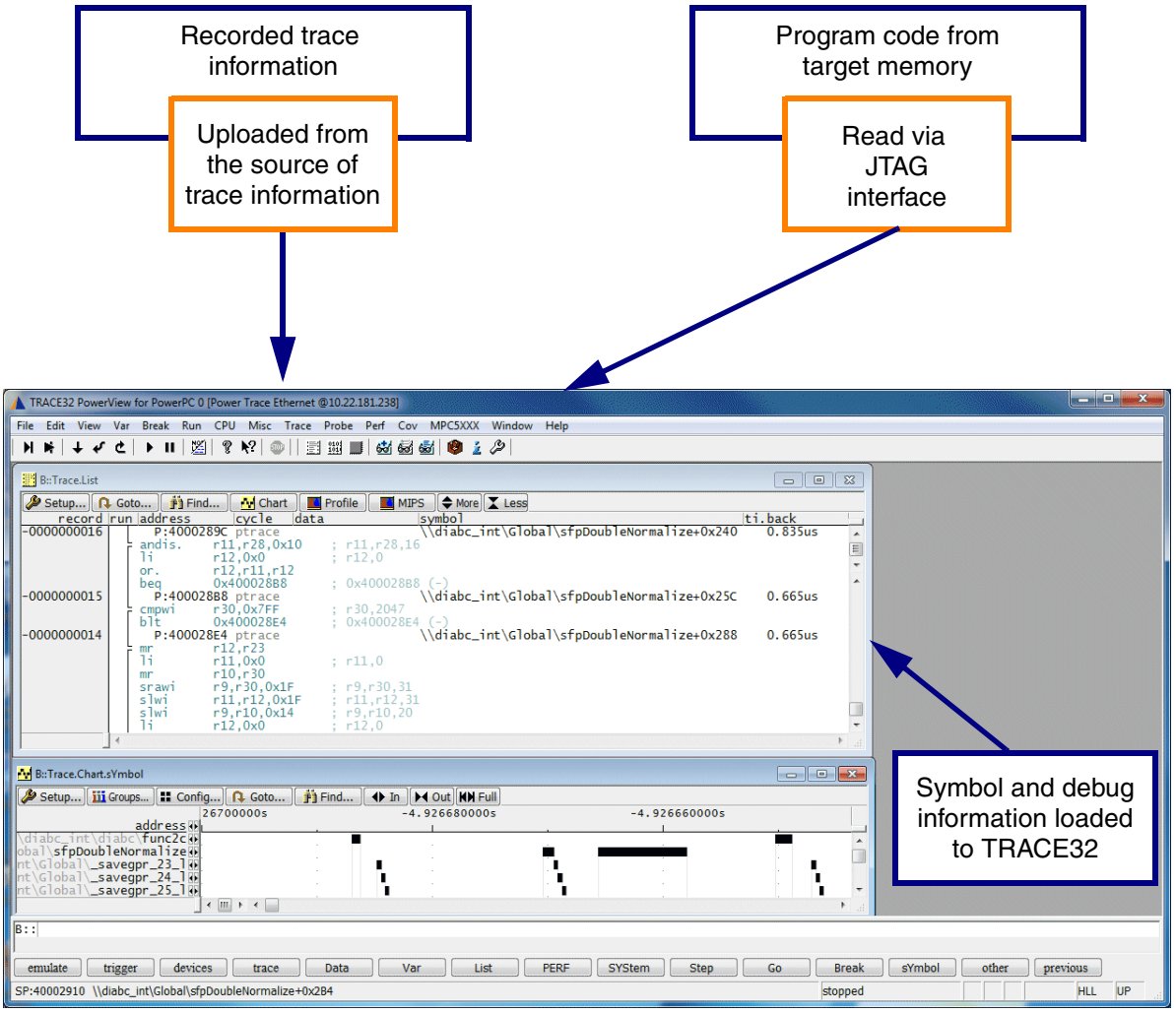


FLOWERRORs are errors and it is recommended to fix them. Please contact your local Lauterbach support if you need assistance.

Sources of Information for the Trace Display

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

- The trace information recorded.
- The program code from the target memory read via the JTAG interface.
- The symbol and debug information already loaded to TRACE32.



Settings in the Trace Configuration Window

The main influencing factor on the trace information is the NEXUS hardware module. It specifies what type of trace information is generated for the user.

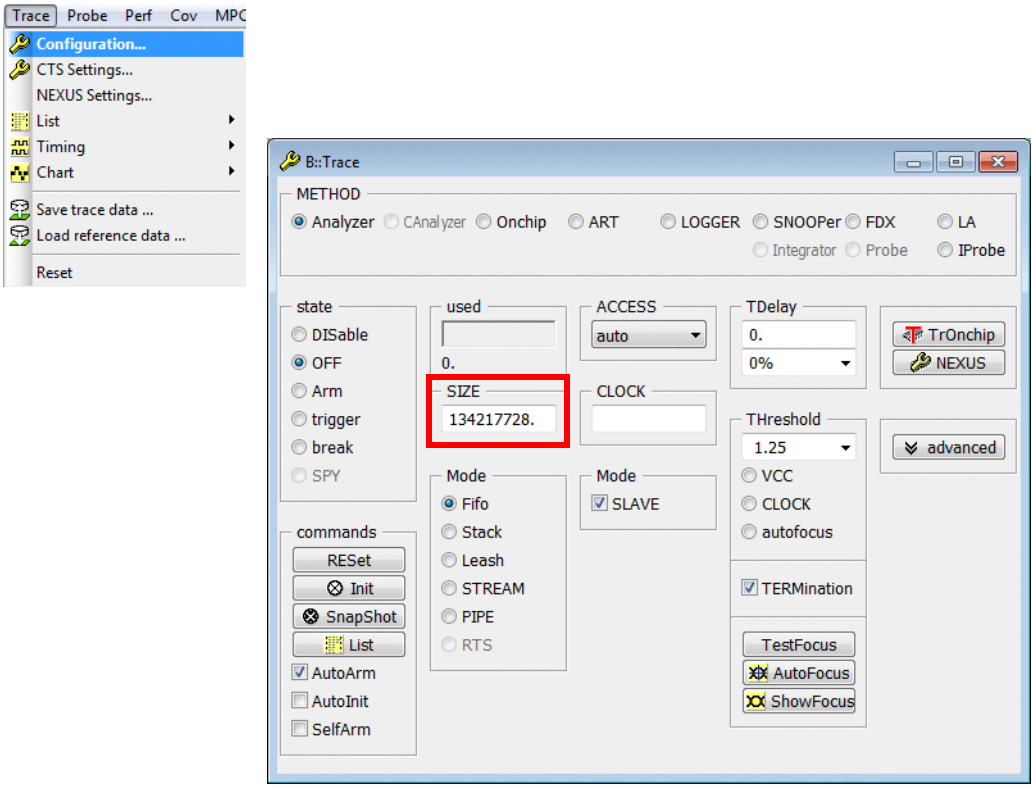
Another important influencing factor are the settings in the TRACE32 Trace Configuration window. They specify how much trace information can be recorded and when the trace recording is stopped.

Recording Modes

The **Mode** settings in the Trace configuration window specify how much trace information can be recorded and when the trace recording is stopped.

The following modes are provided:

- **Fifo, Stack, Leash Mode:** allow to record as much trace records as indicated in the **SIZE** field of the Trace Configuration window.



- **STREAM Mode (PowerTrace II hardware only):** STREAM mode specifies that the trace information is immediately streamed to a file on the host computer. STREAM mode allows a trace memory size of several T Frames.

- **PIPE Mode (PowerTrace II hardware only):** PIPE mode specifies that the trace information is immediately streamed to a named pipe on the host computer.

  PIPE mode creates the path to convey trace raw data to an application outside of TRACE32 PowerView. The named pipe has to be created by the receiving application before TRACE32 can connect to it.

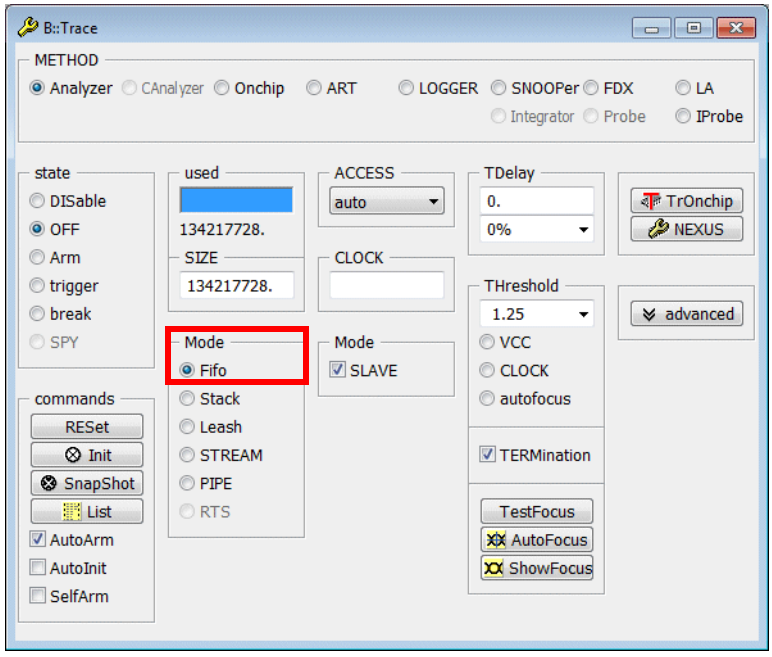
Trace.Mode PIPE	
Trace.PipeWRITE *<pipe_name>*	Connect to named pipe
Trace.PipeWRITE \\.\pipe*<pipe_name>*	Connect to named pipe (Windows)
Trace.PipeWRITE	Disconnect from named pipe

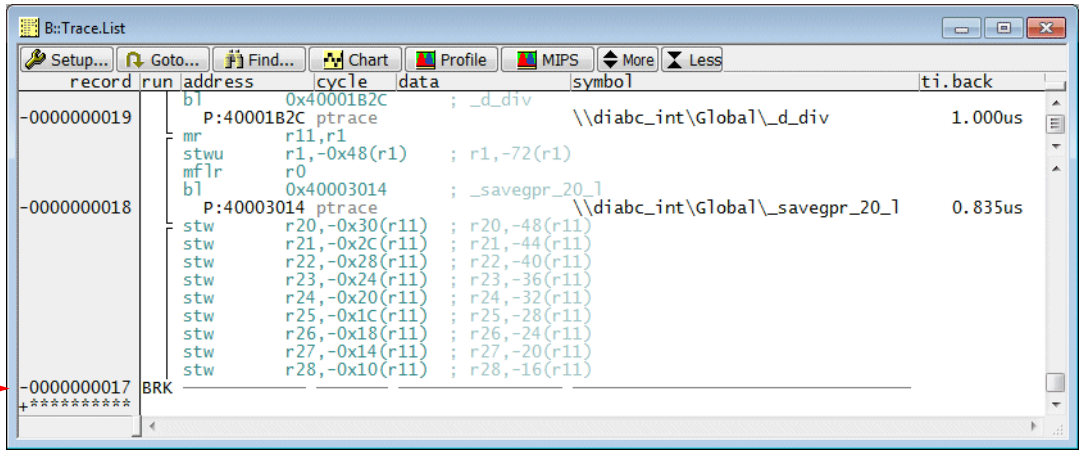
```
…

Trace.Mode PIPE                         ; switch trace to PIPE mode

Trace.PipeWRITE \\.\pipe\pproto00       ; connect to named pipe
                                        ; (Windows)

…

Trace.PipeWRITE                         ; disconnect from named pipe
```

NEXUS packets (no tool timestamp) are conveyed in PIPE mode.

- **RTS Mode (PowerTrace II hardware only):** RTS mode enables the processing while the trace data are recorded. The main use case for **RTS** is a live display of the code coverage results.

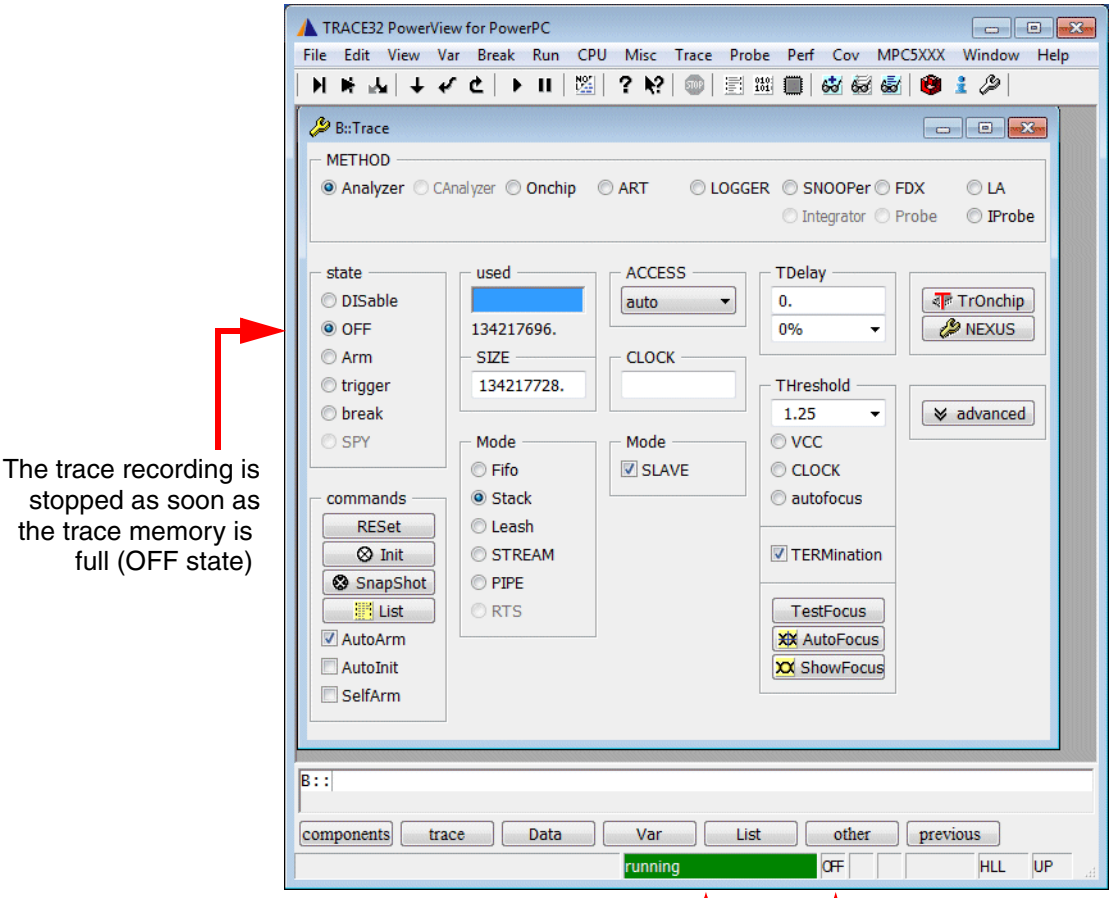
```
Trace.Mode Fifo                         ; default mode

                                        ; when the trace memory is full
                                        ; the newest trace information will
                                        ; overwrite the oldest one

                                        ; the trace memory contains all
                                        ; information generated 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 memory is full
                                        ; the trace recording is stopped

                                        ; the trace memory contains all
                                        ; information generated directly
                                        ; after the start of the program
                                        ; execution
```

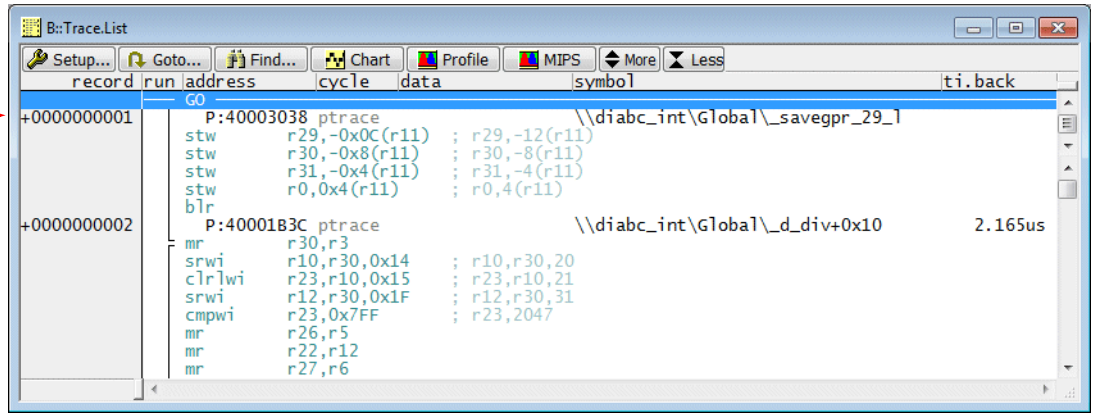


The trace recording is stopped as soon as the trace memory is full (OFF state)

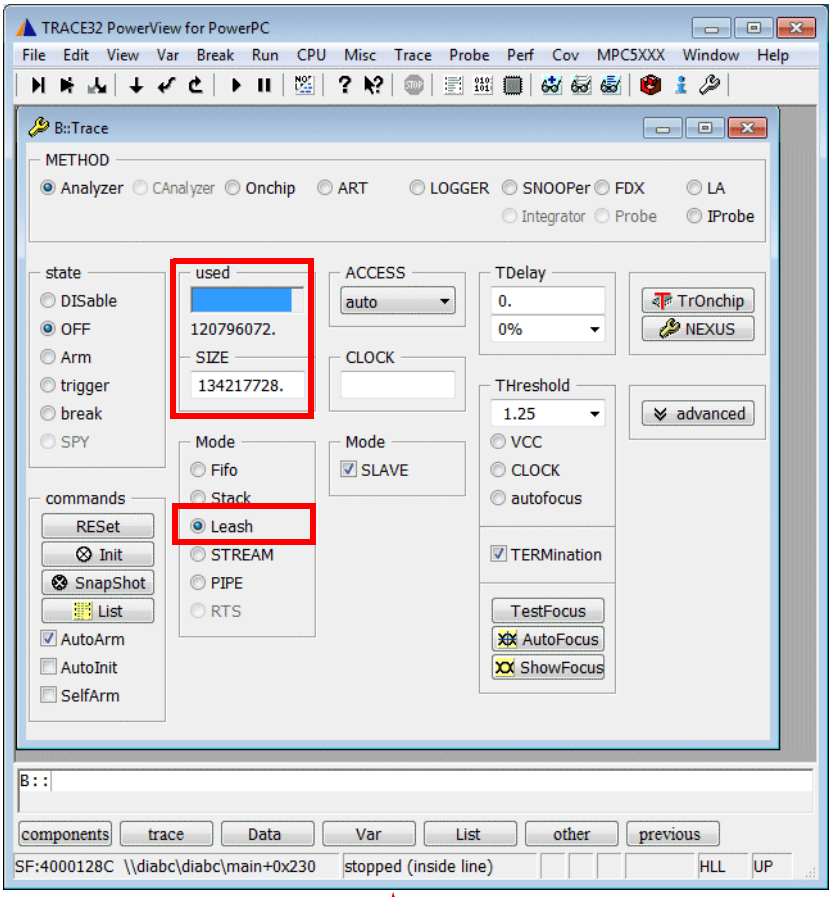
Green **running** in the Debug State Field indicates that program execution is running

OFF in the Trace State Field indicates that the trace recording is switched off

Since the trace recording starts with the program execution and stops when the trace memory 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 memory is nearly
                                        ; full the program execution is
                                        ; stopped

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



The program execution is **stopped** as soon as the trace buffer is nearly full.

Since stopping the program execution when the trace buffer is nearly full requires some logic/time, **used** is smaller than the maximum **SIZE.**

```
NEXUS.HTM ON                            ; enable Indirect Branch History
                                        ; Messaging to get compact raw
                                        ; trace data

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. This procedure extends the size of the trace memory to several T Frames.

- STREAM mode requires a TRACE32 trace hardware that allows streaming the trace information while recording. This is currently supported by PowerTrace II.
- STREAM mode required a 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 specific a different name and location for the streaming file.

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

TRACE32 stops the streaming when less then the 1 GByte free memory left on the drive by default.

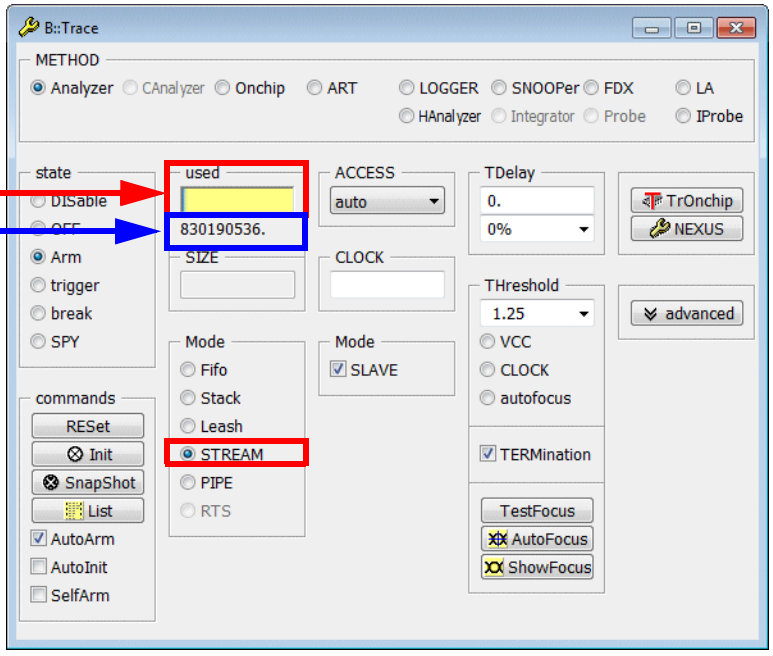
The command **Trace.STREAMFileLimit** *<+/- limit in bytes>* allows a user-defined free memory limitation.

```
Trace.STREAMFileLimit 5000000000.      ; streaming file is limited to
                                       ; 5 GByte

Trace.STREAMFileLimit -5000000000.     ; streaming is stopped when less
                                       ; the 5 GByte free memory is left
                                       ; on the drive
```

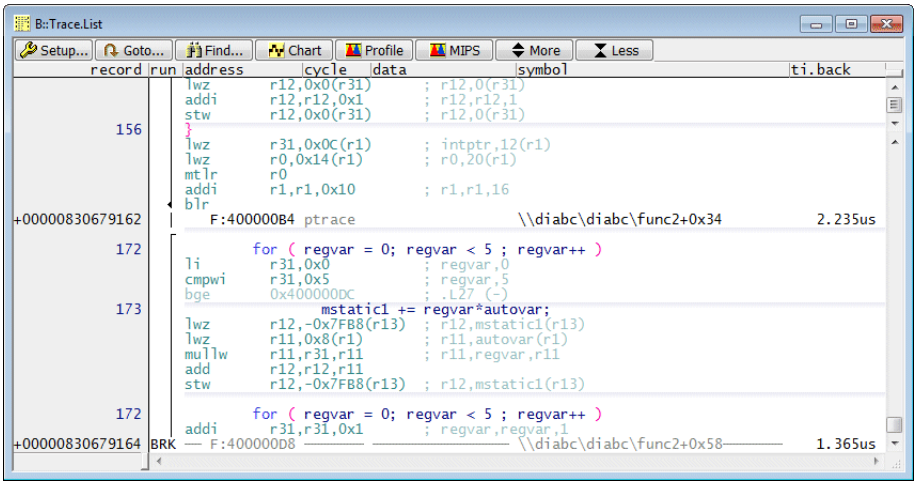
Please be aware that the streaming file is deleted as soon as you de-select the STREAM mode or when you exit TRACE32.

In STREAM mode the **used** field is split:

Indication of how much trace information is intercepted in the trace memory of the TRACE32 trace tool



Number of records saved to streaming file



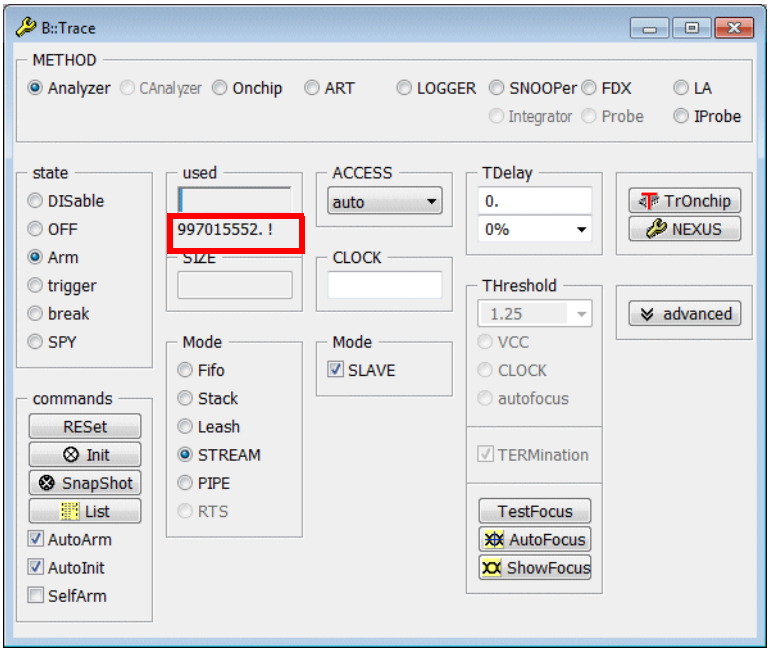
STREAM mode can generate very large record numbers

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, which can be considered to be operating as a large FIFO.

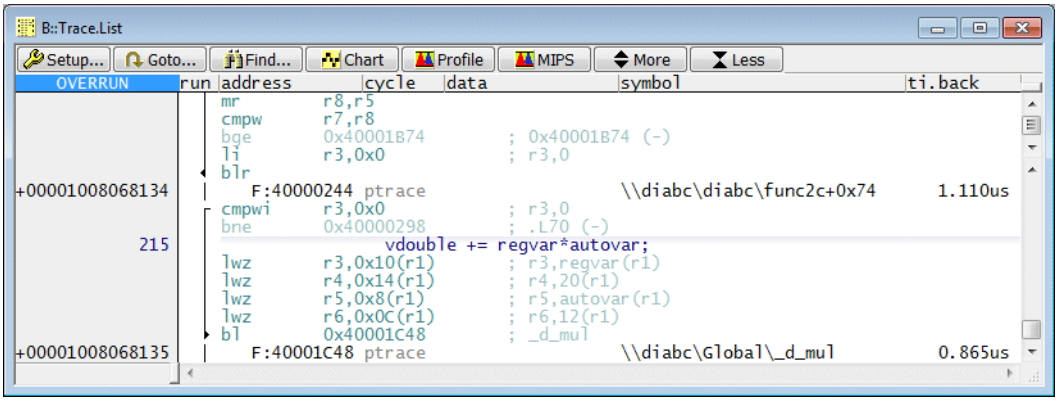
If the average data rate at the trace port exceeds the maximum transmission rate of the host interface in use, a **PowerTrace FIFO Overrun** occurs. TRACE32 stops streaming and empties the PowerTrace FIFO. Streaming is re-started after the PowerTrace FIFO is empty.

A **PowerTrace FIFO Overrun** is indicated as follows:

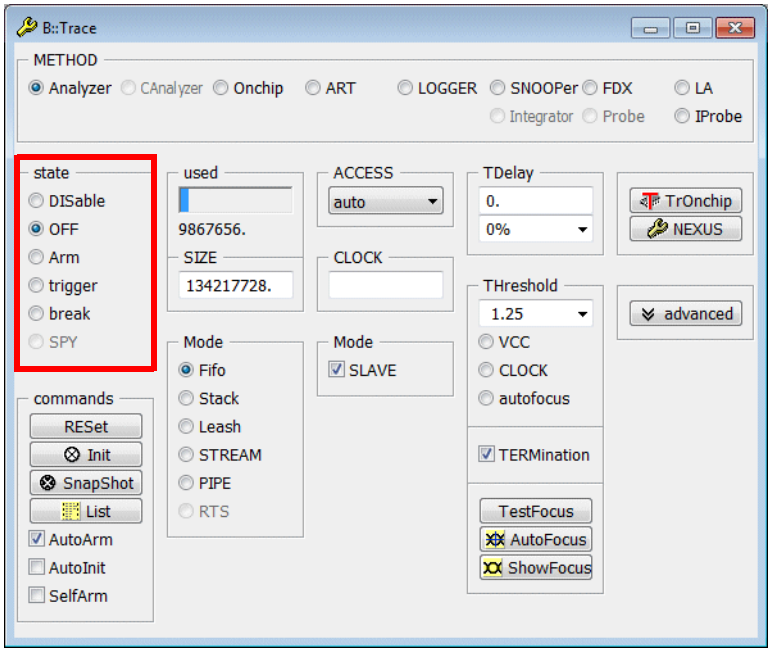
1. A **!** in the **used** area of the Trace configuration window indicates an overrun of the PowerTrace FIFO.



2. The OVERRUN is indicated in all trace display windows.

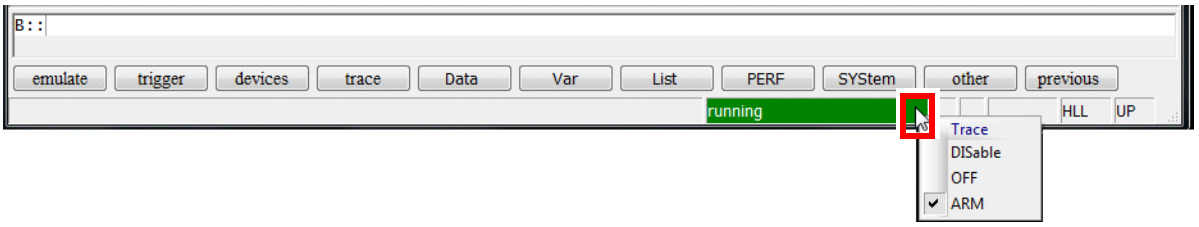


The trace buffer can either sample or allows the read-out for information display.

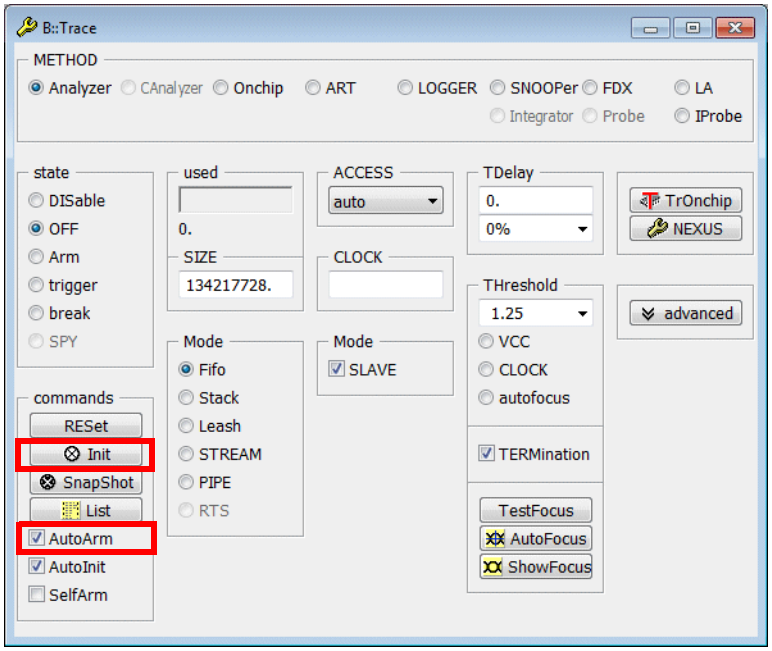


States of the Trace	
DISable	The trace is disabled.
OFF	The trace is not sampling. The trace contents can be red-out and displayed.
Arm	The trace is sampling. The trace contents can not be red.

The current state of the trace is always indicated in the **Trace State** field of the TRACE32 state line.



The Trace states **trigger** and **break** are introduced in detail later in this training.



Init Button	Clear the trace memory. All other settings in the Trace configuration window remain valid.
AutoInit CheckBox	ON: The trace memory is cleared whenever the program execution is started (Go, Step).

The focus of the Trace configuration window is:

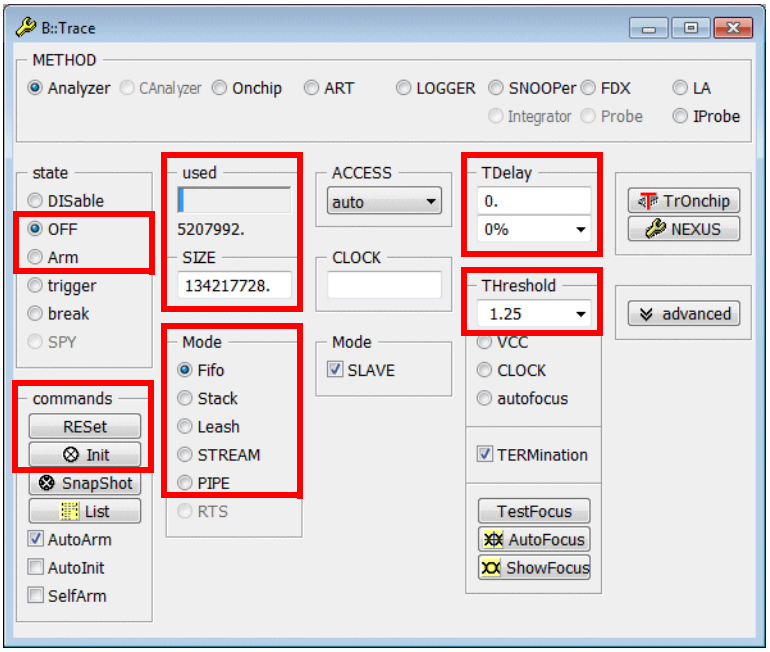
- Setup and maintenance of the TRACE32 trace tool (METHOD Analyzer).
- Setup and maintenance of the onchip trace (METHOD Onchip).

SMP systems: Due to the fact that one TRACE32 instance is used to control all cores, setups and states are identical for all controlled cores.

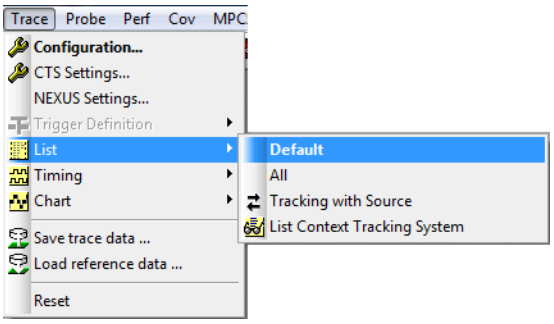
AMP systems: Due to the fact that the setups and states are maintained by multiple TRACE32 instances, the TRACE32 Resource Management maintains consistency for all joint settings and joint states.

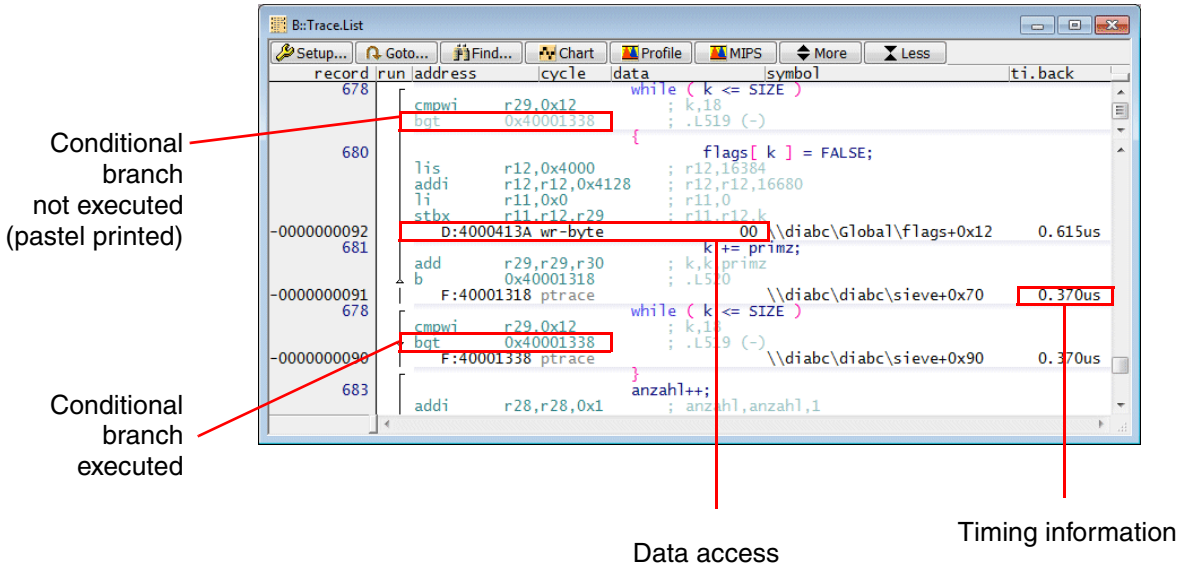
Consistency maintenance means: status changes in one TRACE32 instance affect all other TRACE32 instances.

Joint Settings and States



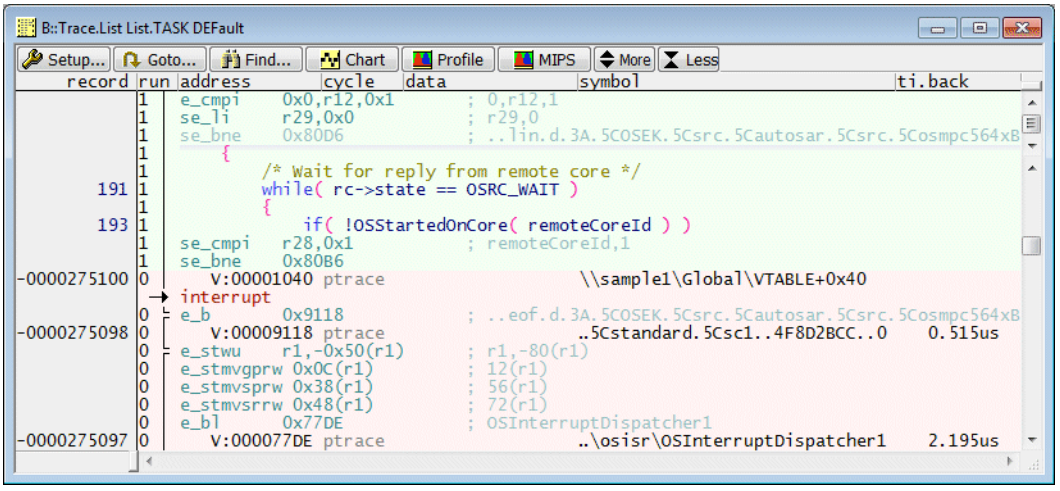
Default Listing



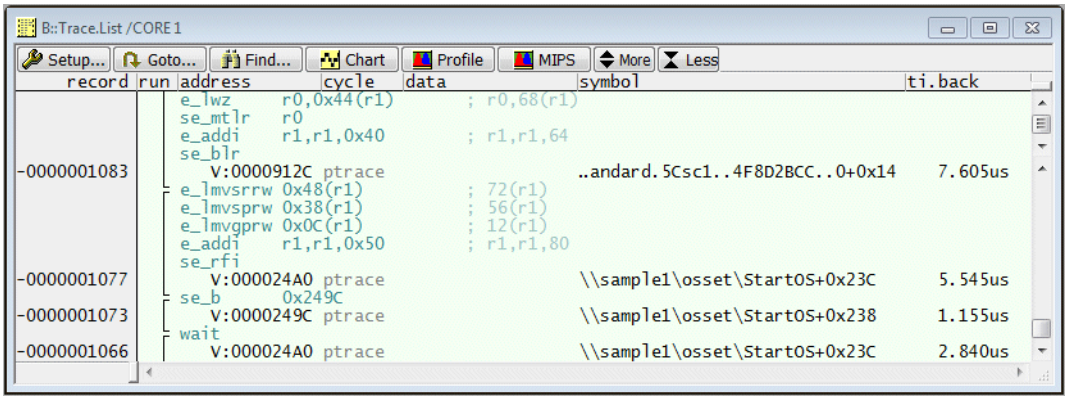


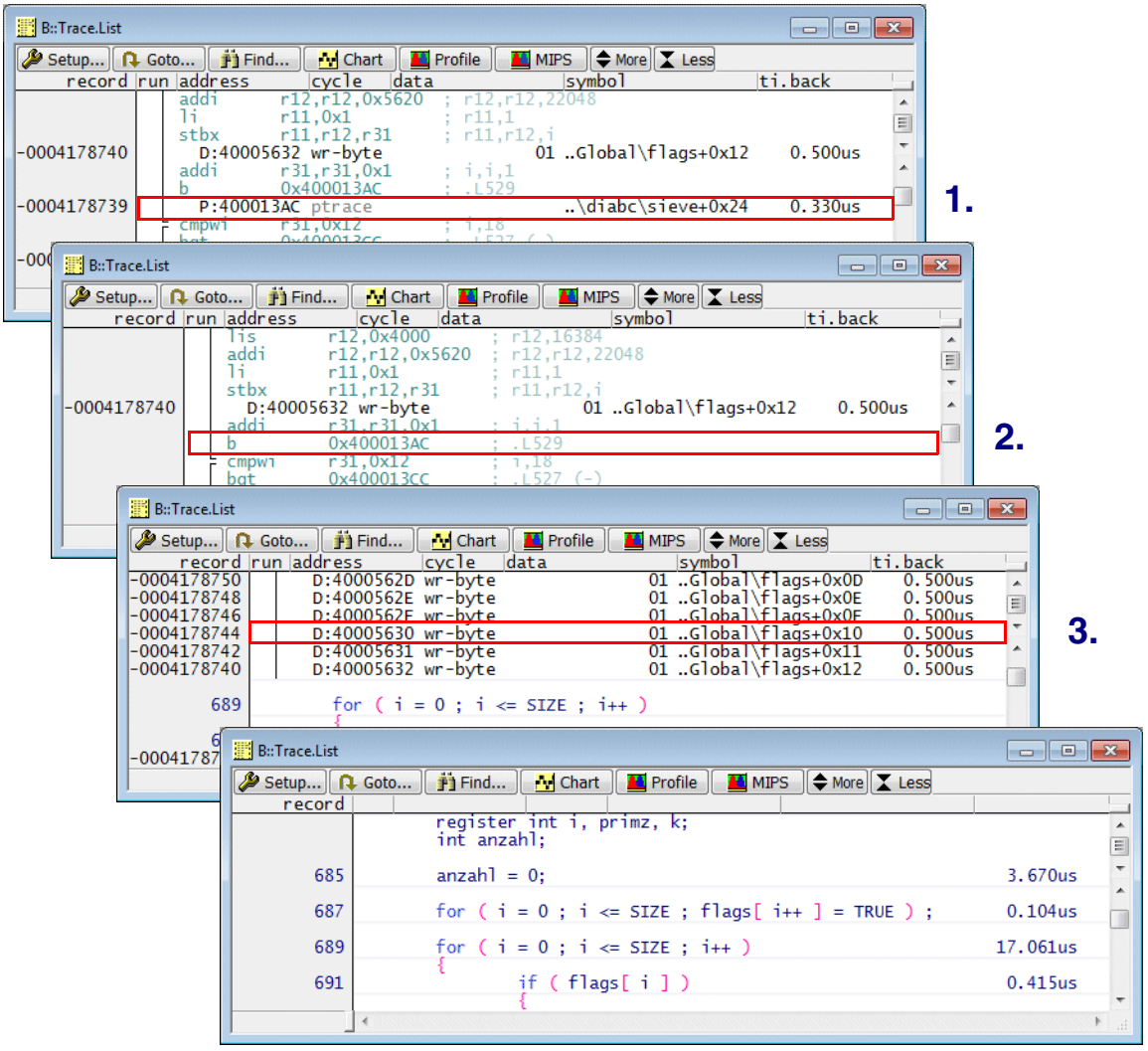
Trace.List	Default trace display.

The trace information for all cores is displayed by default in the **Trace.List** window if you are working with an **SMP** system. The column run and the coloring of the trace information are used for core indication.



Trace.List /CORE *<n>* The option CORE allows a per core display.

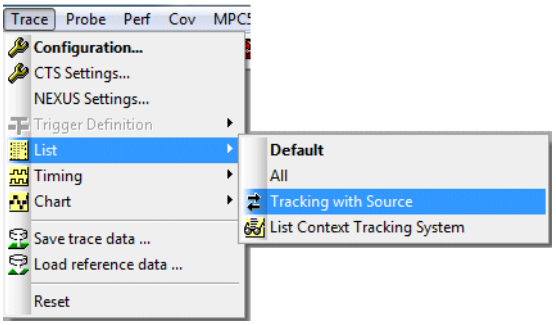


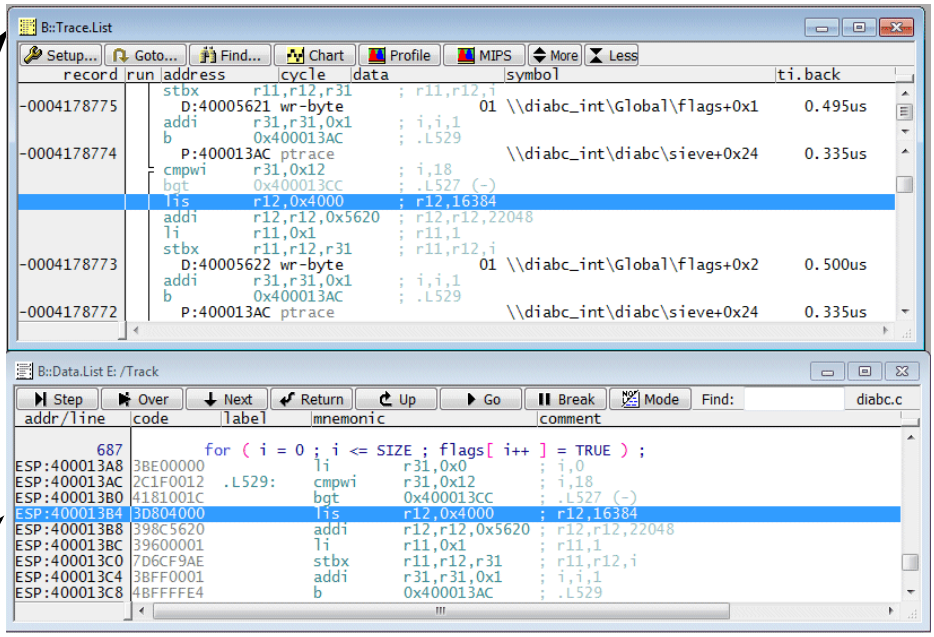


1. time Less	Suppress the display of the program trace package information (ptrace).
2. time Less	Suppress the display of the assembly code.
3. time Less	Suppress the data access information (e.g. wr-byte cycles).

The ***More*** button works vice versa.

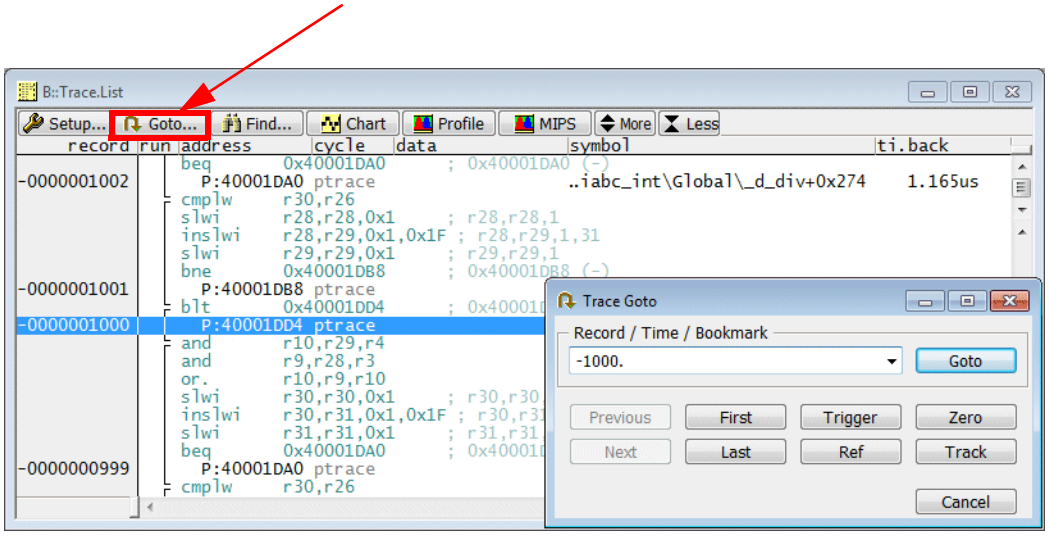
Correlating the Trace Listing with the Source Listing





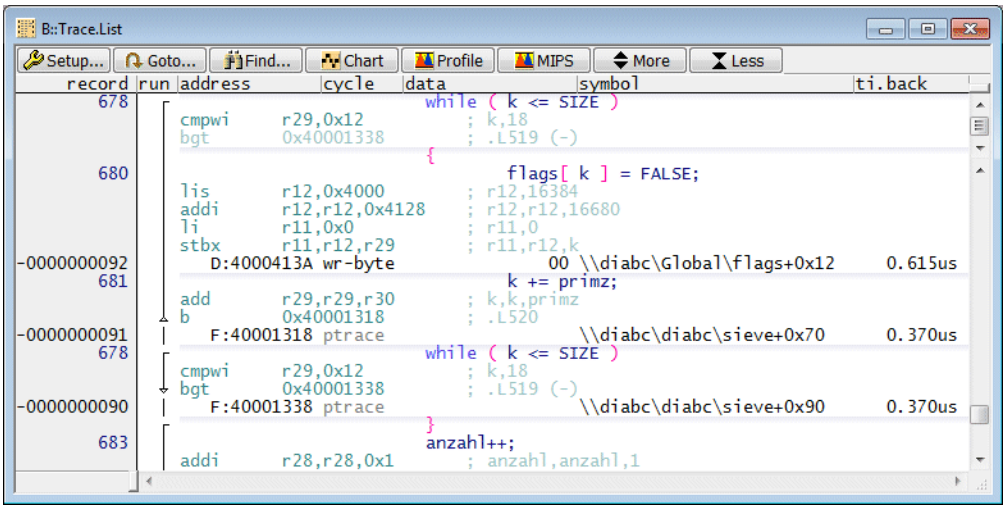
Active Window

All windows opened with the ***/Track*** option follow the cursor movements in the active window



Pg ↑	Scroll page up.
Pg ↓	Scroll page down.
Ctrl - Pg ↑	Go to the first record sampled in the trace buffer.
Ctrl - Pg ↓	Go to the last record sampled in the trace buffer.

Default Display Items



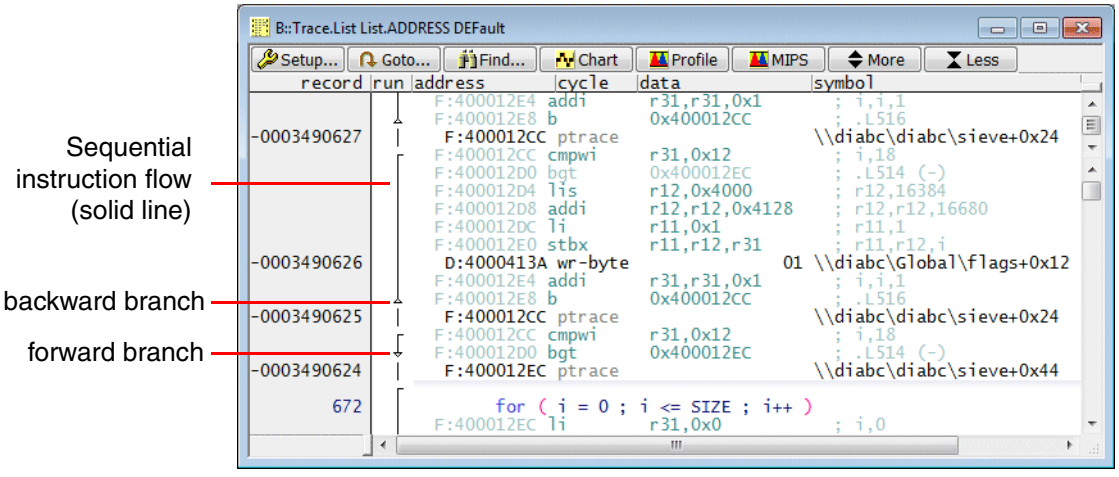
- **Column *record***

  Displays the record numbers

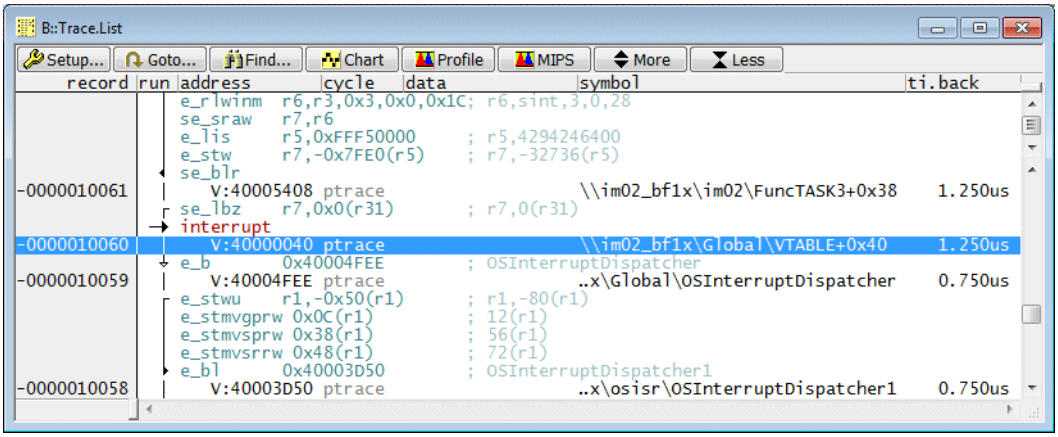
- **Column *run***

  The column run displays some graphic element to provide a quick overview on the instruction flow.

```
Trace.List List.ADDRESS DEFault
```



The column run also indicates interrupts and TRAPs.



- **Column *cycle***

The main cycle types are:

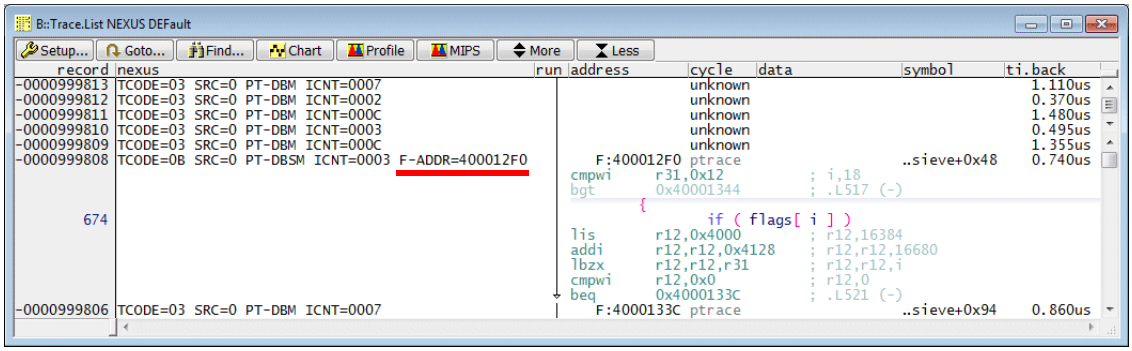
- ptrace (program trace information)

- rd-byte, rd-word, rd-long (read access)

- wr-byte, wr-word, wr-long (write access)

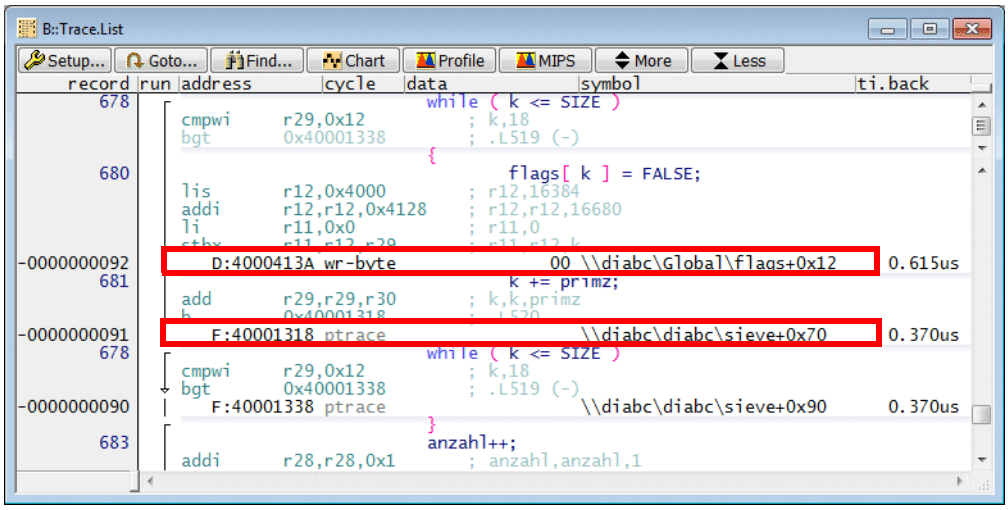
- owner (ownership trace messages)

- unknown (Branch Trace Messages that can not be decoded)



The decoding of the Branch Trace Messages can start, as soon as a full address (F-ADDR) is exported. Branch Trace Messages that can not be decoded are marked as **unknown**.

- **Column *address/symbol***

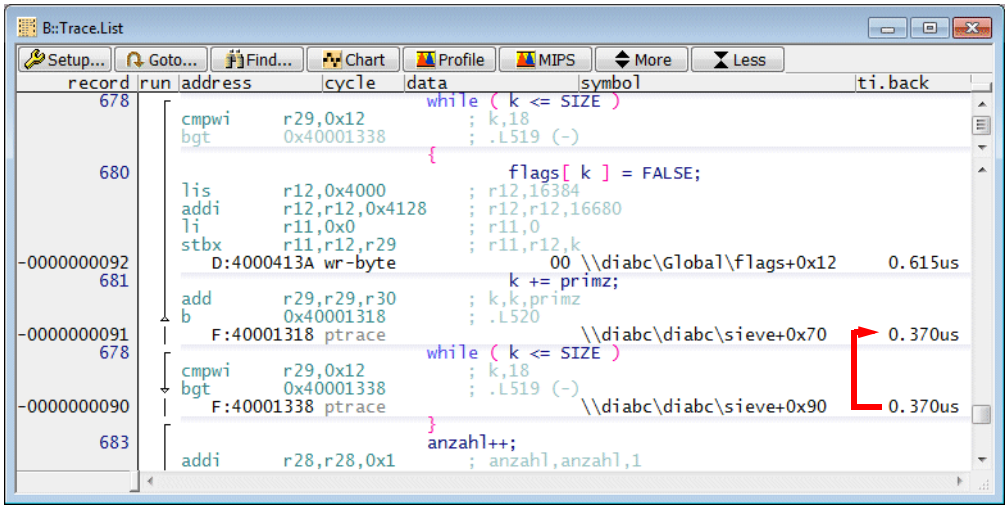


The **address column** shows the following information:
<access class>:<logical_address>

Access Classes	
F	Program address, disassembly shows standard PowerPC instructions
V	Program address, disassembly shows VLE encoded instructions
D	Data address

The **symbol column** shows the corresponding symbolic address.

- **Column *ti.back***

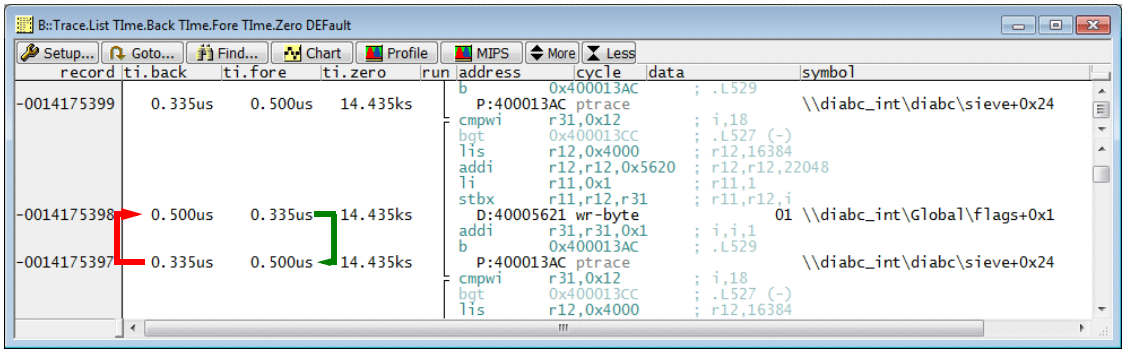


The **ti.back column** shows the time distance to the previous record.

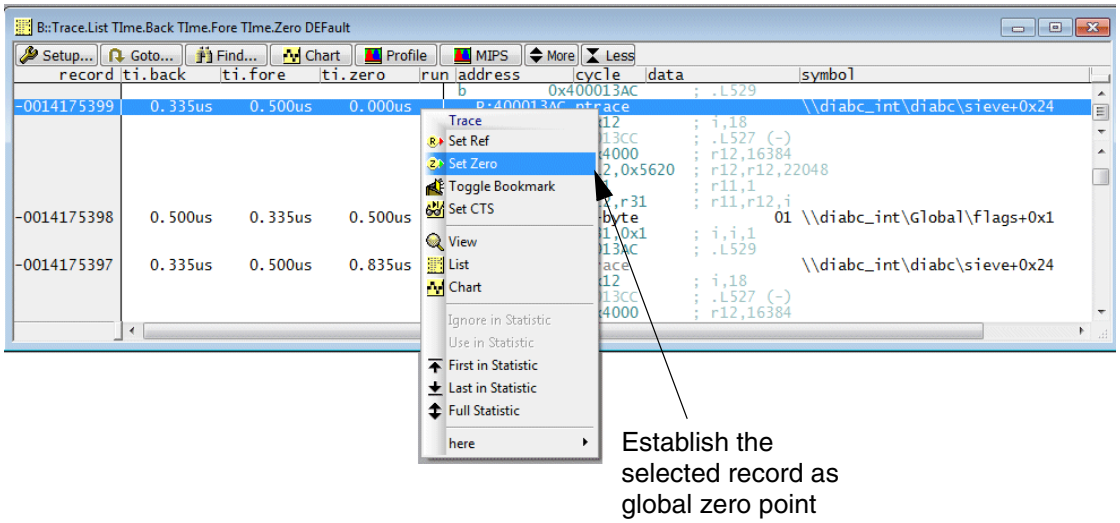
Time Information

TIme.Back	Time relative to the previous record (red)
TIme.Fore	Time relative to the next record (green).
TIme.Zero	Time relative to the global zero point.

```
Trace.List TIme.Back TIme.Fore TIme.Zero DEFault
```

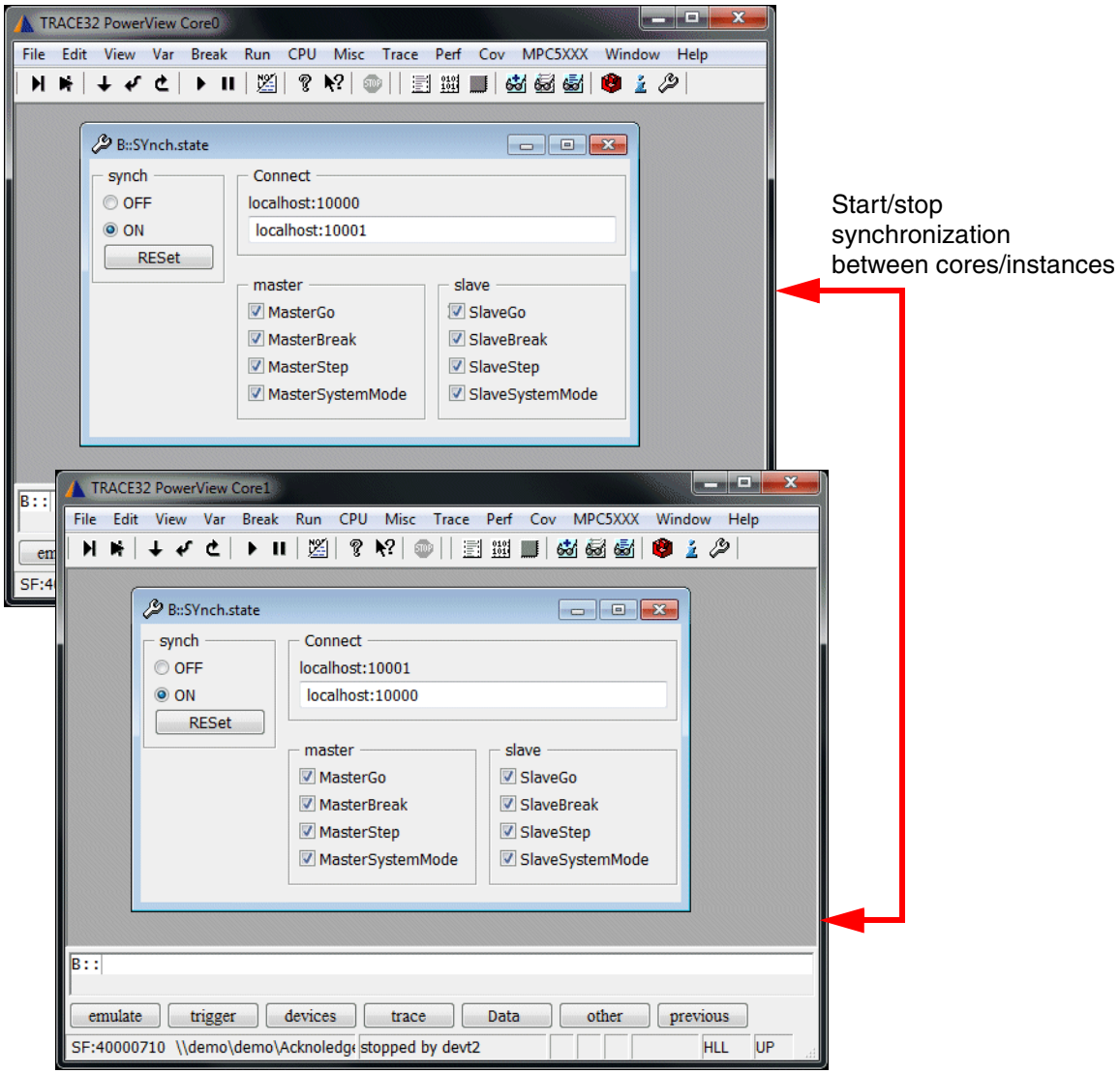


Set the Global Zero Point (tool timestamp only)



Setup

If a AMP multi-core debugging session is set up, start/stop synchronization for the cores is established.

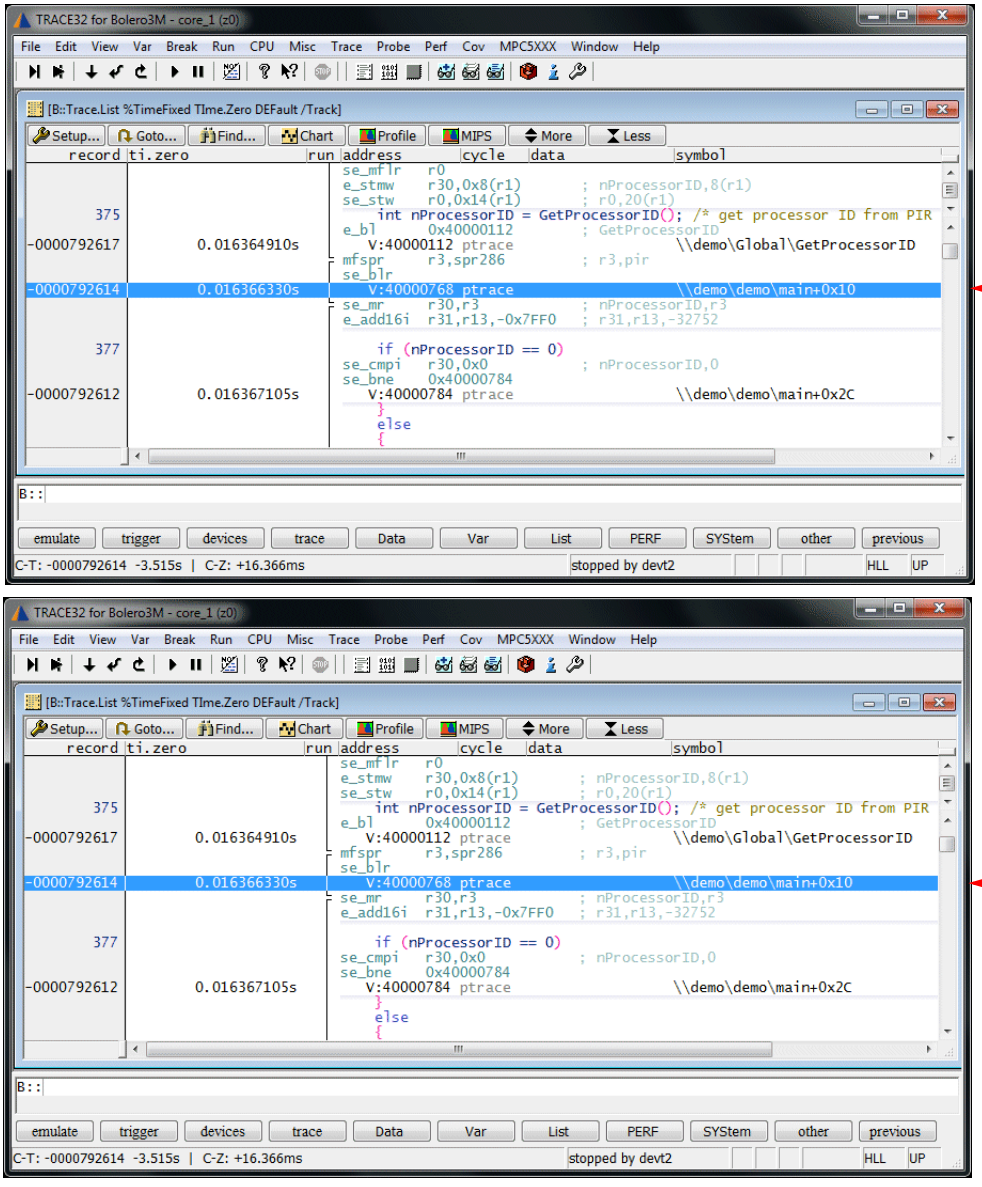


For trace synchronization the following commands have to be executed:

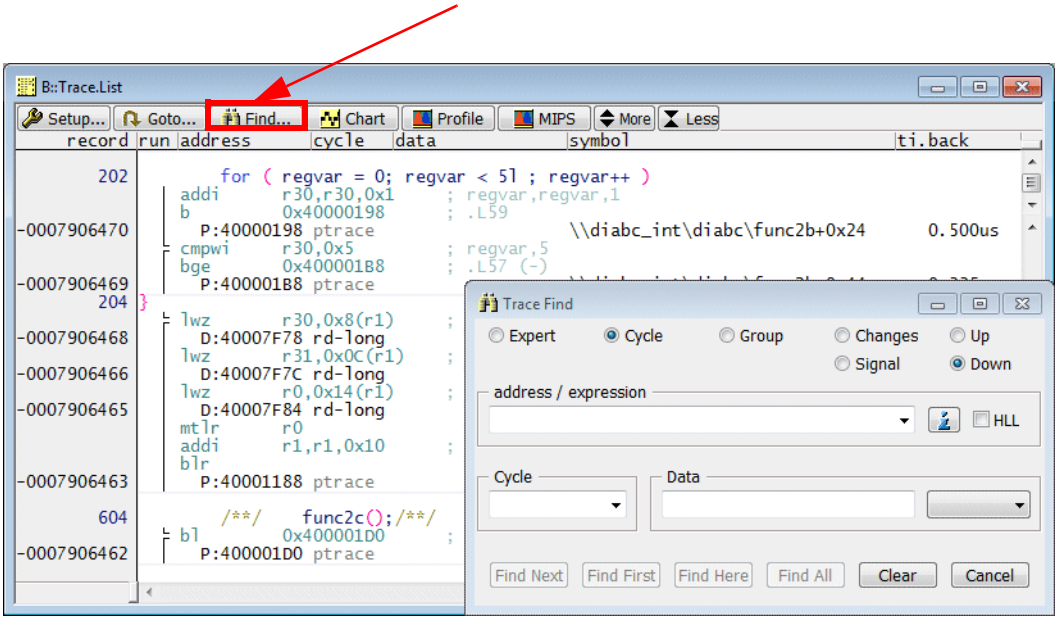
```
SYnch.XTrack localhost:10001          ; in TRACE32 instance for core0

SYnch.XTrack localhost:10000          ; in TRACE32 instance for core1
```

The base for the trace synchronization is the tool timestamp or if enabled the Nexus timestamps.

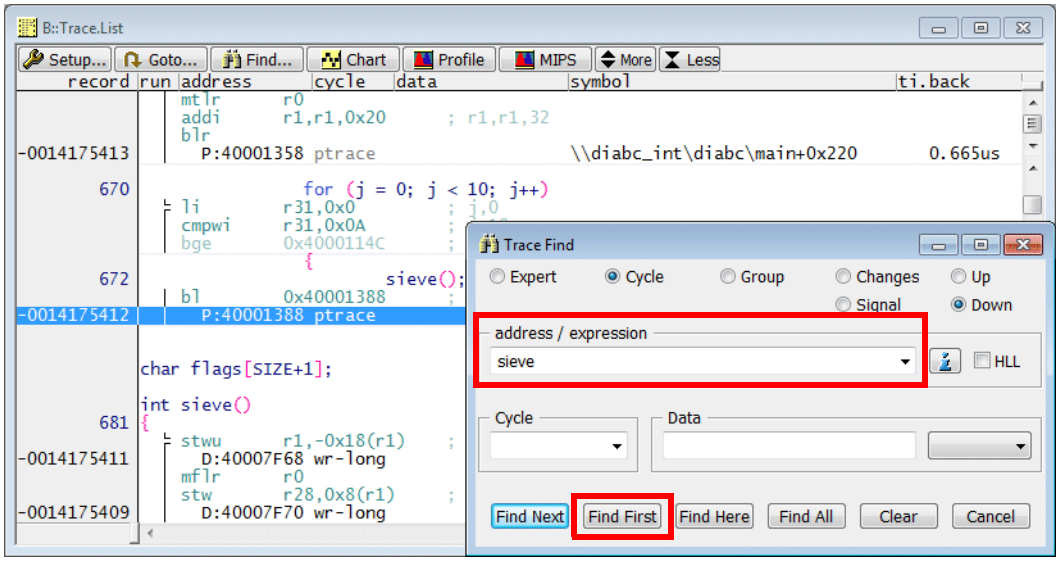
```
Trace.List TIme.Zero DEFault /Track        ; /Track enables here the
                                           ; time synchronisation to
                                           ; trace display windows in
                                           ; other TRACE32 instance
```



Find a Specific Record



Example: Find a specific symbol address.



A more detailed description on how to find specific events in the trace is given in **"Application Note for Trace.Find"** (app_trace_find.pdf).

Belated Trace Analysis

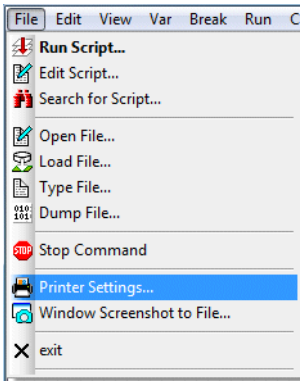
There are several ways for a belated trace analysis:

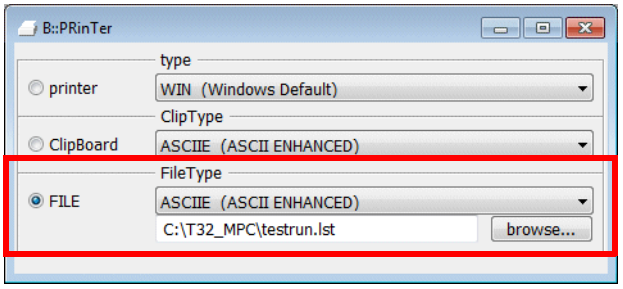
1. Save a part of the trace contents into an ASCII file and analyze this trace contents by reading.
2. Save the trace contents in a compact format into a file. Load the trace contents at a subsequent date into a TRACE32 Instruction Set Simulator and analyze it there.

Save the Trace Information to an ASCII File

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

1. Select **Print** in the **File** menu to specify the file name and the output format.






```
PRinTer.FileType ASCIIE                 ; specify output format
                                        ; here enhanced ASCII

PRinTer.FILE testrun.lst                ; specify the file name
```

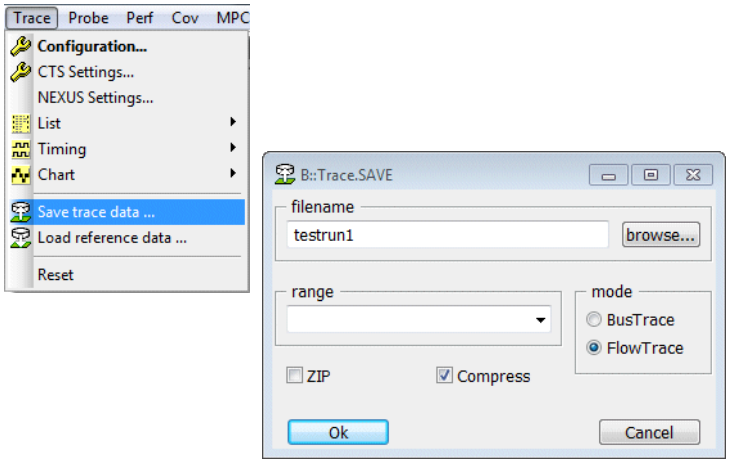
2. It only makes sense to save a part of the trace contents into an ASCII-file. Use the record numbers to specify the trace part you are interested in.

   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.

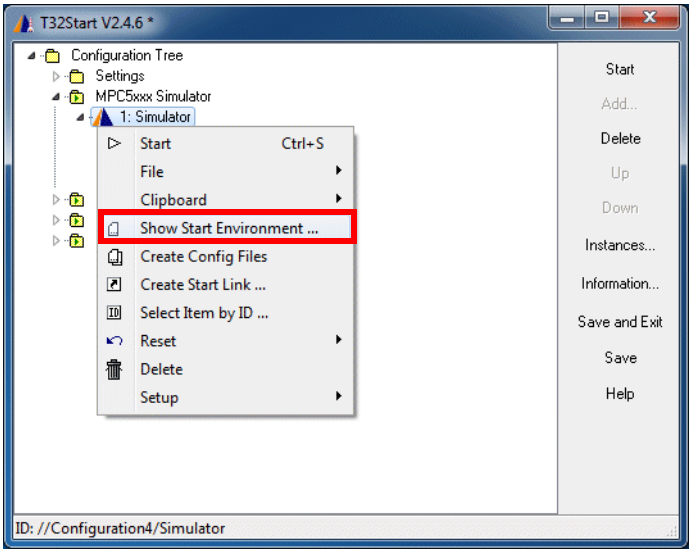
1. **Save the contents of the trace memory into a file.**

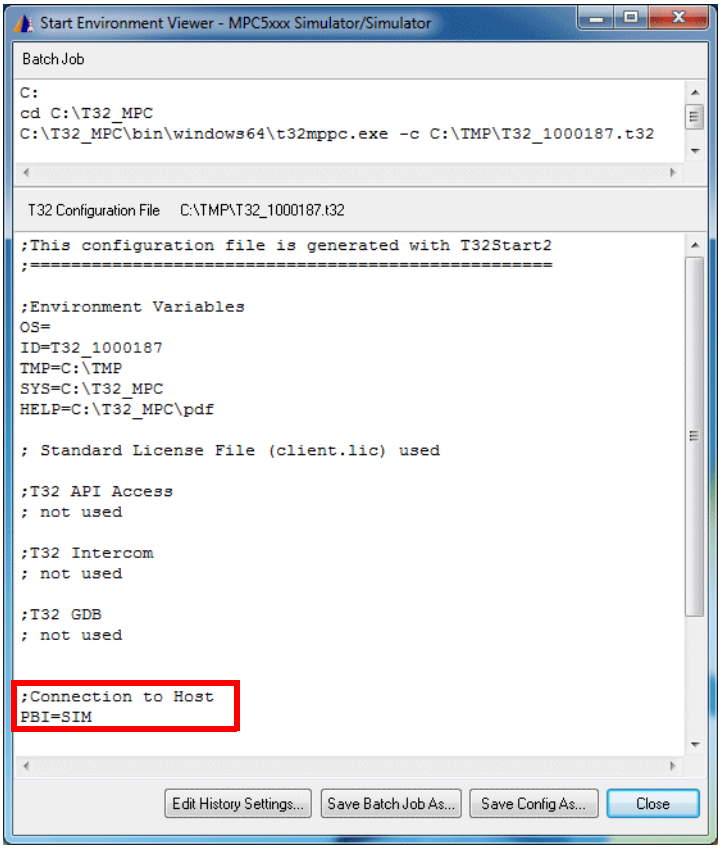


The default extension for the trace file is **.ad**.

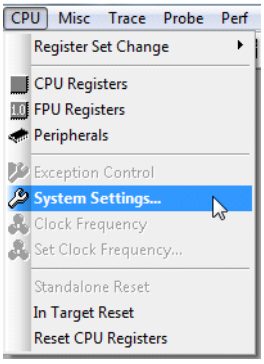
```
Trace.SAVE testrun1
```

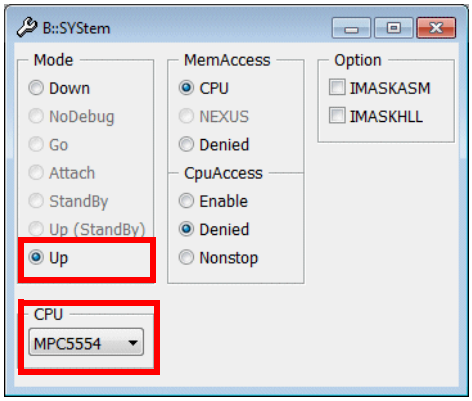
2. Start a TRACE32 Instruction Set Simulator (PBI=SIM).



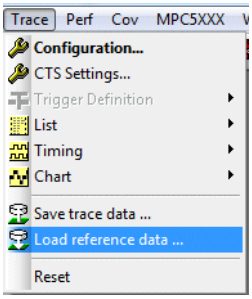


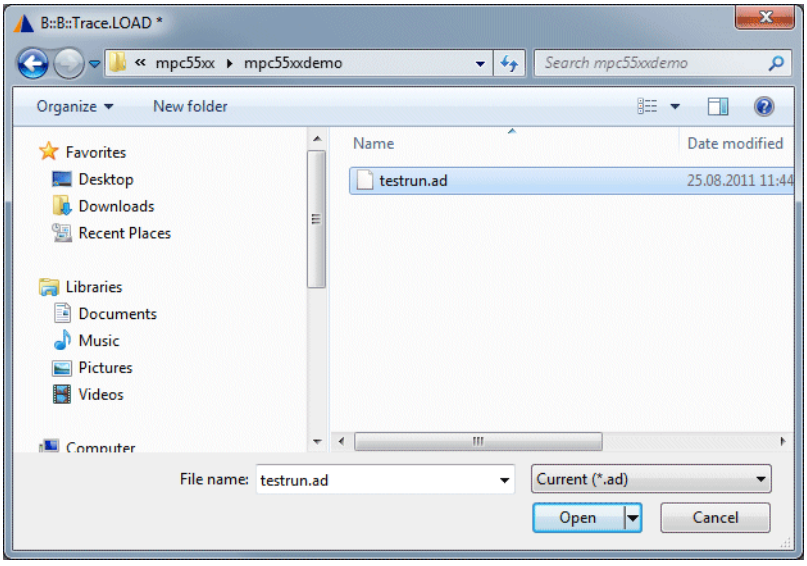
3. **Select your target CPU within the simulator. Then establish the communication between TRACE32 and the simulator.**





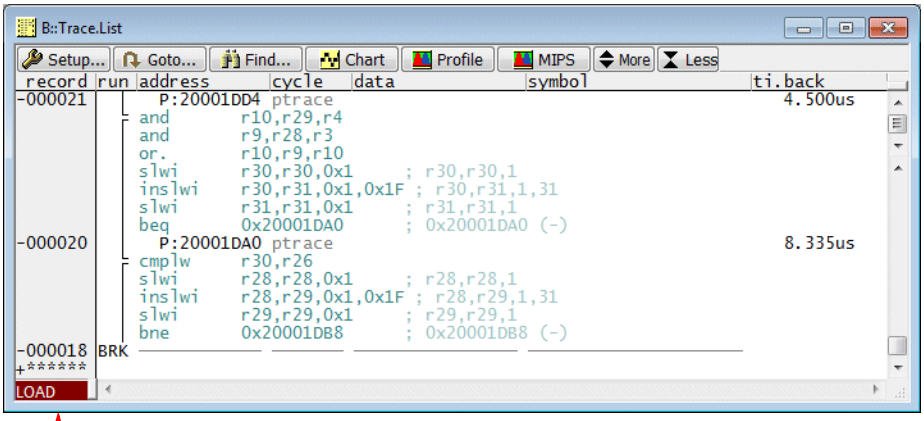
4. **Load the trace file.**






```
Trace.LOAD testrun
```

5. Display the trace contents.



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

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

```
Data.LOAD.Elf diabc_ext.x /NoCODE
```

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

Trace-based Debugging (CTS)

Trace-based debugging allows to re-run the recorded program section within TRACE32 PowerView.

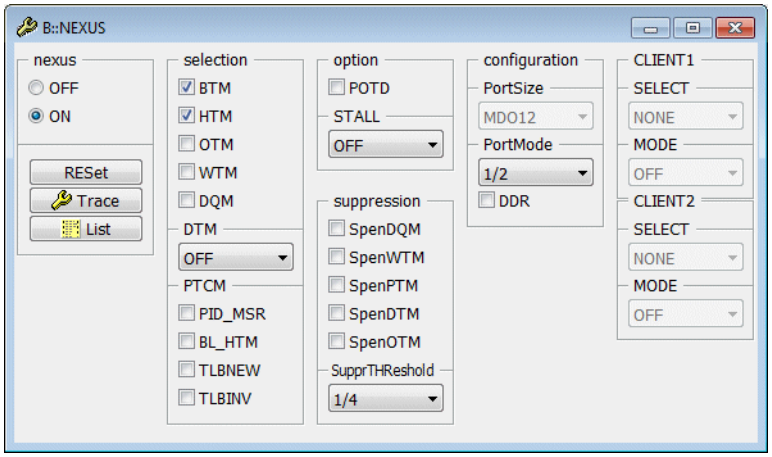
If Data Trace Messages were enabled for ReadWrite, it is also possible to watch memory, variable and register changes while re-running the recorded program section.

Re-Run the Program

Setup

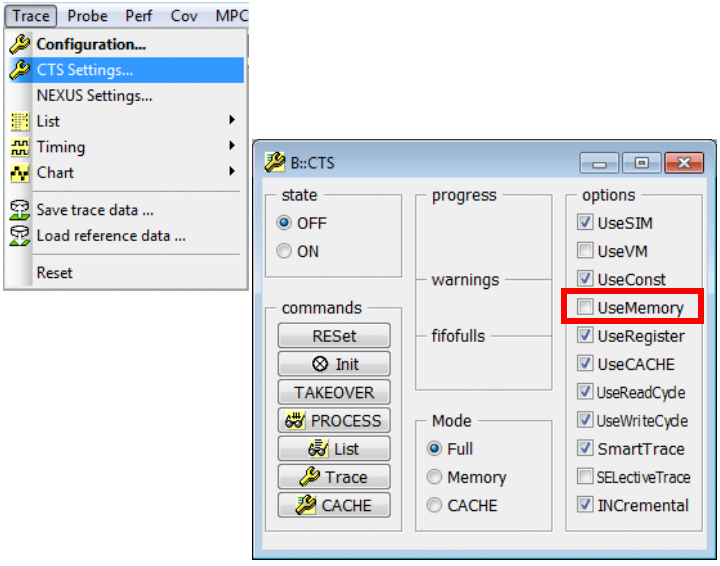
In order to re-run the program, it is sufficient to only enable Branch Trace Messaging. One of the following configurations is suitable:

- BTM ON
- BTM ON + HTM ON
- BTM ON + HTM ON + BL_HTM ON



If you use an OS, it is recommended to also record the task switch information. See **“OS-Aware Tracing (ORTI File)”** in Training Nexus Tracing, page 193 (training_nexus.pdf).

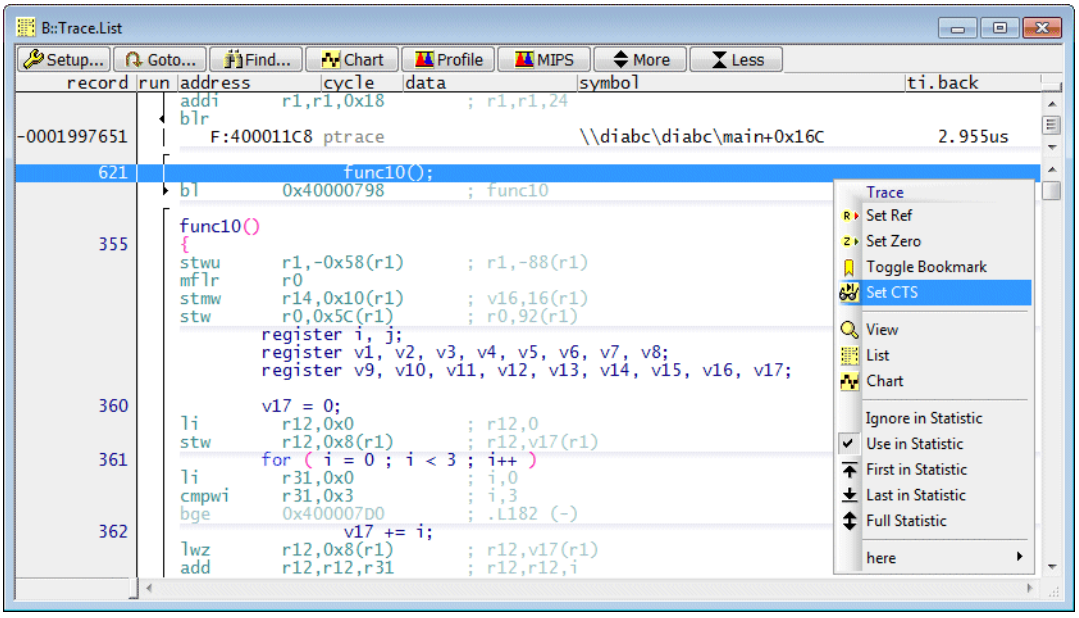
Un-check **UseFinalMemory** in the CTS configuration window. A full explanation on this is given later in the chapter **"CTS Technique"**, page 99



CTS.state

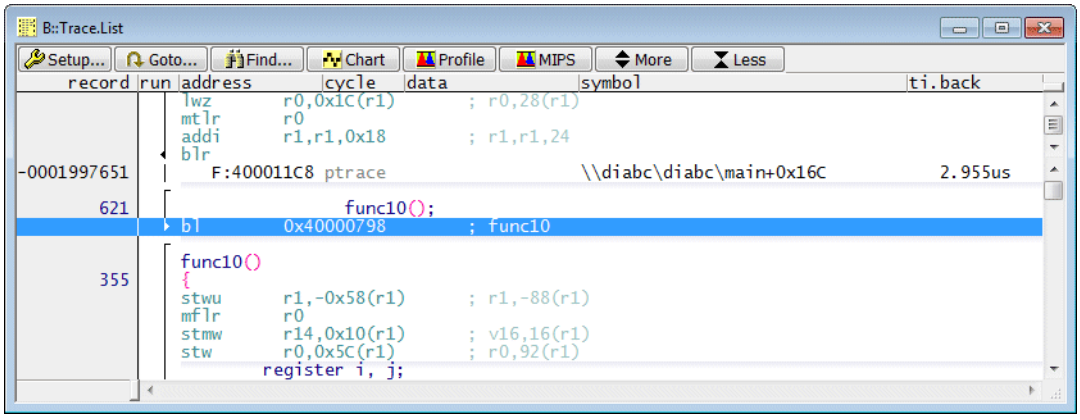
CTS.UseFinalMemory OFF

Specify the starting point for the trace re-run by selecting **Set CTS** from the Trace pull-down menu. The starting point in the example below is the entry to the function **func10**.



Selecting **Set CTS** has the following effect:

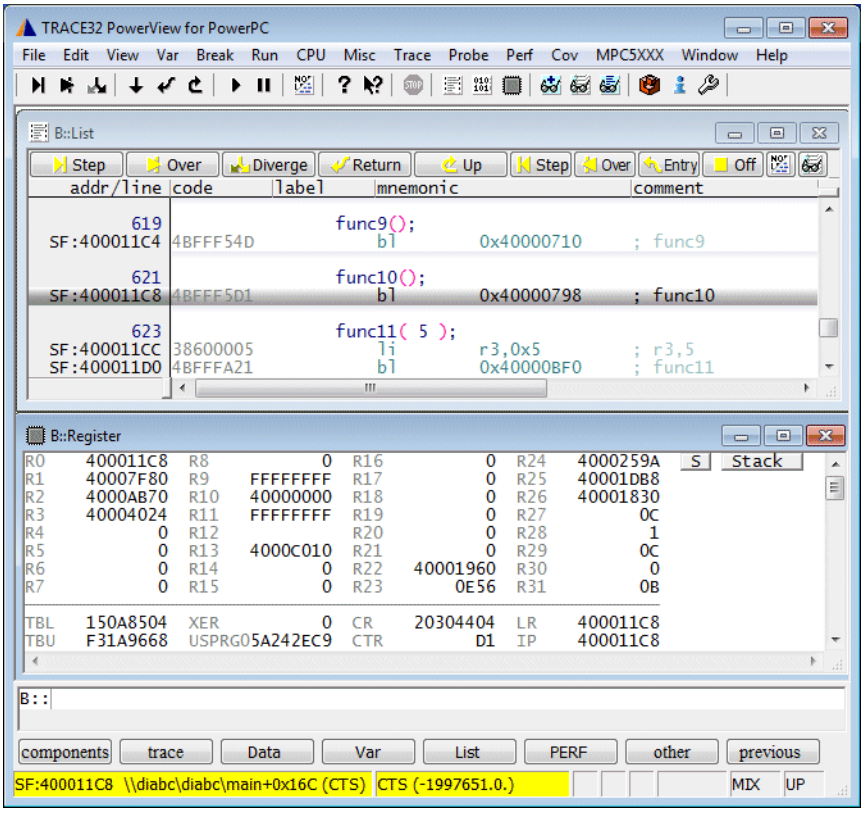
- TRACE32 PowerView will use the preceding trace packet as starting point for the trace re-run.

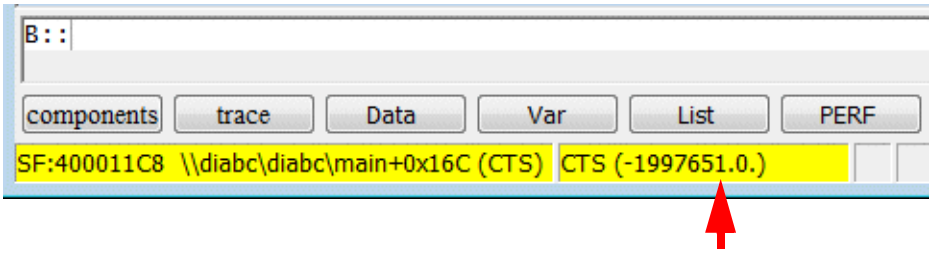


- The TRACE32 PowerView GUI does no longer show the current state of the target system, but it shows the target state as it was, when the starting point instruction was executed. This display mode is called CTS View.

CTS View means:

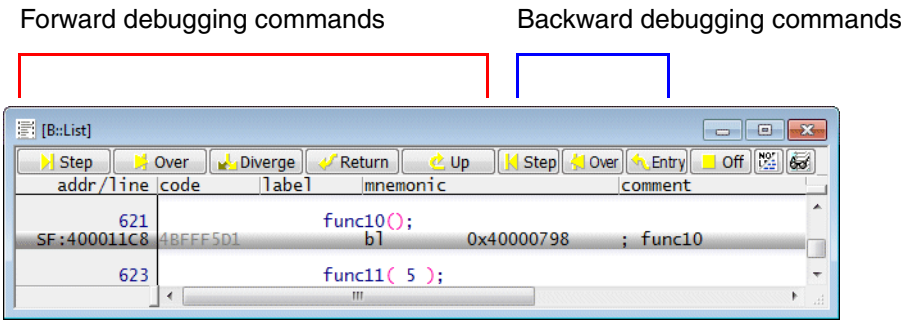
- The instruction pointer is set to the values it had when the starting point instruction was executed. This is done for all cores if an SMP system is under test.
- The content of the core registers is reconstructed (as far as possible) to the values they had when the starting point instruction was executed. This is done for all cores if an SMP system is under test. If TRACE32 can not reconstruct the content of a register it is displayed as empty.
- TRACE32 PowerView uses a yellow look-and-feel to indicate CTS View.
- The **Off** button in the Source Listing can be used to switch off the CTS View.



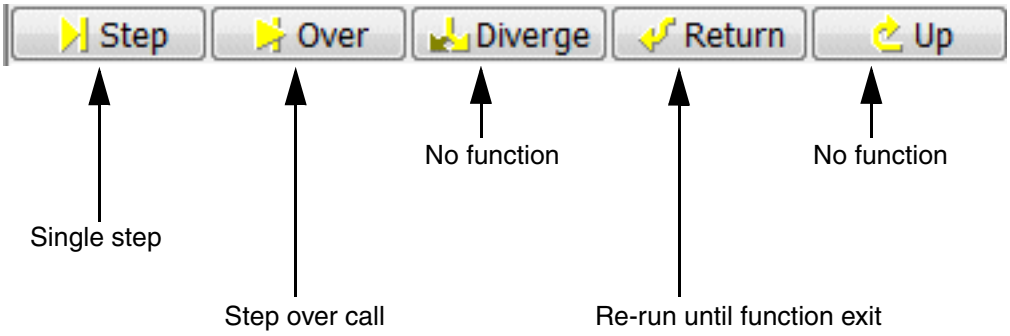


TRACE32 PowerView displays the state of the target as it was when the instruction of the trace record -1997651.0 was executed

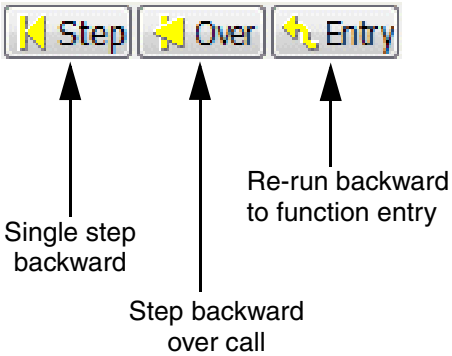
Now you can start to re-run the recorded program section within TRACE32 PowerView by forward or backward debugging.



Forward Debugging



Backward Debugging

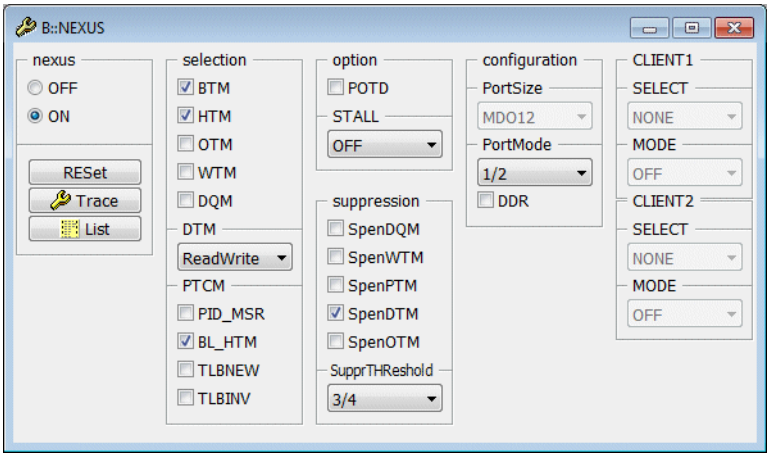


Re-Run the Program and Watch the Variables

This feature only makes send for the IEEE-ISTO 5001-2008 and the IEEE-ISTO 5001-2012 standard.

Setup

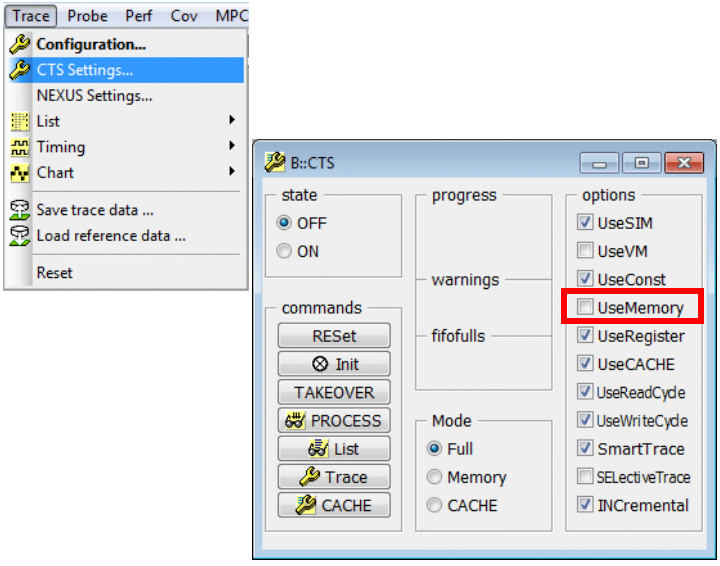
In order to re-run the program and watch the variables, the following Nexus setups are recommended:



- Enable Branch Trace Messaging (BTM ON / BTM ON + HTM ON / BTM ON + HTM ON + BL_HTM ON)
- Enable Data Trace Messages for read/write accesses, but suppress Data Trace Messages on overflow threat.

```
; Configuration example

NEXUS.BTM ON

NEXUS.HTM ON

NEXUS.BL_HTM ON

NEXUS.DTM ReadWrite

NEXUS.SupprTHReshold 3/4     ; Advise Nexus to suppress specified
                             ; messages when Nexus FIFO is 3/4 filled

NEXUS.SpenDTM ON             ; Advise Nexus to suppress Data Trace
                             ; Messages when the specified filling
                             ; level is reached
```

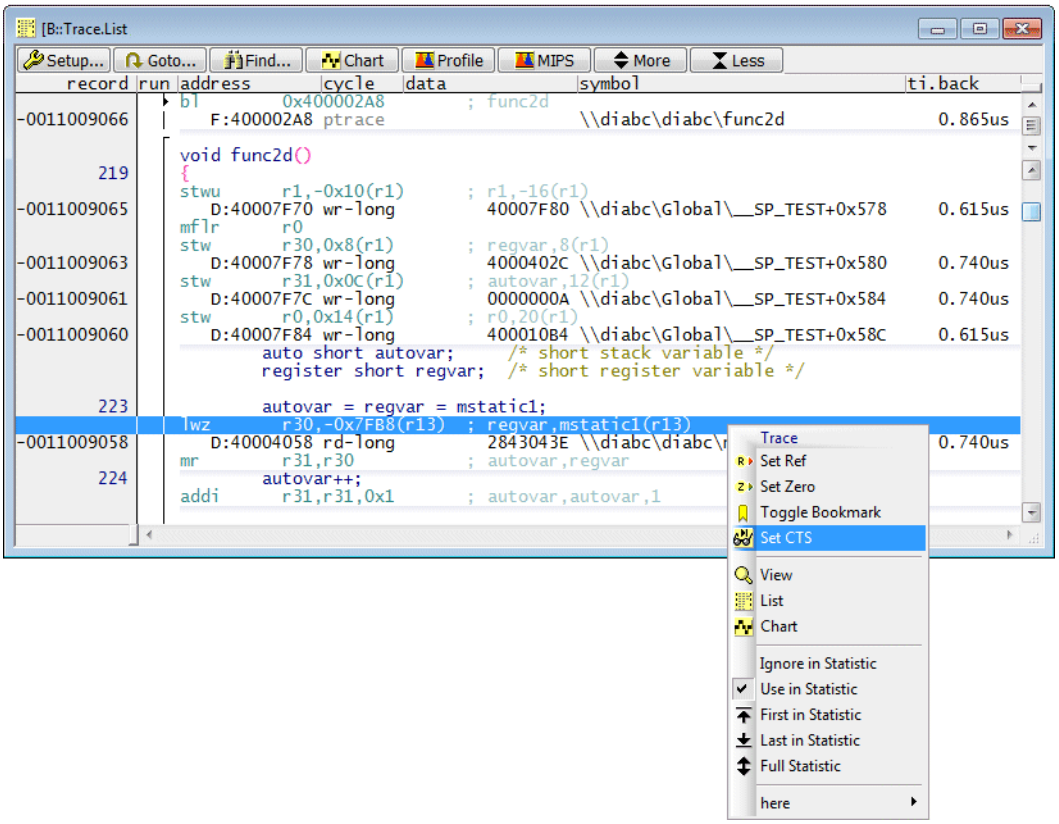
Un-check **UseMemory** in the CTS configuration window. A full explanation on this is given later in the chapter **"CTS Technique"**, page 99.



CTS.state

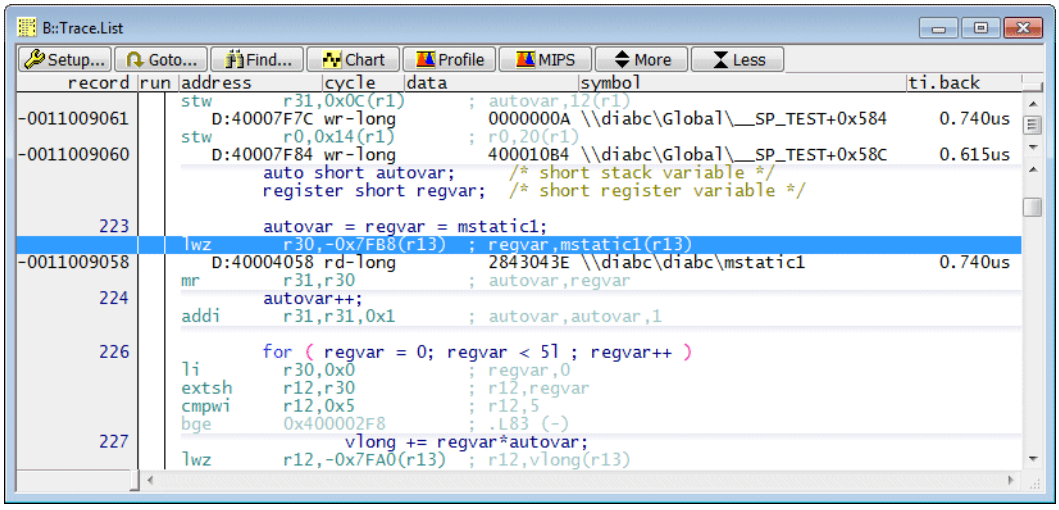
CTS.UseMemory OFF

Specify the starting point for the trace re-run by selecting **Set CTS** from the Trace pull-down menu. The starting point in the example below is the read access to the variable **mstatic1** in function **func2d**.



Selecting **Set CTS** has the following effect:

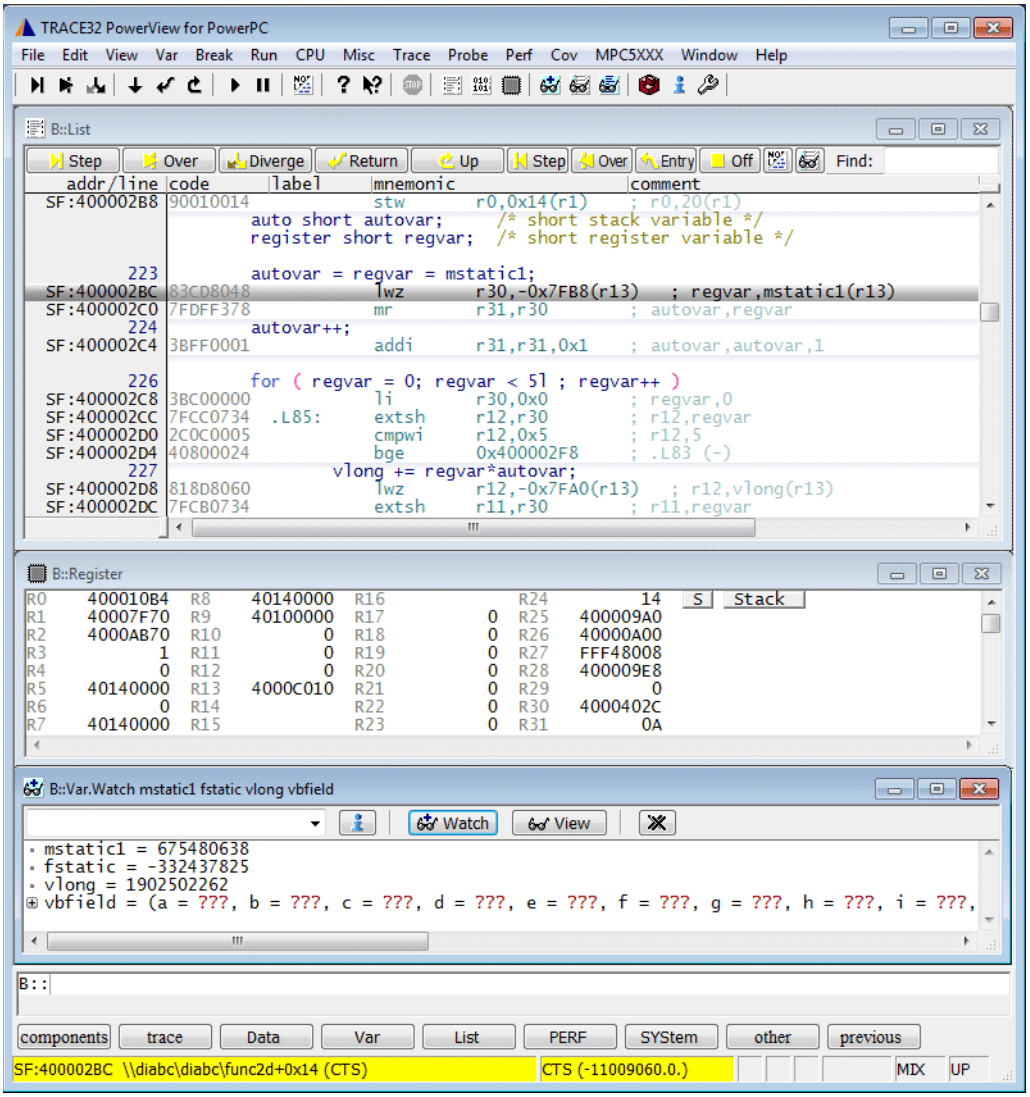
- TRACE32 PowerView will use the preceding trace packet as starting point for the trace re-run.

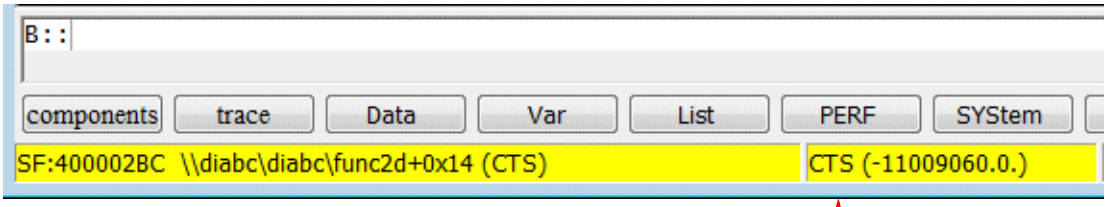


- The TRACE32 PowerView GUI does no longer show the current state of the target system, but it shows the target state as it was, when the starting point instruction was executed. This display mode is called CTS View.

CTS View means:

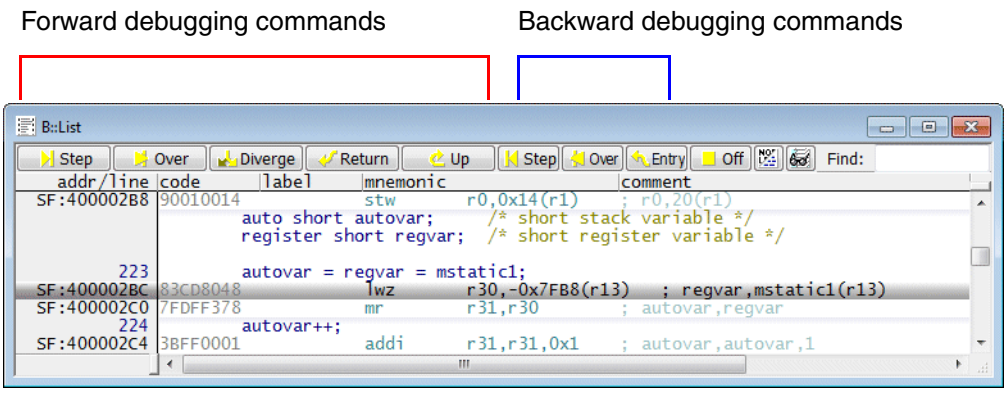
- The instruction pointer is set to the values it had when the starting point instruction was executed. This is done for all cores if an SMP system is under test.
- The content of the core registers is reconstructed (as far as possible) to the values they had when the starting point instruction was executed. This is done for all cores if an SMP system is under test. If TRACE32 can not reconstruct the content of a register it is displayed as empty.
- The contents of the variables changed by the recorded program section are reconstructed (as far as possible) to the values they had when the starting point instruction was executed. If TRACE32 can not reconstruct the content of a variable ??? are displayed.
- TRACE32 PowerView uses a yellow look-and-feel to indicate CTS View.
- The **Off** button in the Source Listing can be used to switch off the CTS View.



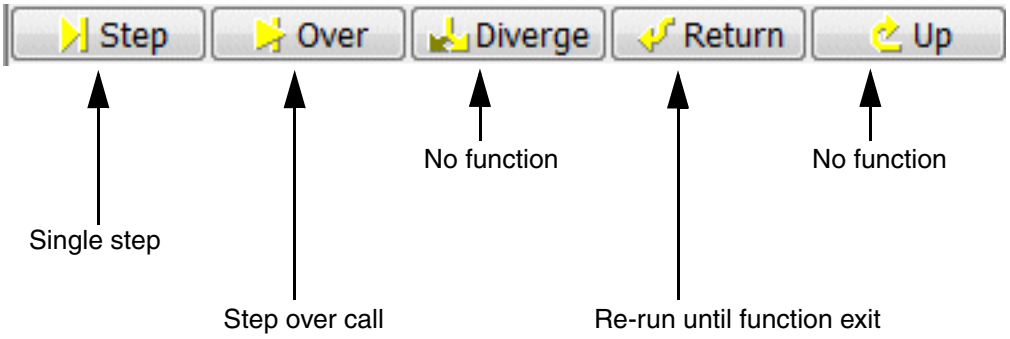


TRACE32 PowerView displays the state of the target as it was when the instruction of the trace record -11009060.0 was executed

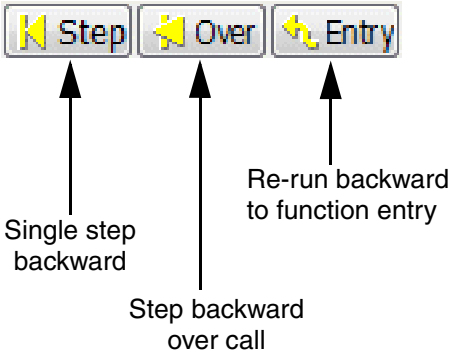
Now you can start to re-run the recorded program section within TRACE32 PowerView by forward or backward debugging.



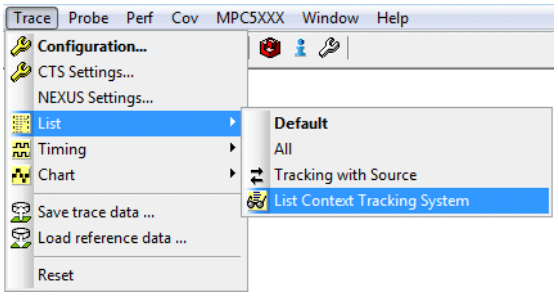
Forward Debugging

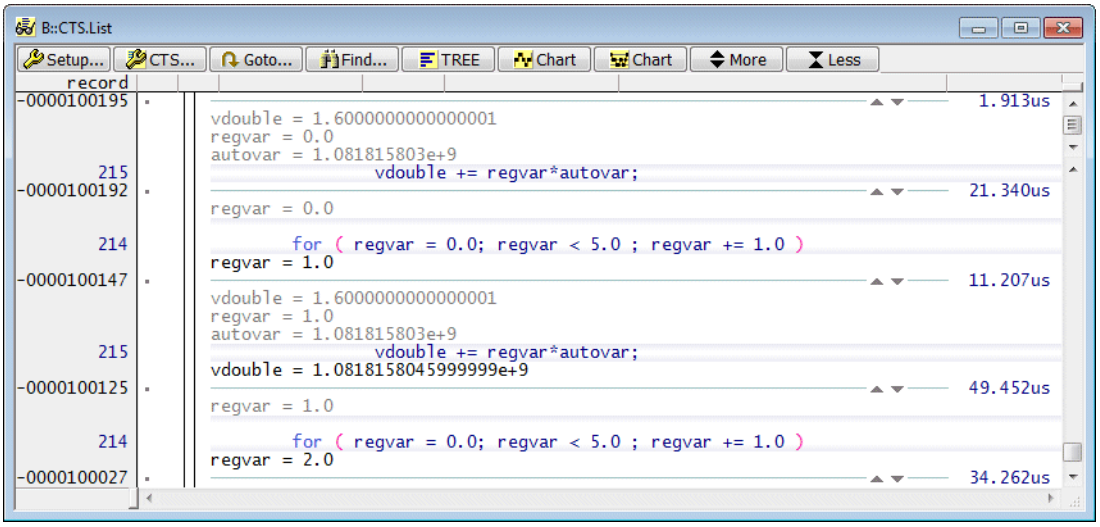


Backward Debugging



The technology used for Trace-Based Debugging allows additionally to display a full HLL trace.

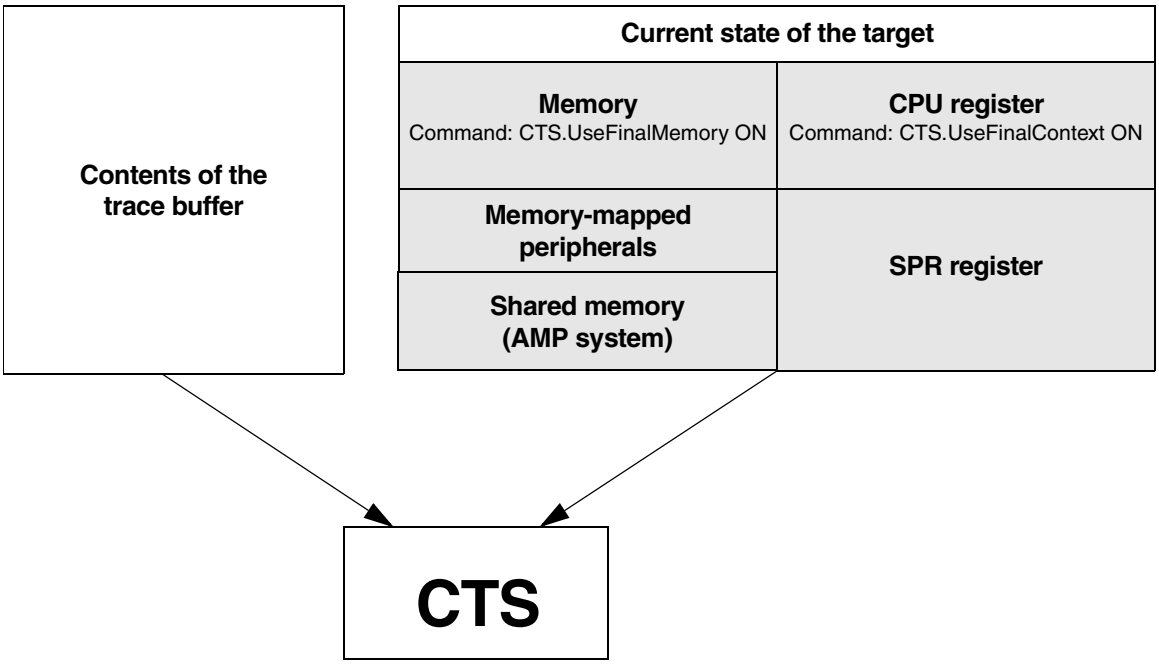




For each HLL step the following information is displayed:

- The values of the local and global variables used in the HLL step
- The result of the HLL step
- The time needed for the HLL step

CTS.List	List pure HLL trace.



CTS reads and evaluates the current state of the target together with the information recorded to the trace memory by default.

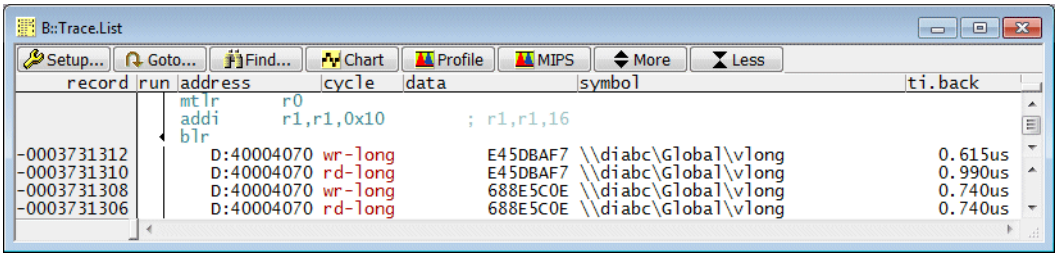
The following commands are used to configure CTS properly:

CTS.UseFinalMemory ON Default setting within TRACE32

If **CTS.UseFinalMemory** is ON and TRACE32 detects that a memory address was not changed by the recorded program section, TRACE32 PowerView displays the current content of this memory in CTS display mode.

- If Data Trace Messaging is disabled (NEXUS.DTM OFF), TRACE32 can not detect which memory content was changed. This is the reason why **CTS.UseFinalMemory** has to be set to OFF.
- If Data Trace Messaging is enabled (NEXUS.DTM ReadWrite) it is not guaranteed, that all read/write accesses are recorded. This is the reason why **CTS.UseFinalMemory** has to be set to OFF.

Please be aware, that CTS ignores all read/write cycles that can not be assigned to its instruction (displayed in red).



MAP.VOLATILE *<range>* Declare specified address range as volatile.

CTS supposes by default that memory is only written by the core(s) for which trace information is recorded into the trace memory. But other bus master such as the DMA controller or other, not recorded cores, can change memory too. And external interfaces can change memory mapped peripheral registers.

All memory ranges, that are not only changed by the core(s) for which trace information is recorded, have to be excluded from the CTS memory/variable reconstruction.

```
MAP.VOLATILE 0xF0000000—0xFFFFFFFF          ; exclude peripheral register
                                            ; address space from the CTS
                                            ; reconstruction

MAP.VOLATILE 0x40018000--0x4001BFFF         ; exclude memory that is
                                            ; changed not only by the
                                            ; recorded core(s)
                                            ; from the CTS reconstruction
```

If Data Trace Messaging is disabled (NEXUS.DTM OFF) and **CTS.UseFinalMemory** is switch OFF, but your target memory contains constants, you can configure TRACE32 to use these constants for the CTS reconstruction by the following commands:

MAP.CONST *<address_range>*

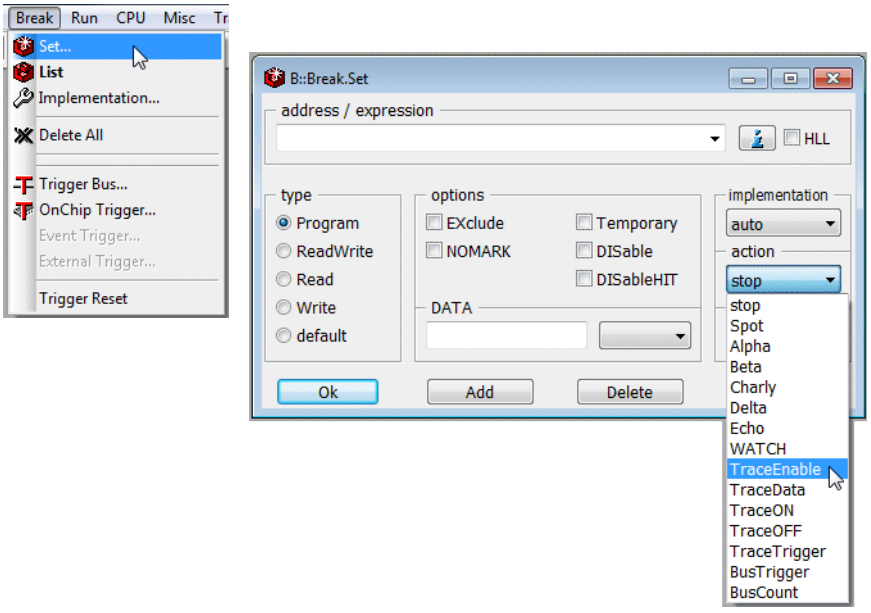
CTS.UseConst ON

CTS.UseFinalContext ON Default setting within TRACE32

If **CTS.UseFinalContext** is ON and TRACE32 detects that a register was not changed by the recorded program section, TRACE32 PowerView displays the current content of this register in CTS View mode.

CTS.UseFinalContext has to be set to OFF, if you used **Stack** mode for tracing recording.

Filter and Trigger (Core) Overview



TraceEnable, TraceData, TraceON and TraceOFF are so-called filters. **Filters** can be used advise the NEXUS module to generate trace information only for events of interest.

TraceEnable: Advise the NEXUS module to generate trace messages only for the specified instruction(s) or read/write accesses.

TraceData: Advise the NEXUS module to generate trace messages for all executed instructions and for the specified read/write accesses.

TraceON: Advise the NEXUS module to start the generation of trace messages at the specified event.

TraceOFF: Advise the NEXUS module to stop the generation of trace messages at the specified event.

TraceTrigger, BusTrigger and BusCount are so-called triggers. **Triggers** can be used to advise the NEXUS module to signal the occurrence of an event. TRACE32 can react on this occurrence by stopping the trace recording, by counting the event ….

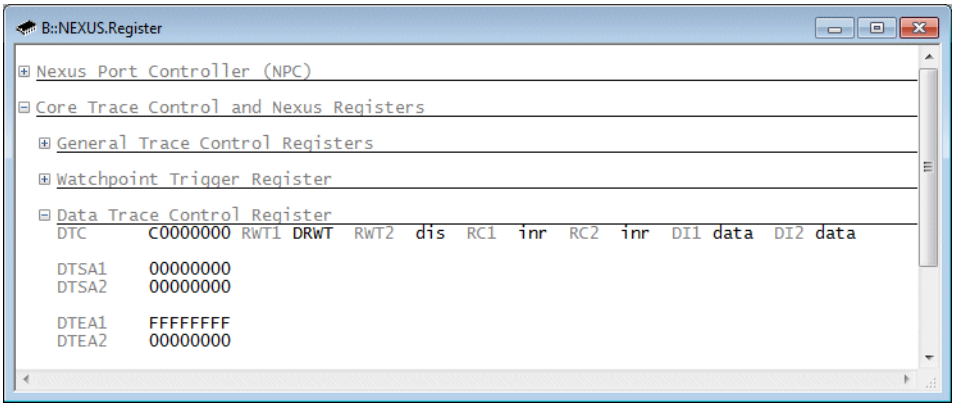
TraceTrigger: Stop the trace recording at the specified event.

BusTrigger: Generate a pulse on the trigger bus at the specified event.

BusCount: Count the specified event.

The MPC5xxx provides the following resources for filter and trigger:

- Data Trace Control Register (DTC): to filter Data Trace Messages (2-4 address ranges)




```
NEXUS.Register
```

- Watchpoint Trigger Register: to activate a trace action on a specified event. The source for the specified event are the Watchpoints that are also used for the on-chip breakpoints.

Core type:	On-chip Breakpoints	Instruction Address Breakpoints	Data Address Breakpoints	Data Value Breakpoints
e200z0 e200z0h	4 instruction 2 read/write no counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	none
e200z0Hn3	4 instruction 2 read/write 2 data value no counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	2 single breakpoints (associated with data address BPs)
e200z1 e200z3 e200z6 e200z650 e200z750	4 instruction 2 read/write 2 counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	none

Core type:	On-chip Breakpoints	Instruction Address Breakpoints	Data Address Breakpoints	Data Value Breakpoints
e200z335	4 instruction 2 read/write 2 data value 2 counters	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints -- or -- 1 breakpoint range	2 single breakpoints (associated with data address BPs)
e200z446 e200z4d e200z760	8 instruction 2 read/write 2 data value 2 counters	8 single breakpoints -- or -- 2 breakpoint ranges and 4 single breakpoints	2 single breakpoints -- or -- 1 breakpoint range	2 single breakpoints (associated with data address BPs)
e200z210 e200z215 e200z225 e200z420 e200z425 e200z720 e200z4201 e200z4203 e200z4204 e200z4251 e200z7260	8 instruction 4 read/write 2 data value no counters	8 single breakpoints -- or -- 4 breakpoint ranges	4 single breakpoints -- or -- 2 breakpoint ranges	2 single breakpoints (associated with data address BPs)

- The **MPC57xx** provides also means to control Program Trace Messaging and Data Trace Messaging from the application.

  Nexus Development Control Register:

PTMARK Bit 1	Program Trace Messaging when PMM bit is set
DTMARK Bit 1	Data Trace Messaging when PMM bit is set

Machine Status Register:

PMM Bit	Performance monitor mark bit. PMM Bit 1, PTMARK Bit 1 -> Program Trace Messaging is enabled PMM Bit 1, DTMARK Bit 1 -> Data Trace Messaging is enabled

The table below summarizes the influence of the filter/ trigger on the messaging.

	WTM Watchpoint Trace Messages	**BTM** Branch Trace Messages	**DTM** Data Trace Messages	**OTM** Ownership Trace Messages	**DQM** Data Acquisition Messages
TraceEnable on single instruction	Watchpoint Hit Message for instruction	Disabled	Unaffected	Unaffected	Unaffected
TraceEnable on instruction range	Unused	Filter applies	Filter applies	Unaffected	Unaffected
TraceEnable on read/write access	Unused	BTM disabled	DTM enabled Filter applies	Unaffected	Unaffected
TraceData	Unused	Unaffected	DTM enabled Filter applies	Unaffected	Unaffected
Global TraceON/ TraceOFF	Unused	Filter applies	Filter applies	Unaffected	Unaffected
Program TraceON/ TraceOFF	Unused	BTM enabled Filter applies	Unaffected	Unaffected	Unaffected
Data TraceON/ TraceOFF	Unused	Unaffected	Filter applies	Unaffected	Unaffected
TraceTrigger BusTrigger BusCount	WHM for instruction or data address/data value	Unaffected	Unaffected	Unaffected	Unaffected

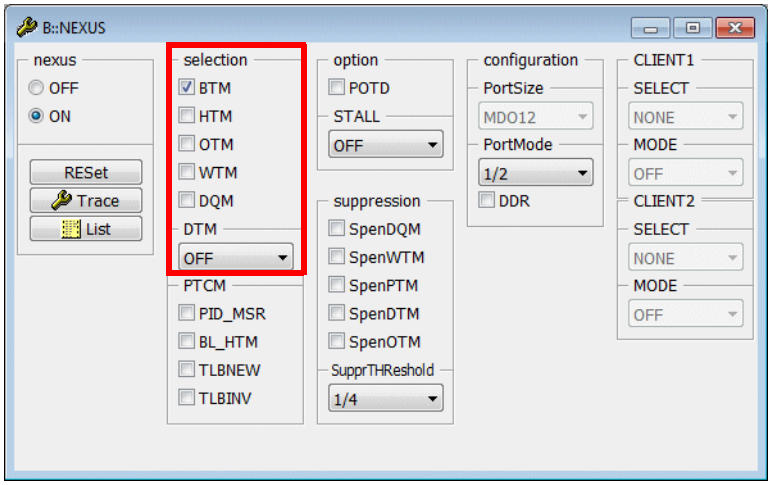
Examples for TraceEnable on Instructions

Resource: Watchpoints

Controlled message types

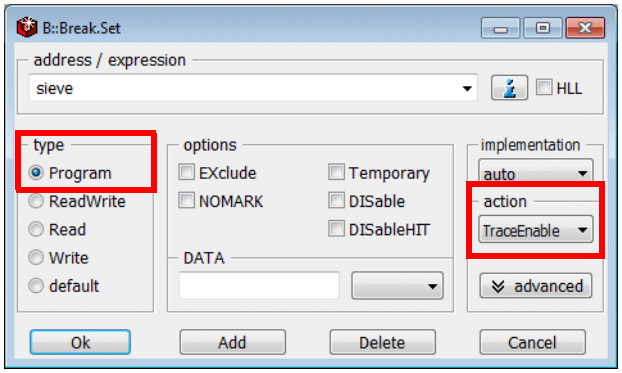
WTM Watchpoint Trace Messages	**BTM** Branch Trace Messages	**DTM** Data Trace Messages	**OTM** Ownership Trace Messages	**DQM** Data Acquisition Messages
Watchpoint Hit Message(s) is generated for the specified instruction(s)	Disabled	Unaffected	Unaffected	Unaffected

Disable message types, that are unaffected by the filter and not required for your analysis.



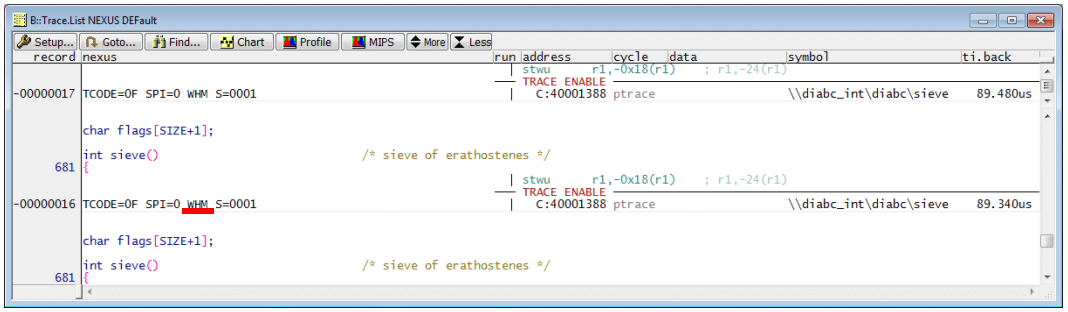
Example 1: Advise the NEXUS module to generate only trace information for the entries to the function sieve.

1. Set a Program breakpoint to the start address of the function sieve and select the action TraceEnable.



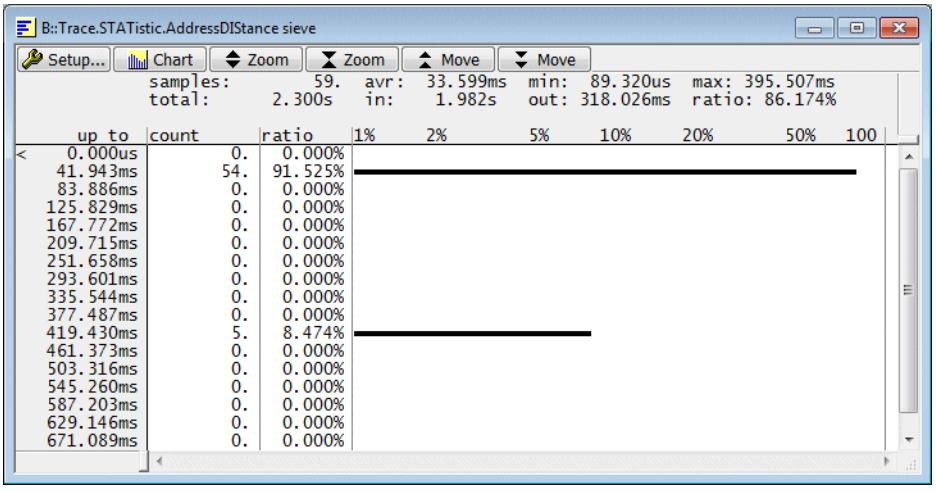
2. Start the program execution and stop it.

3. Display the result.



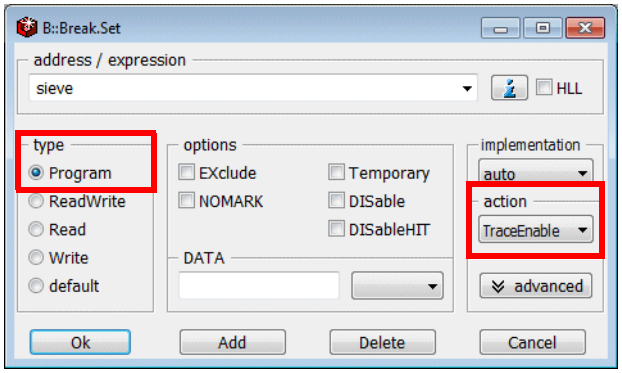
The following **Trace.STATistic** command calculates the time intervals for a program address event. The program address event is here the entry to the function sieve:

```
Trace.STATistic.AddressDIStance sieve
```

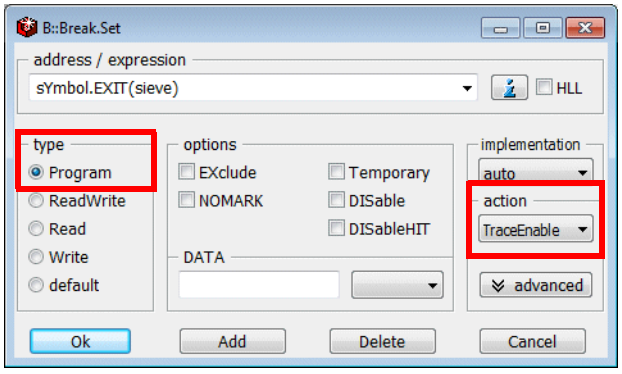


Example 2: Advise the NEXUS module to generate trace information for the entries to the function sieve and for the exits of the function sieve.

1. Set a Program breakpoint to the start address of the function sieve and select the action TraceEnable.



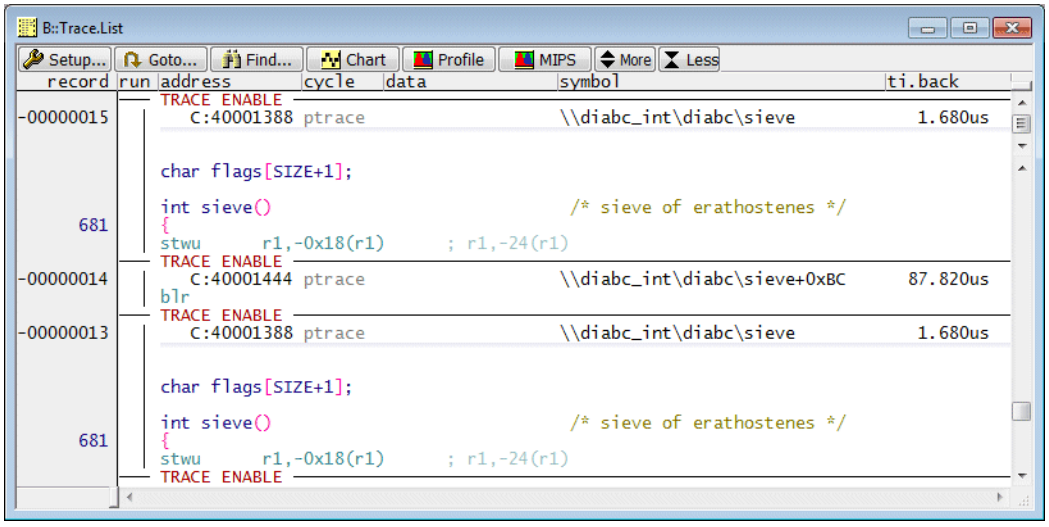
2. Set a Program breakpoint to the exit address of the function sieve and select the action TraceEnable.



sYmbol.EXIT(<*symbol*>**)**	Returns the exit address of the specified function

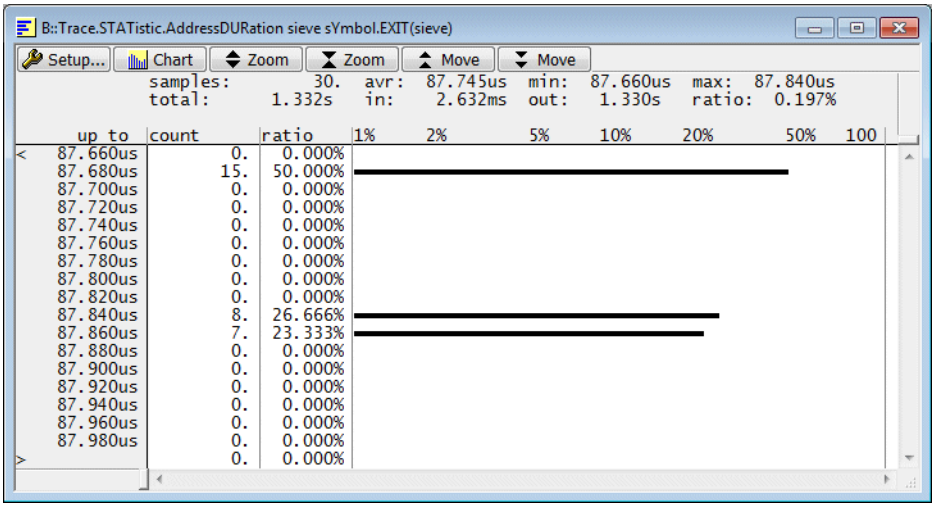
3. Start the program execution and stop it.

4. Display the result.



The following **Trace.STATistic** command calculates the time intervals between two program address events A and B. The entry to the function sieve is A in this example, the exit from the function is B.

```
Trace.STATistic.AddressDURation sieve sYmbol.EXIT(sieve)
```



Example for TraceEnable on Instruction Range

Resource: Watchpoints**,** limited to one instruction address range

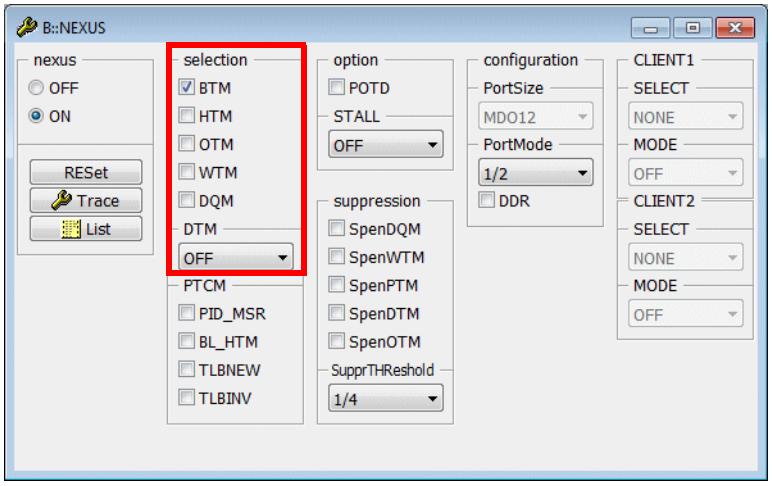
Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	Filter applies if BTM is enabled	Filter applies if DTM is enabled	Unaffected	Unaffected

Enable BTM. This filter requires that Branch History messaging is disabled.

Enable DTM if you are interested in the read/write accesses performed by the specified instruction address range.

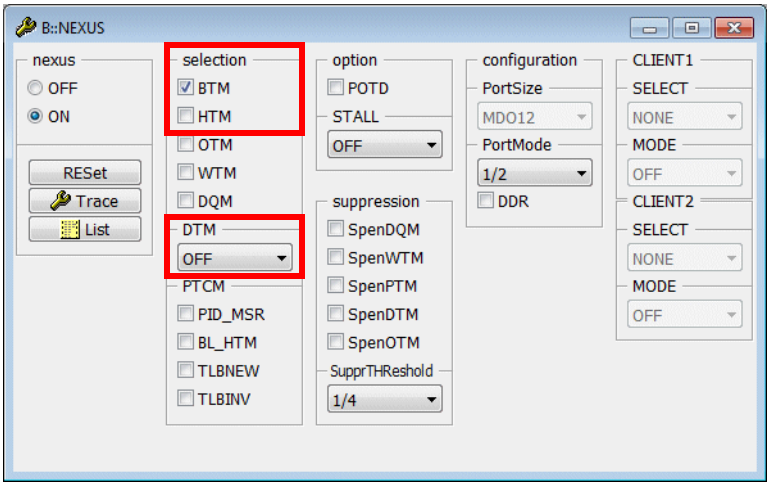
Disable message types, that are unaffected by the filter and not required for your analysis.



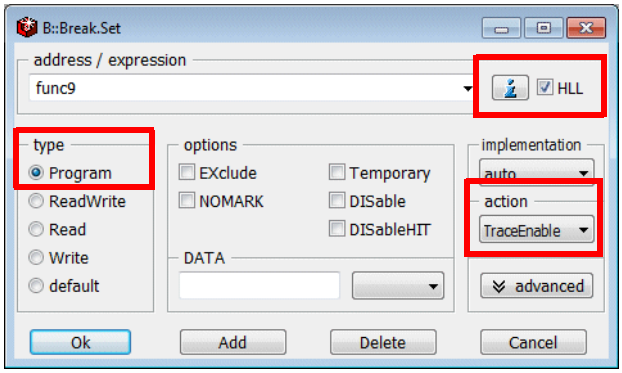
Example: Advise the NEXUS module to generate trace information for all taken branches within the function func9.

1. Enable Branch Trace messaging, but don't enable Indirect Branch History messaging.

   Disable Data Trace messaging.

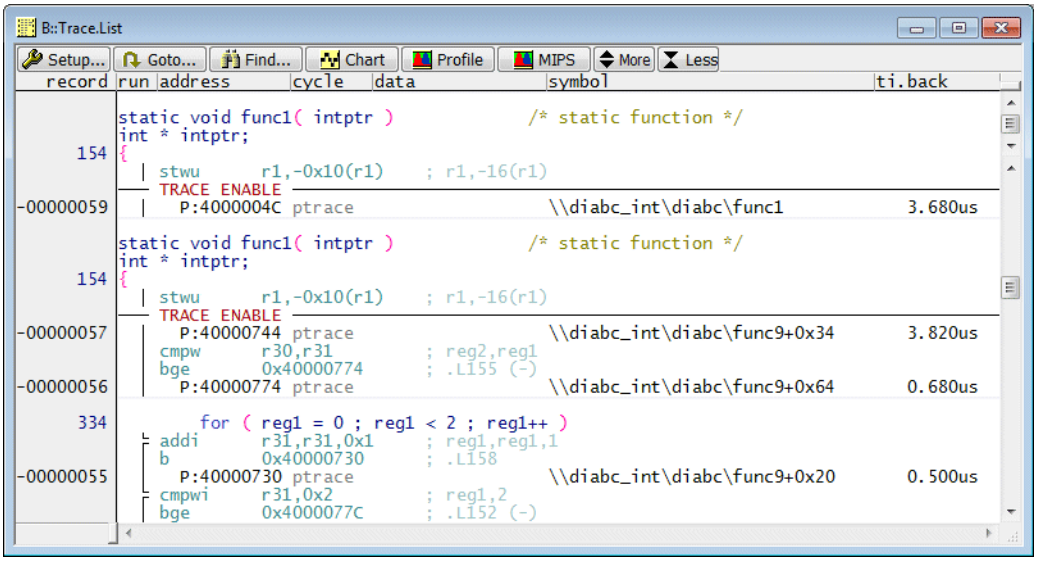


2. Set a Program breakpoint to the complete address range of the function func9 (HLL check box ON) and select the action TraceEnable.



3. Start the program execution and stop it.

4. Display the result.



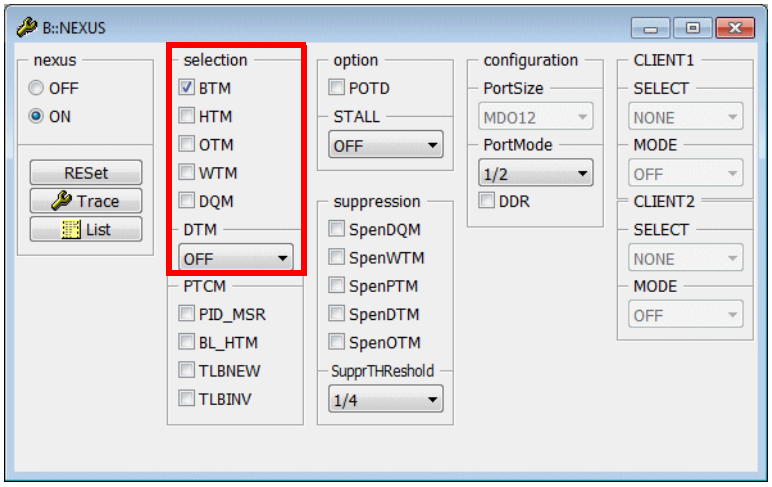
Examples for TraceEnable on Read/Write Accesses

Resource: DTC Register

Controlled message types

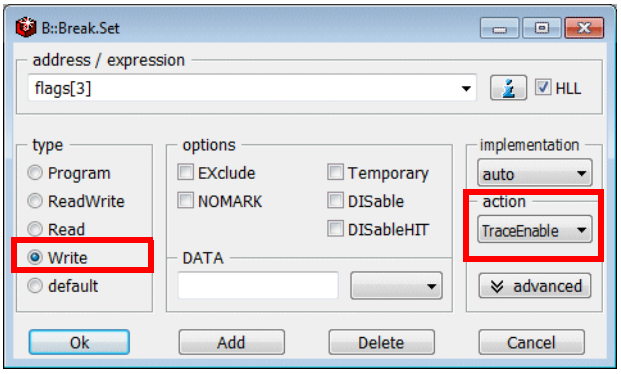
WTM	BTM	DTM	OTM	DQM
Unused	BTM is disabled by filter	DTM is enabled by filter Filter applies	Unaffected	Unaffected

Disable message types, that are unaffected by the filter and not required for your analysis.



Example: Disable Branch Trace messaging and advise the NEXUS module to only generate trace information for the write accesses to the variable flags[3].

1. Set a Write breakpoint to the variable flags[3] and select the action TraceEnable

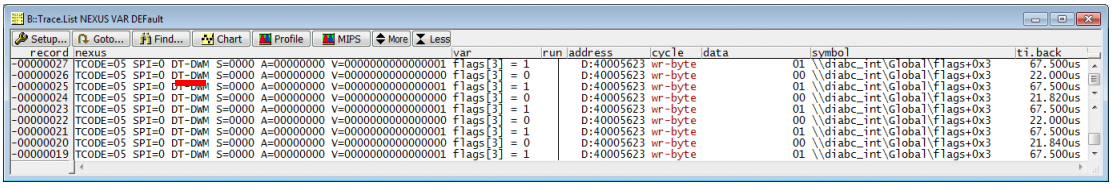


- no data value possible (limitation of DTC Register)

- accessing instruction not possible (limitation of DTC Register)

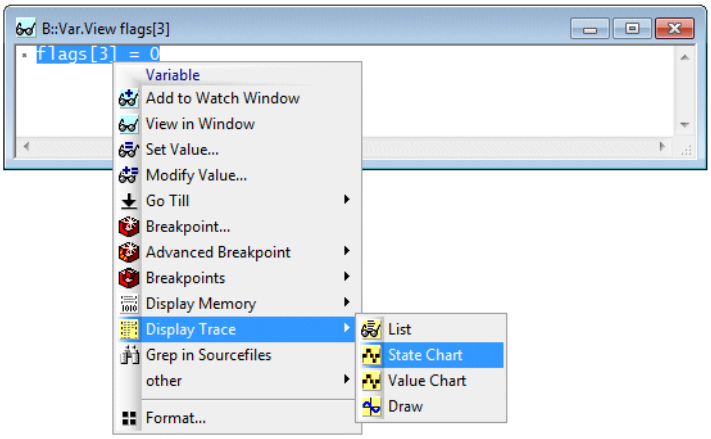
2. Start the program execution and stop it.

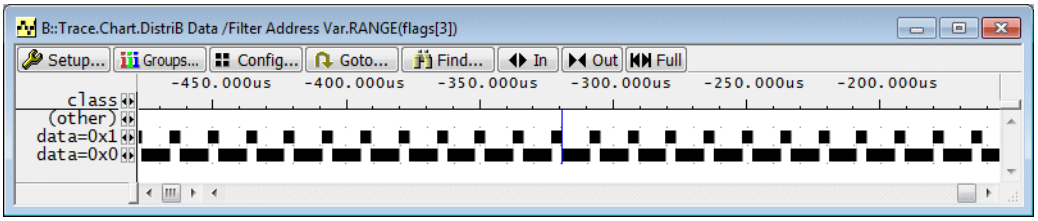
3. Display the result.



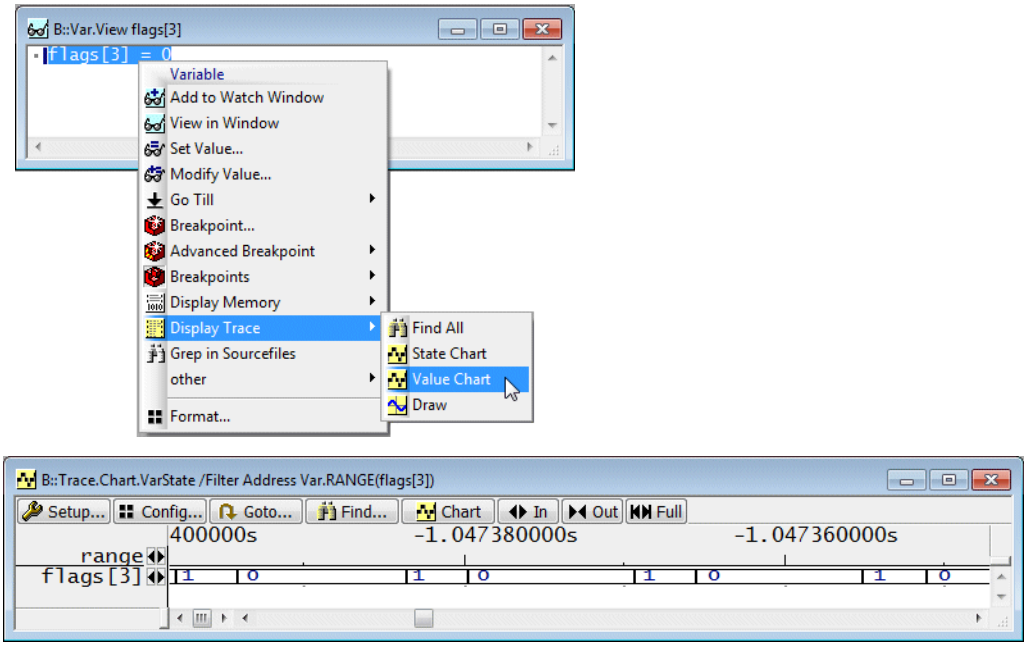
The Variable pull-down provides various way to analyze the variable contents over the time.

```
; open a window to display the variable
Var.View flags[3]
```

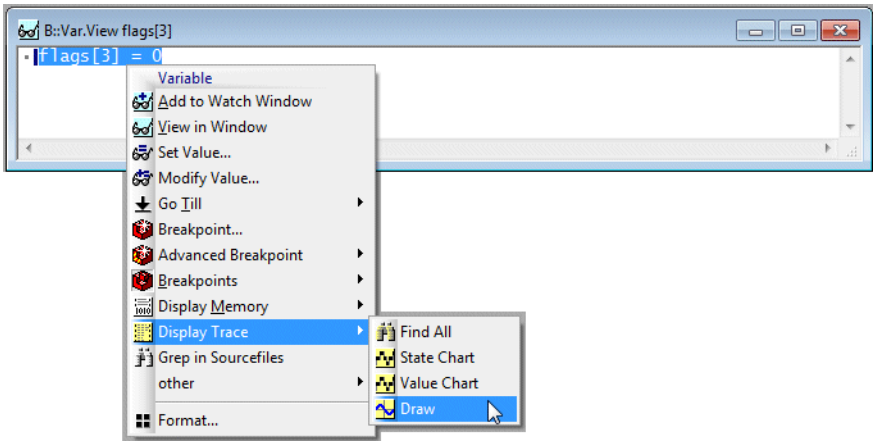


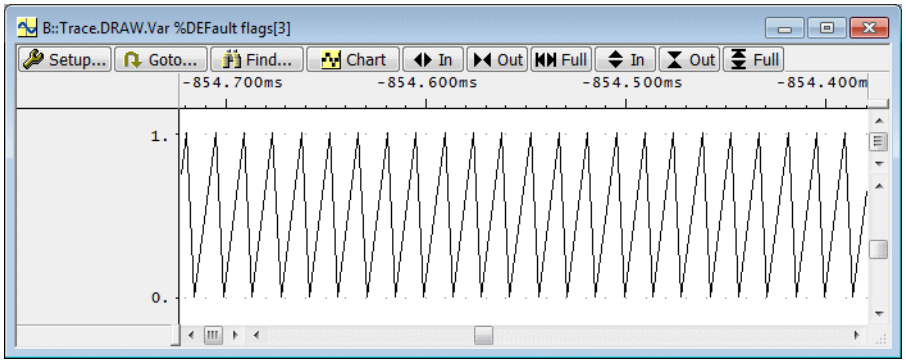


Display the value changes of a variable graphically
Trace.Chart.DistriB Data /Filter Address Var.RANGE(<*var*>**)**



Display variable contents over the time numerically
Trace.Chart.VarState





Display variable contents over the time graphically
Trace.DRAW.Var %DEFault *<var>*

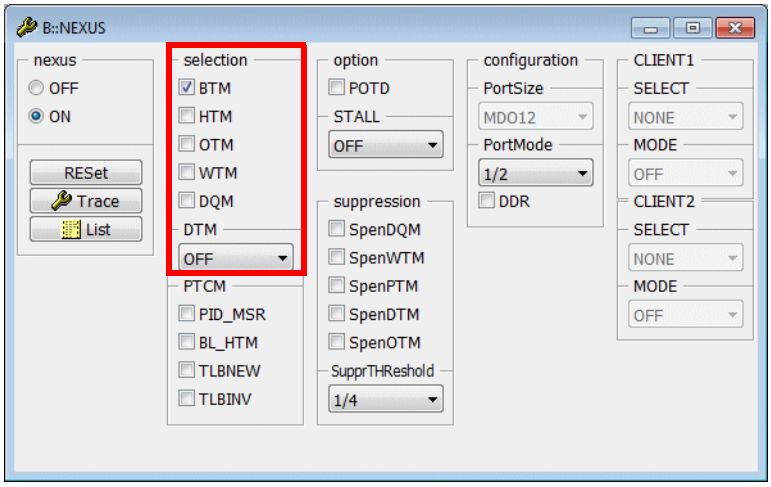
Example for TraceData

Resource: DTC Register

Controlled message types

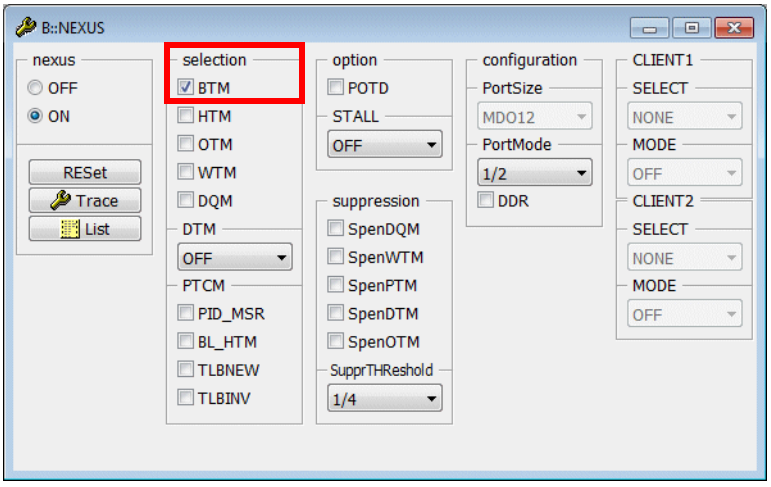
WTM	BTM	DTM	OTM	DQM
Unused	Unaffected	DTM is enabled by filter Filter applies	Unaffected	Unaffected

Disable message types that are unaffected by the filter and not required for the analysis.

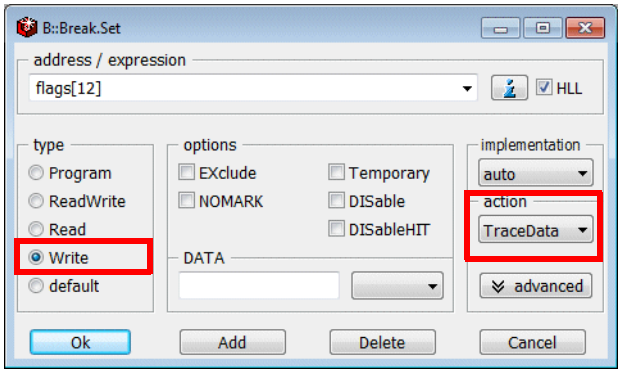


Example: Advise the NEXUS module to generate trace information for the write accesses to flags[12] and to generate trace information for all executed instructions.

1. Enable Branch Trace messaging.

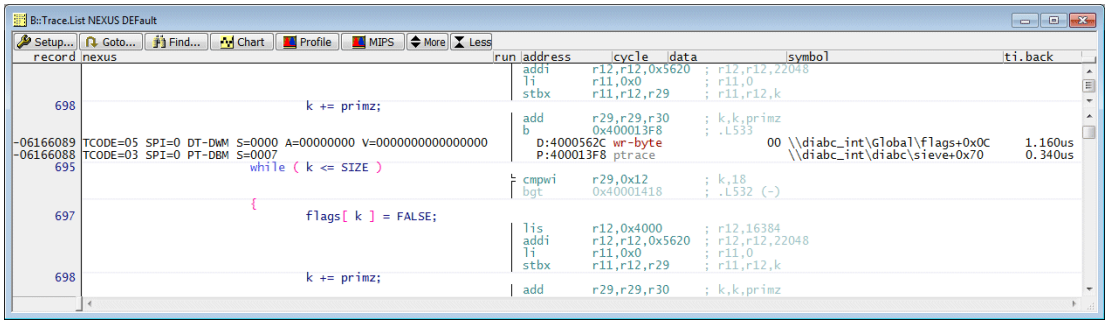


2. Set a Write breakpoint to the variable flags[12] and select the action TraceData.



3. Start the program execution and stop it.

4. Display the result.



Please be aware that in the case of a TraceData filter a correlation of the data access and the instruction is in most cases not possible.

Global TraceON/Trace OFF

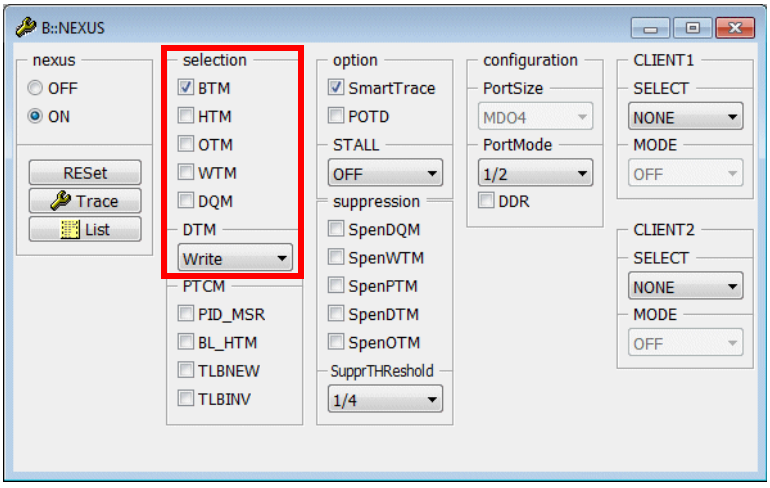
Resource: Watchpoints

Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	Filter applies	Filter applies	Unaffected	Unaffected

Enable Branch Trace Messaging and Data Trace Messaging if this information is required for your analysis.

Disable messages types that are unaffected and not required for the analysis.

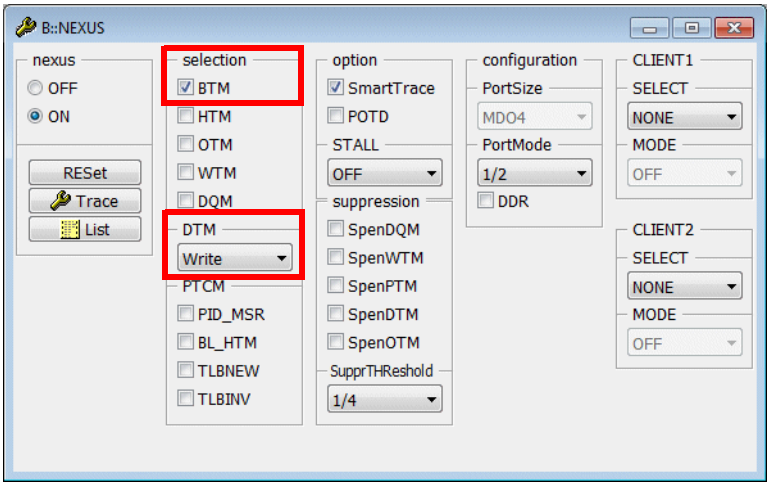


Example:

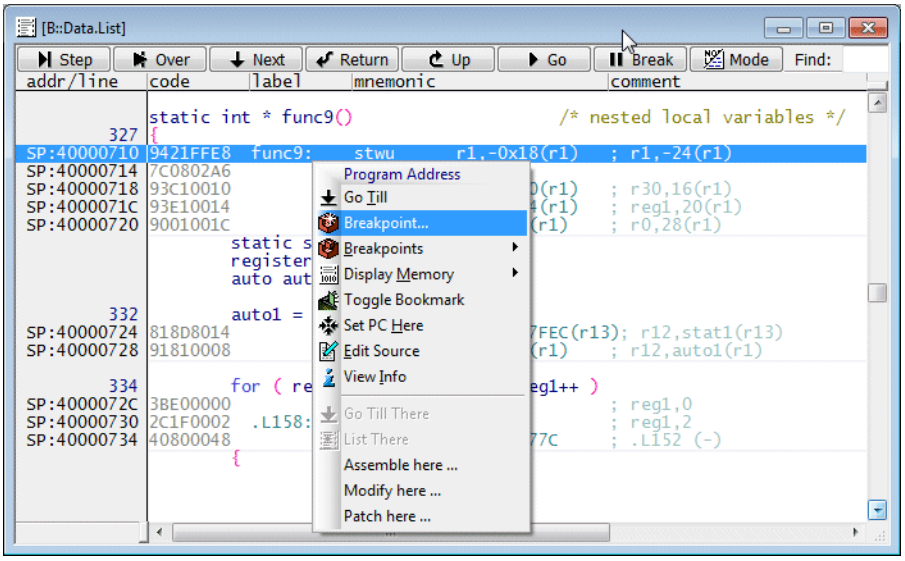
Advise the NEXUS module to start Branch Trace messaging and Data Write Messages at the entry to the function func9.

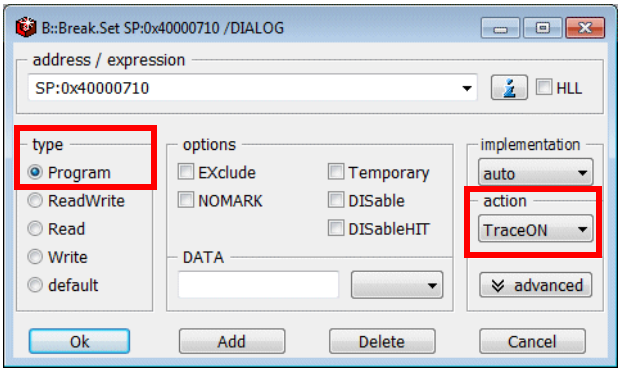
Advise the NEXUS module to stop Branch Trace messaging and Data Write Messages at the exit of the function func9.

1. Enable Branch Trace Messages and Data Write Messages.

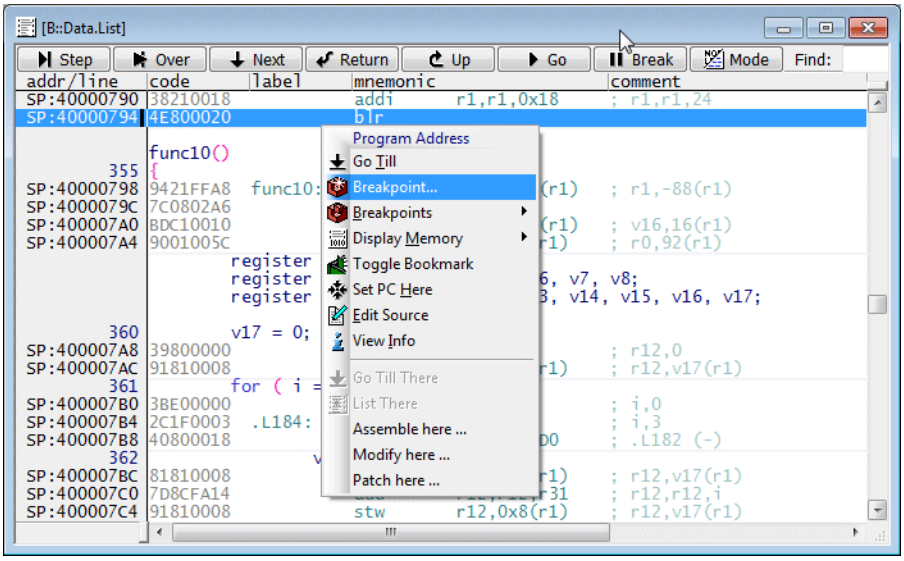


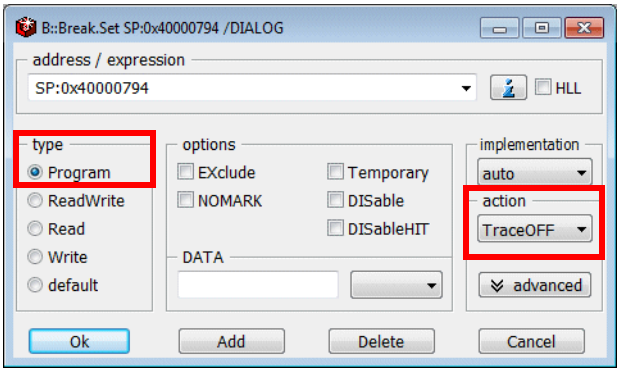
2. Set a Program breakpoint to the entry of the function func9 and select the action TraceON.

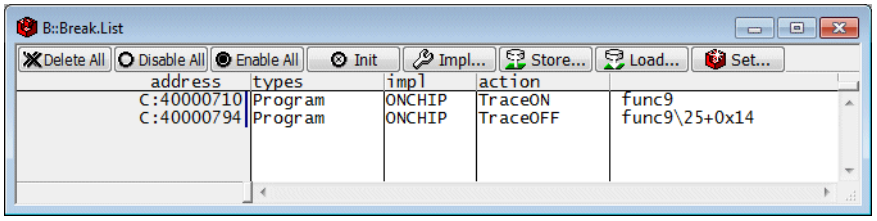




3. Set a Program breakpoint to the exit of the function func9 and select the action TraceOFF.

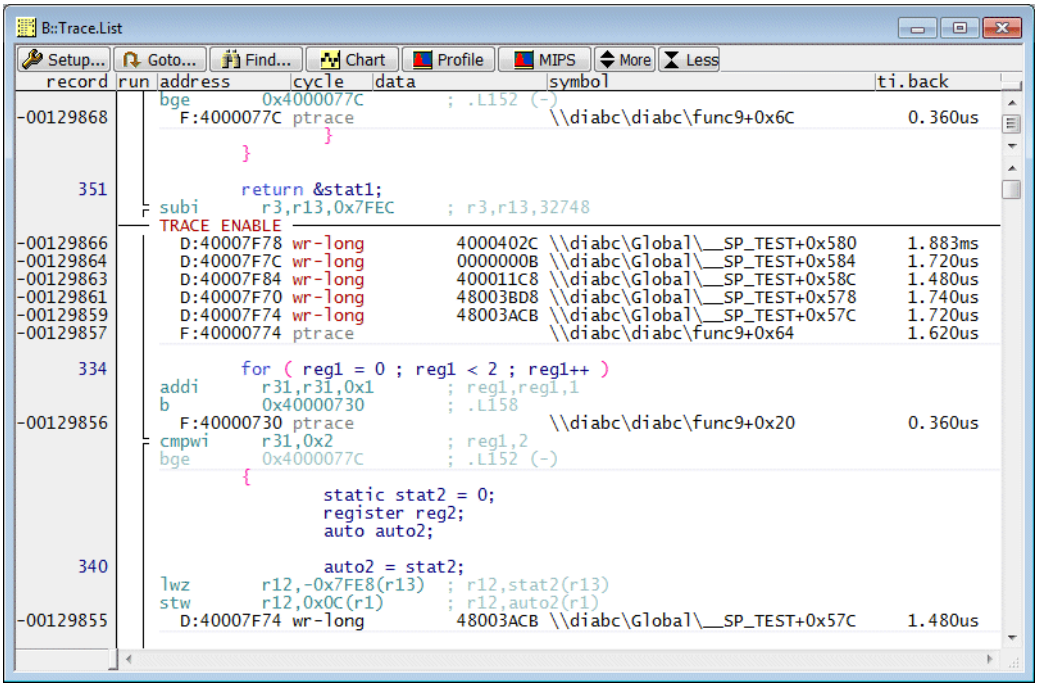






4. Start the program execution and stop it.

5. Display the result.



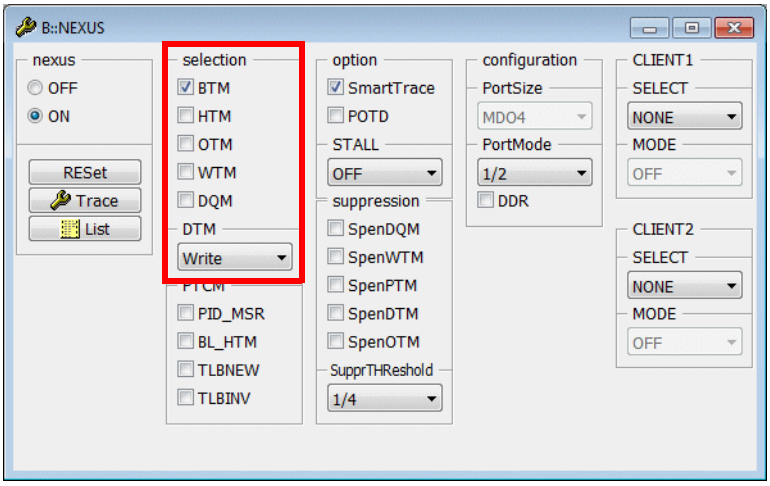
The event that switched the trace generation on is not visible in the trace.

Resource: Watchpoints

Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	BTM is enabled by filter Filter applies	Unaffected	Unaffected	Unaffected

Disable messages types that are unaffected and not required for the analysis.



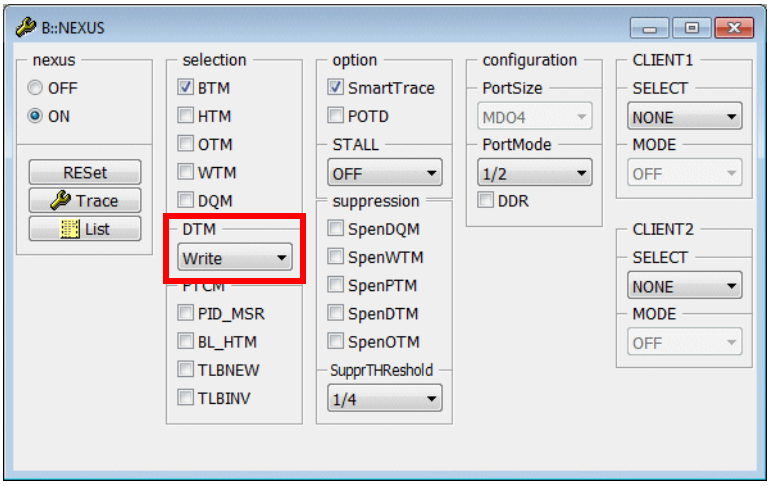
Example:

Advise the NEXUS module to generate trace information for all write accesses.

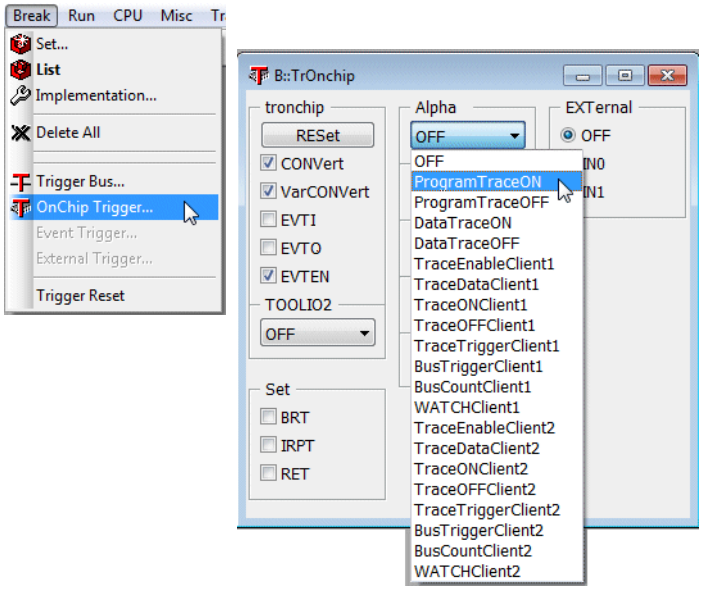
Advise the NEXUS module to start the Branch Trace messaging at the entry to the function func9.

Advise the NEXUS module to stop Branch Trace messaging at the exit of the function func9.

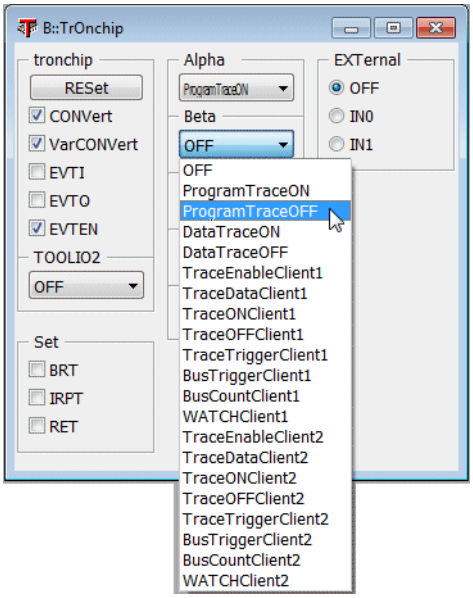
1. Enable Data Trace messaging for write accesses.



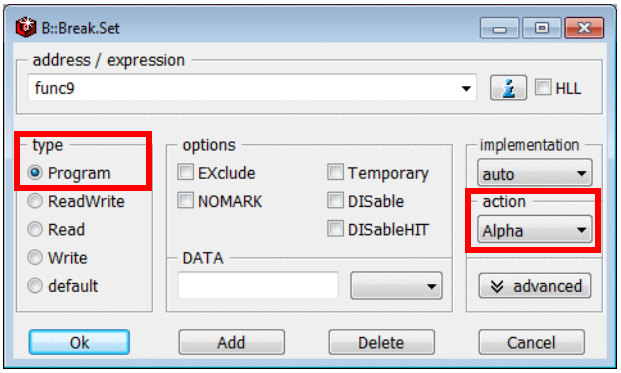
2. Open the TrOnchip window and select ProgramTraceON for Alpha.



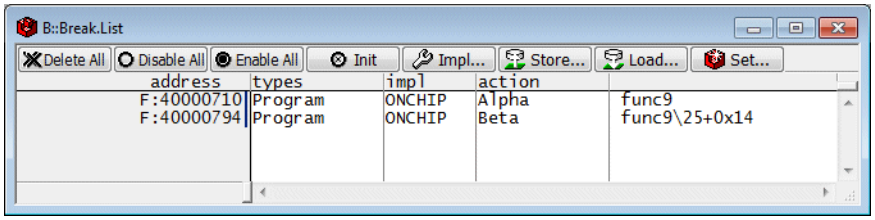
3. Select ProgramTraceOFF for Beta.



4. Set a Program breakpoint to the entry of the function func9 and select Alpha.

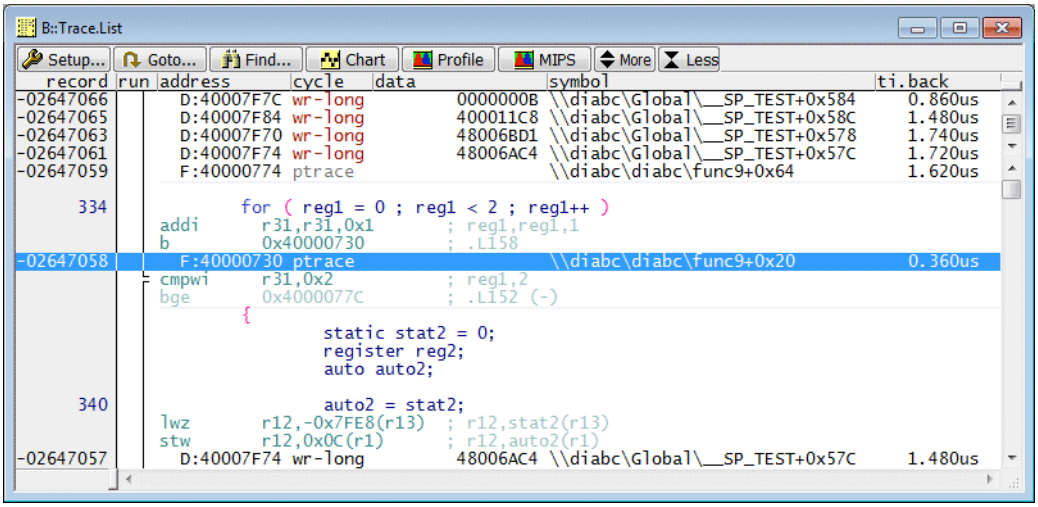


5. Set a Program breakpoint to the exit of the function func9 and select Beta.



6. Start and stop the program execution.

7. Display the result.



Command line example

```
; establish a default start situation
Break.Delete /ALL
TrOnchip.RESet

; messaging setup
NEXUS.BTM ON
NEXUS.DTM Write

; filter settings
TrOnchip.Alpha ProgramTraceON
TrOnchip.Beta ProgramTraceOFF
Break.Set func9 /Program /Alpha
Break.Set sYmbol.EXIT(func9) /Program /Beta

Go

…

Break

; display result
Trace.List
```

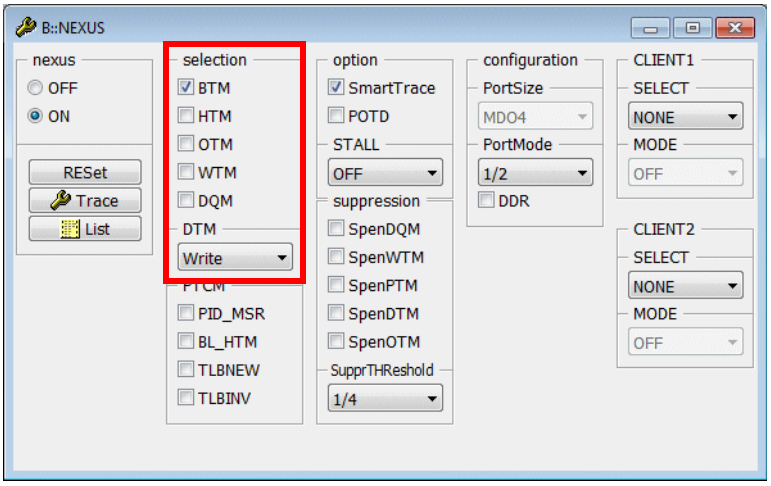
Resource: Watchpoints

Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	Unaffected	Filter applies	Unaffected	Unaffected

Enable Data Trace messaging as required for the analysis.

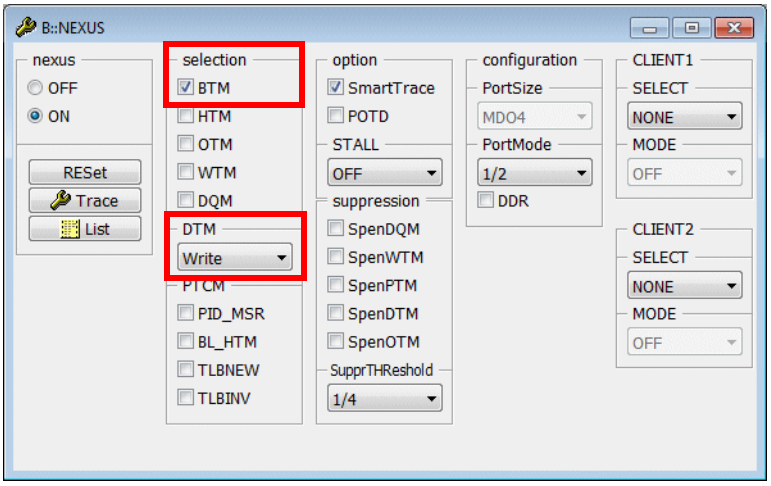
Disable messages types that are unaffected and not required for the analysis.



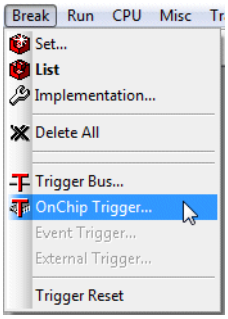
Example:

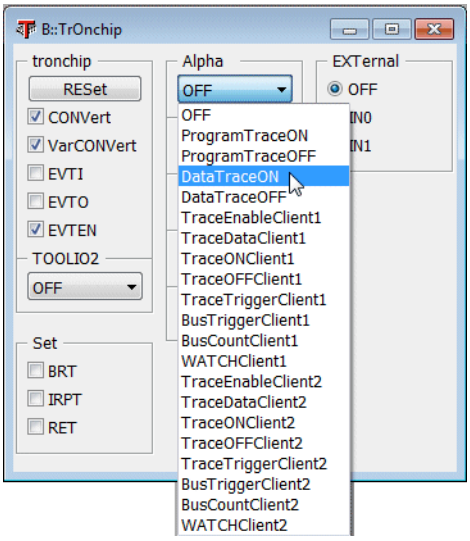
Enable Branch Trace messaging. Advise the NEXUS module to start the generation of Data Write Messages at the entry to the function func9. Advise the NEXUS module to stop the generation of Data Write Messages at the exit of the function func9.

1. Enable Branch Trace messaging and Data Trace messaging for write accesses.

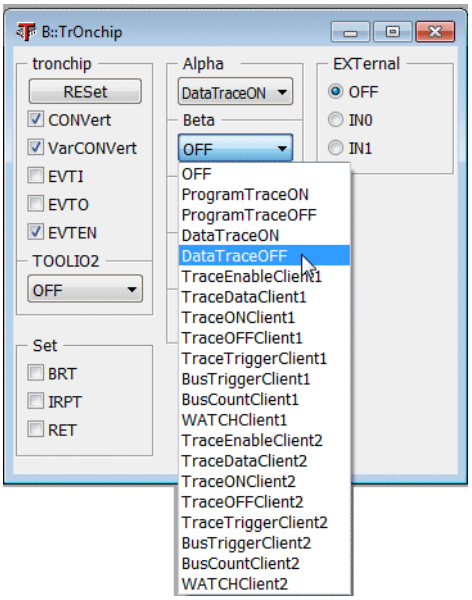


2. Open the TrOnchip window and select DataTraceON for Alpha.

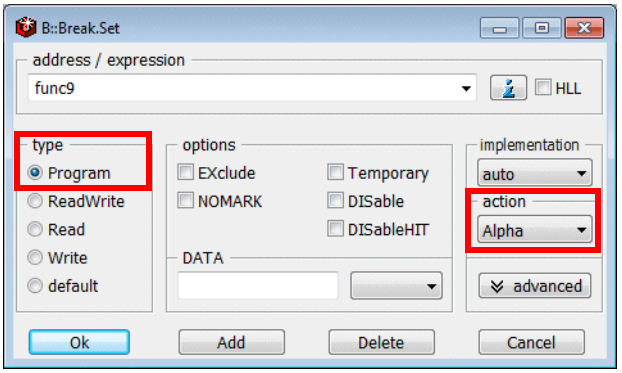




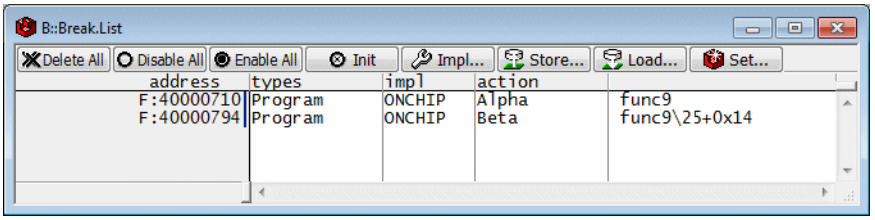
3. Select DataTraceOFF for Beta.



4. Set a Program breakpoint to the entry of the function func9 and select Alpha.

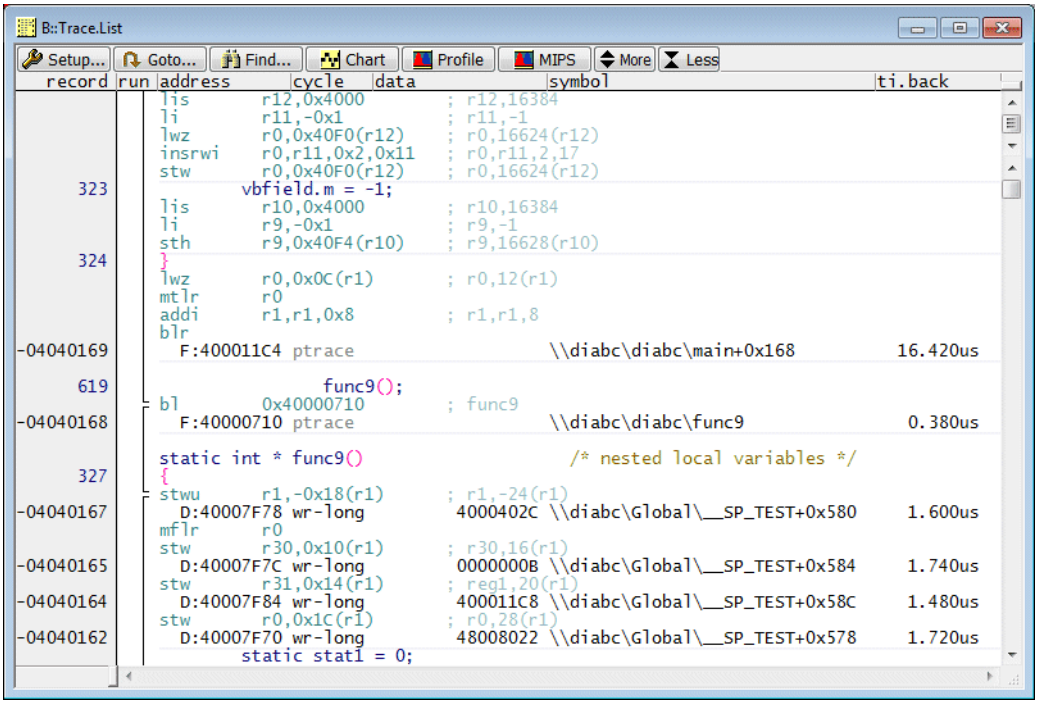


5. Set a Program breakpoint to the exit of the function func9 and select Beta.



6. Start and stop the program execution.

7. Display the result.



Example for TraceTrigger

Resource: Watchpoints and logic in NEXUS Adapter (parallel trace only)

Controlled message types

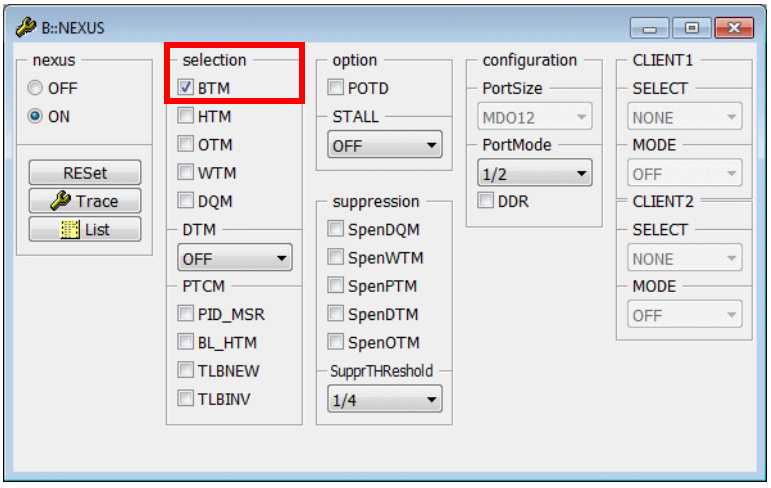
WTM	BTM	DTM	OTM	DQM
Watchpoint Hit Message(s) is generated for the specified instruction(s) or data address+data value	Unaffected	Unaffected	Unaffected	Unaffected

Disable messages types that are unaffected and not required for the analysis.

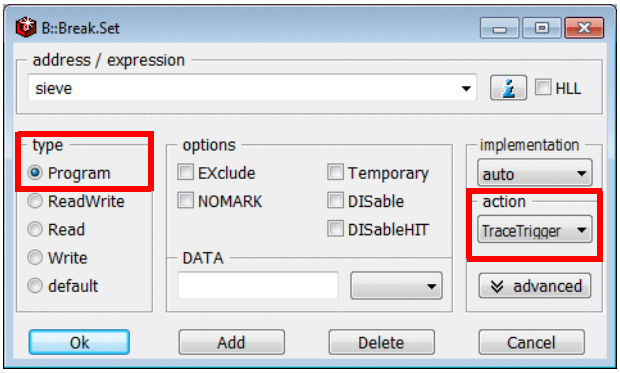
Example:

Enable Branch Trace messaging. Advise the NEXUS module to generate a trigger for the trace if the function sieve is entered. Use this trigger to stops the trace recording.

1. Enable Branch Trace messaging.

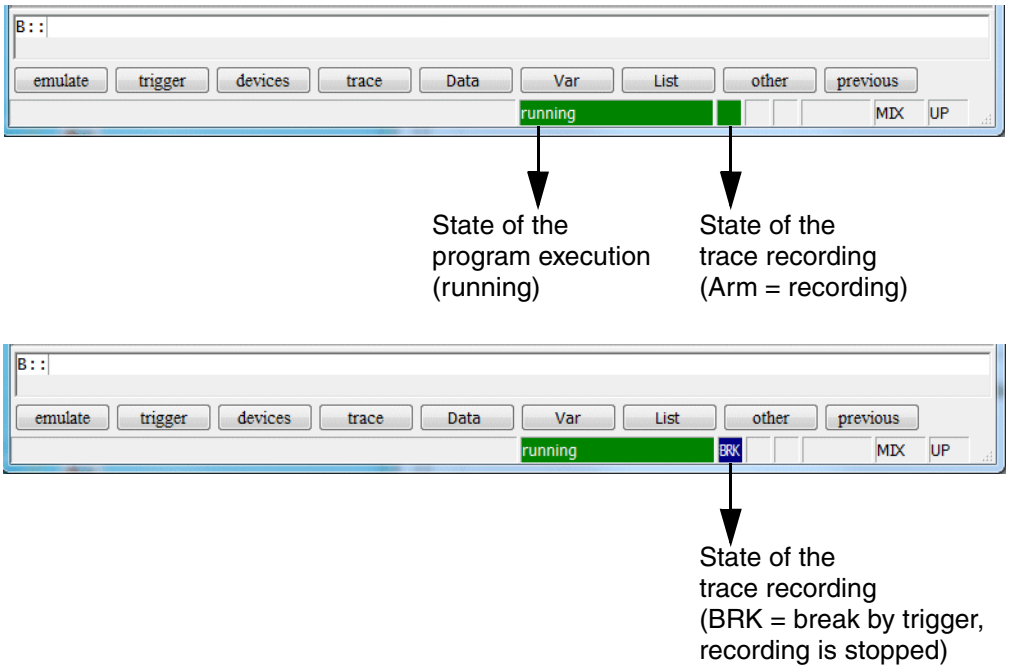


2. Set a Program breakpoint to the start address of the function sieve and select the action TraceTrigger.

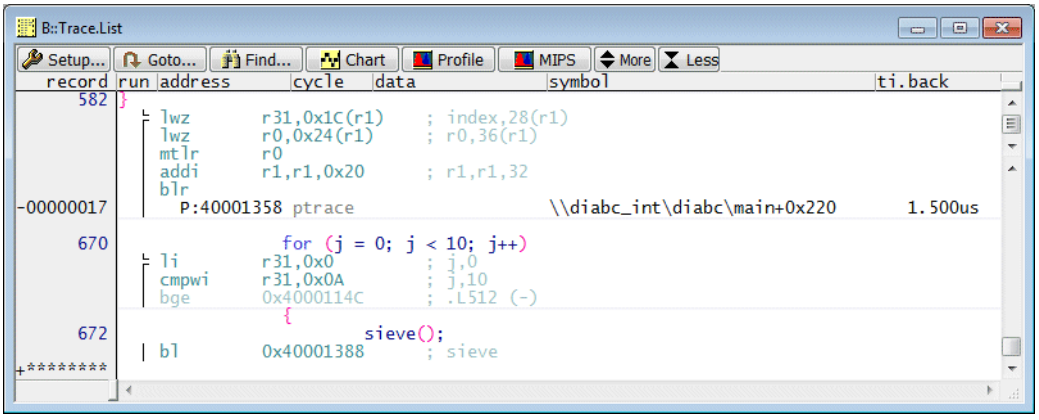


3. Start the program execution

The state of the trace changes from Arm to BRK when the trigger occurs.



4. Display the result.

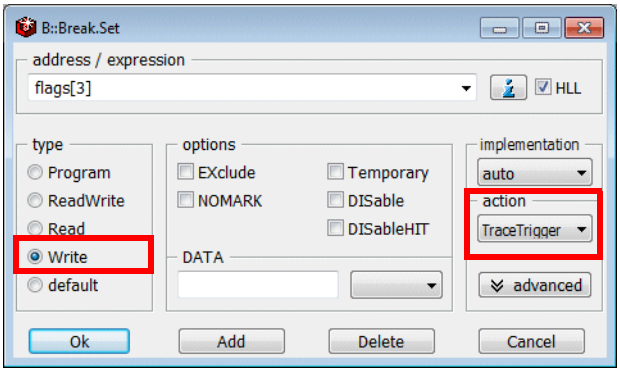


The trace generation is usually stopped before trace information is generated for the event that caused the trigger.

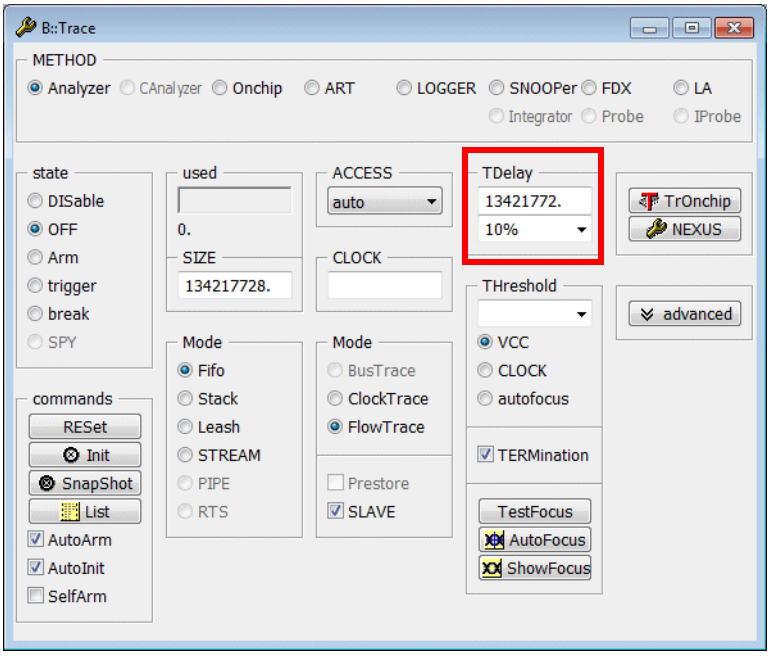
Example for TraceTrigger with a Trigger Delay

Example: Advise the NEXUS module to generate a trigger for the trace if a write access occurs to the variable flags[3]. Advise TRACE32 to fill another 10% of the trace memory before the trace recording is stopped.

1. Set a Write breakpoint to the variable flags[3] and select the action TraceTrigger.



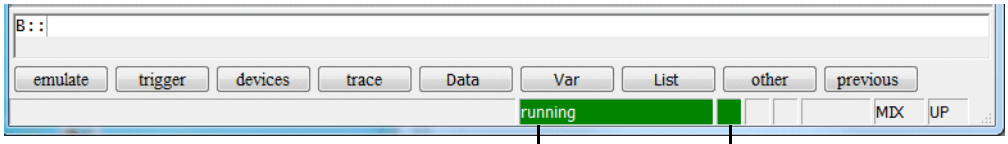
2. Define the trigger delay in the ***Trace Configuration*** Window.



3. Start the program execution.

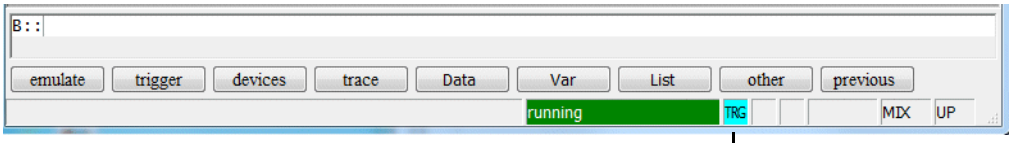
The state of the trace changes from Arm to TRG when the trigger occurs.

The state of the trace changes from TRG to BRK when the delay counter elapses.

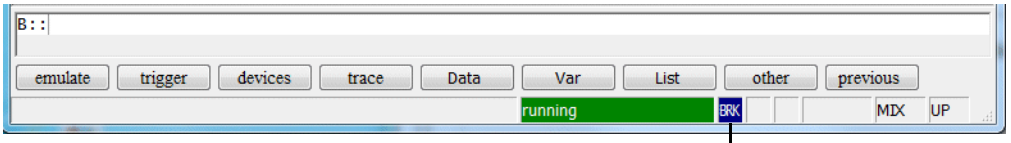


State of the
program execution
(running)

State of the
trace recording
(Arm = recording)

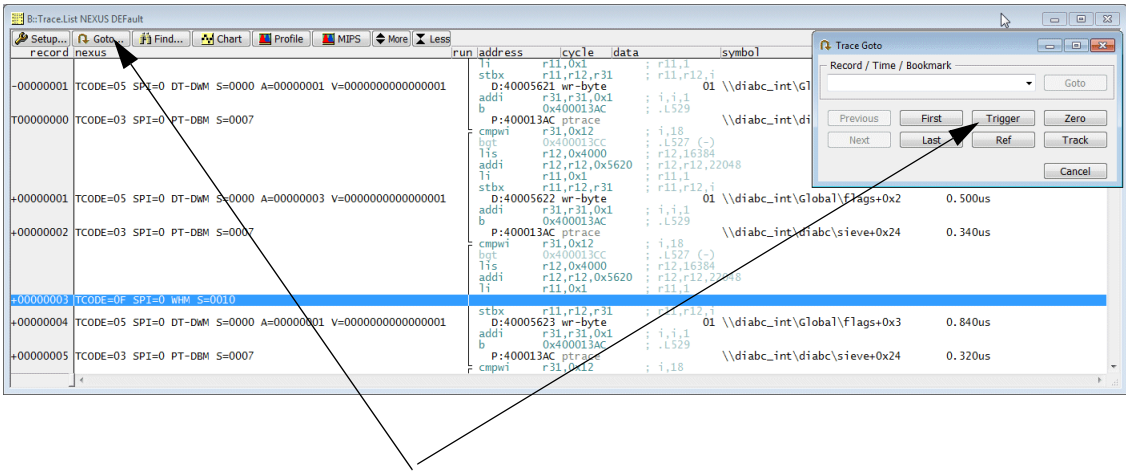


State of the
trace recording
(TRG = trigger occurred,
delay counter started)



State of the
trace recording
(BRK = delay counter elapsed,
recording is stopped)

4. Display the result.



Push the ***Trigger*** button in the ***Trace Goto*** window to find the record, where TraceTrigger was detected by the trace (WHM message). Here the sign of the record numbers has changed. The TraceTrigger event is usually shortly after this point.

Example for BusTrigger

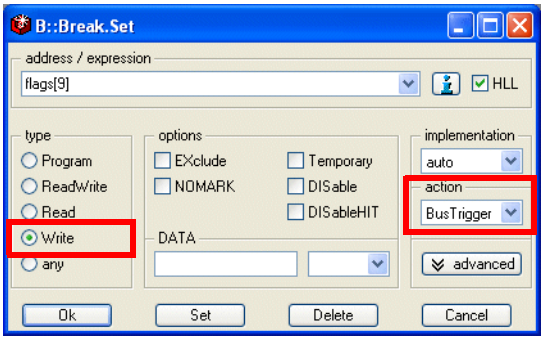
Resource: Watchpoints and logic in NEXUS Adapter (parallel trace only)

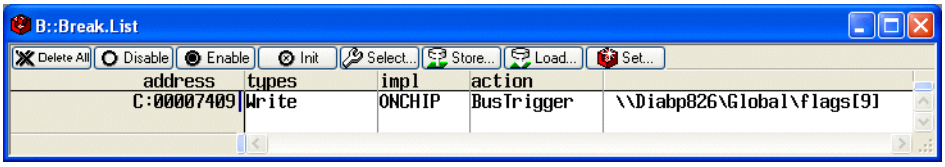
Controlled message types

WTM	BTM	DTM	OTM	DQM
Watchpoint Hit Message(s) is generated for the specified instruction(s) or data address+data value	Unaffected	Unaffected	Unaffected	Unaffected

Example: Generate a 100 ns high pulse on the trigger connector of POWERTRACE/ETHERNET or POWER DEBUG II when a write access to flags[9] occurs.

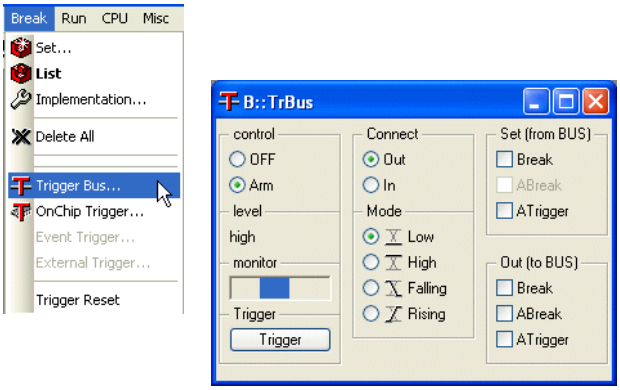
1. Set a write breakpoint to the variable flags[9] and select the action BusTrigger.





2. Start the program execution.

3. Open the ***TrBus*** window to watch the trigger.



Example for BusCount (Watchpoint)

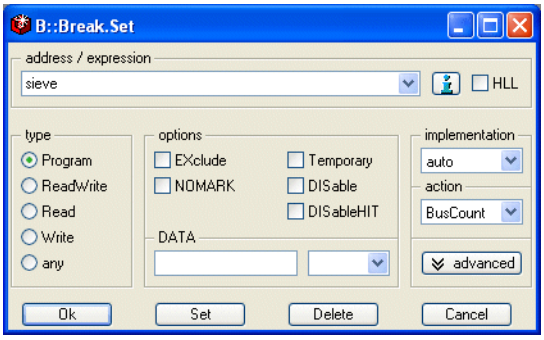
Resource: Watchpoints and logic in NEXUS Adapter (parallel trace only). Only one event possible.

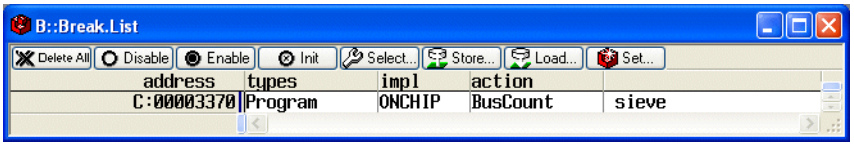
Controlled message types

WTM	BTM	DTM	OTM	DQM
Watchpoint Hit Message(s) is generated for the specified instruction(s) or data address+data value	Unaffected	Unaffected	Unaffected	Unaffected

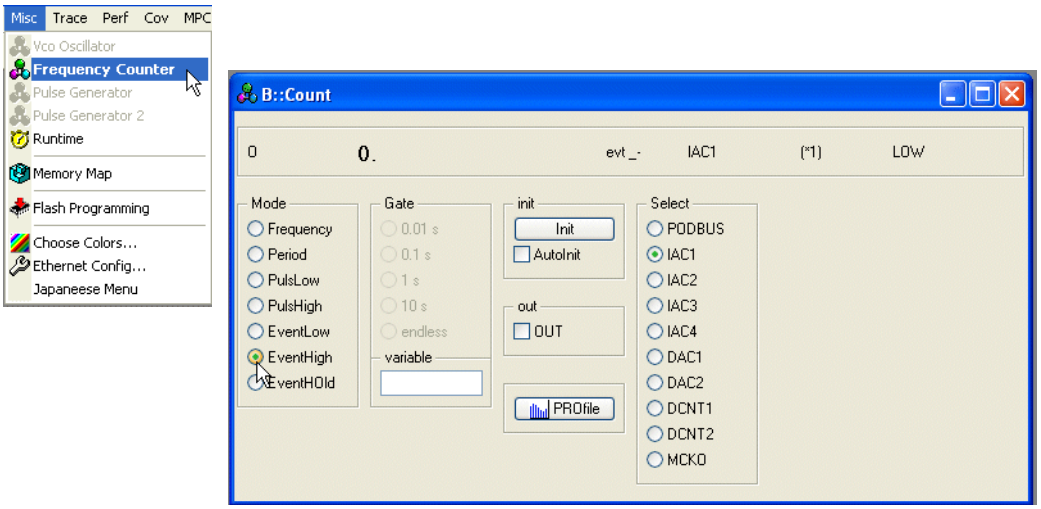
Example 1: Count how often the function sieve is called.

1. Set a Program breakpoint to the start address of the function sieve and select the action BusCount.

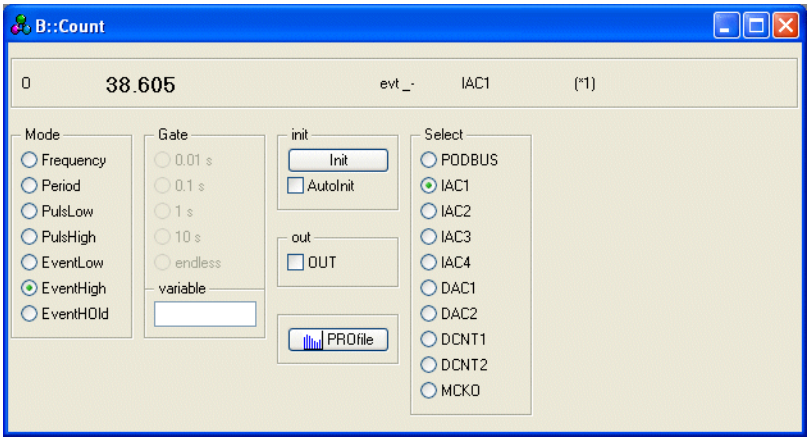




2. Open the TRACE32 counter window and select EventHigh.

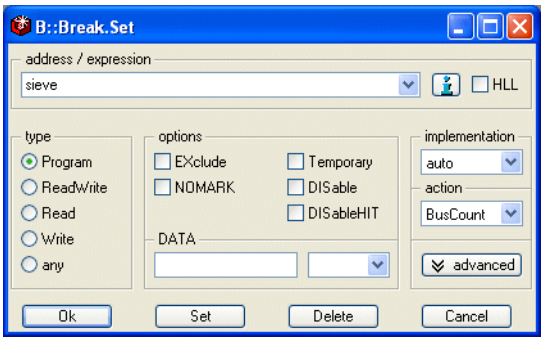


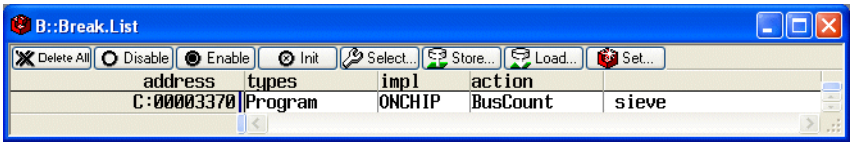
3. Start the program execution and display the result.



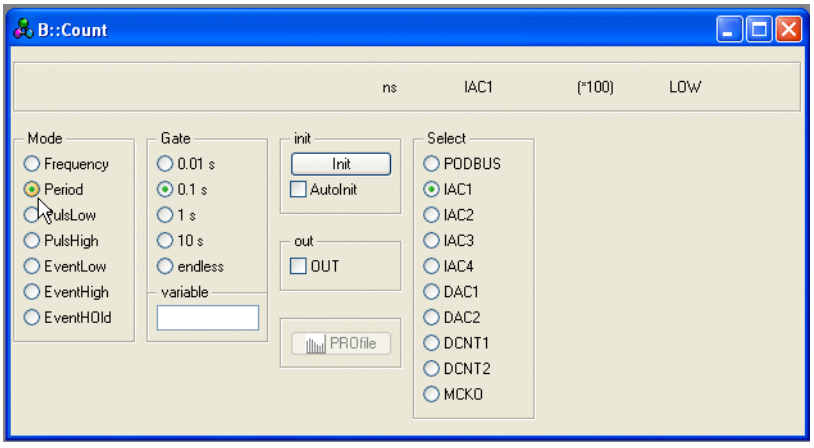
Example 2: Measure the averaged time distance in which the function sieve is called.

1. Set a Program breakpoint to the start address of the function sieve and select the action BusCount.

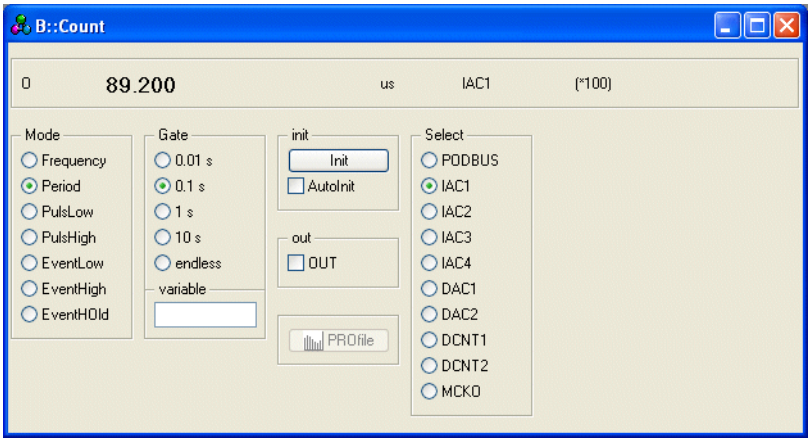




2. Open the TRACE32 counter window and select Period.



3. Start the program execution and display the result.



Filter and Trigger (Core) - SMP Debugging

Filters and Triggers are programmed to all cores that are controlled by the TRACE32 instance.

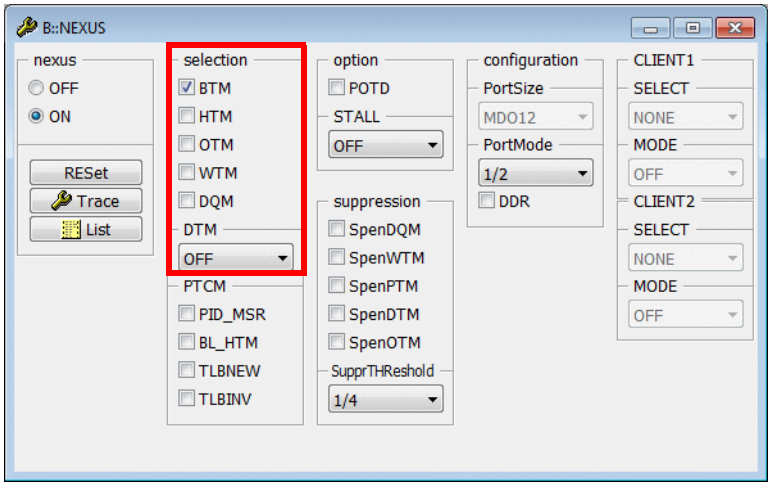
The fact that TRACE32 does not know on which core of the SMP system a program section is running has the consequence that the same filters/triggers are programmed to all cores. So, from the perspective of TRACE32, you can say the resources for filters/triggers are shared by all cores.

Resource: Watchpoints

Controlled message types

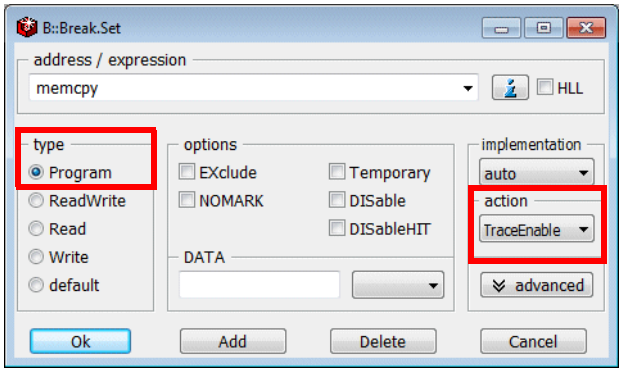
WTM Watchpoint Trace Messages	**BTM** Branch Trace Messages	**DTM** Data Trace Messages	**OTM** Ownership Trace Messages	**DQM** Data Acquisition Messages
Watchpoint Hit Message(s) is generated for the specified instruction(s)	Disabled	Unaffected	Unaffected	Unaffected

Disable message types, that are unaffected by the filter and not required for your analysis.



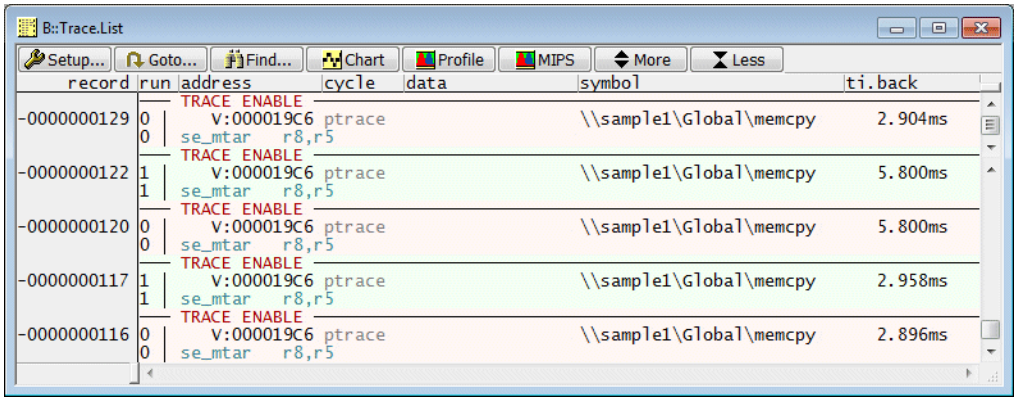
Example 1: Advise the NEXUS module to generate only trace information for the entries to the function memcpy.

1. Set a Program breakpoint to the start address of the function memcpy and select the action TraceEnable.



2. Start the program execution and stop it.

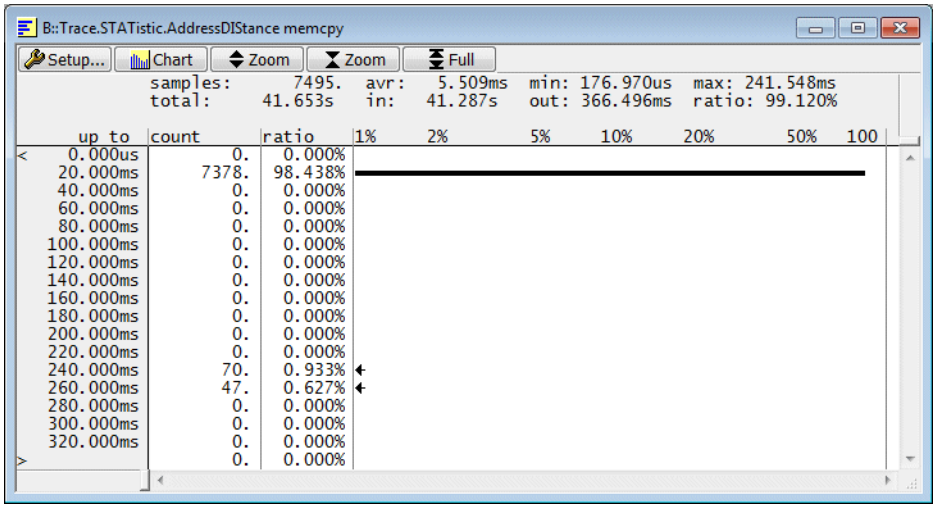
3. Display the result.




```
Break.Delete /ALL                          ; delete all breakpoints

Break.Set memcpy /Program /TraceEnable     ; program filter

Go                                         ; start program execution

…

Break                                      ; stop program execution

Trace.List                                 ; display result
```

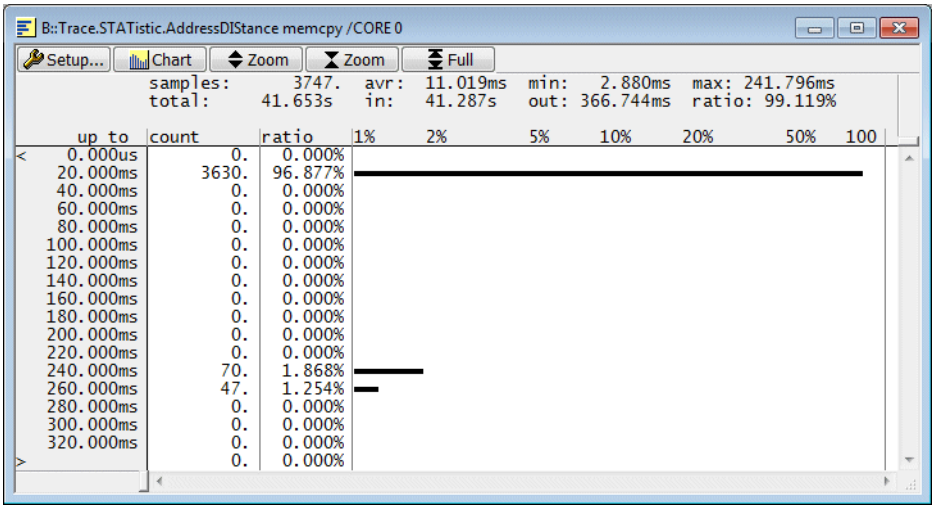
The following **Trace.STATistic** command calculates the time intervals for a program address event. The program address event is here the entry to the function memcpy. The core information is discarded for this calculation.

```
Trace.STATistic.AddressDIStance memcpy
```



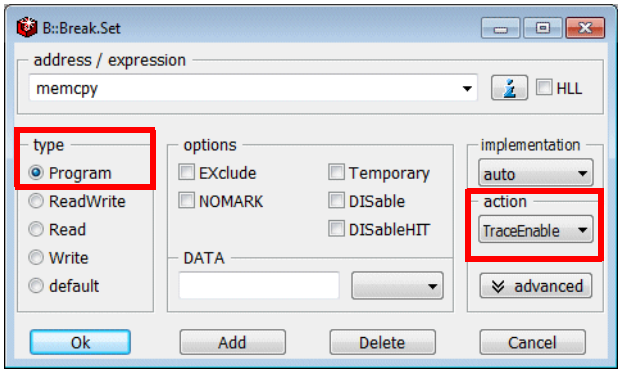
If you need the result per core, use the following command:

```
Trace.STATistic.AddressDIStance memcpy /CORE 0
```

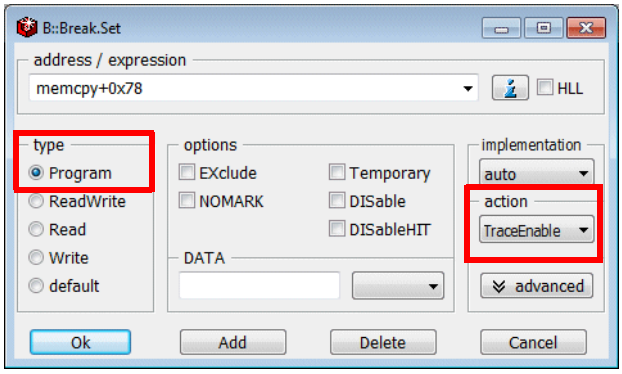


Example 2: Advise the NEXUS module to generate trace information for the entries to the function memcpy and for the exits of the function memcpy.

1. Set a Program breakpoint to the start address of the function memcpy and select the action TraceEnable.



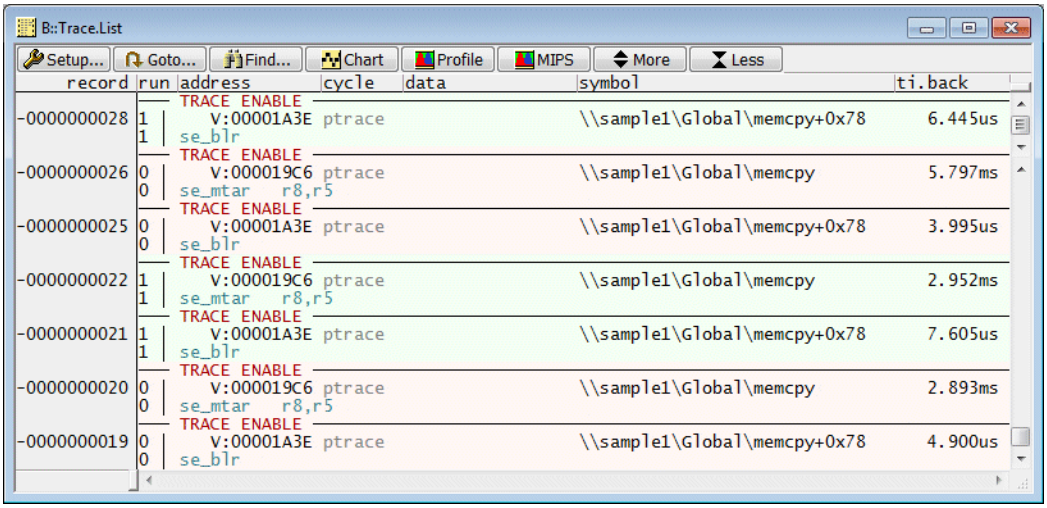
2. Set a Program breakpoint to the exit address of the function memcpy and select the action TraceEnable.



sYmbol.EXIT(<*symbol*>**)**	Returns the exit address of the specified function

3. Start the program execution and stop it.

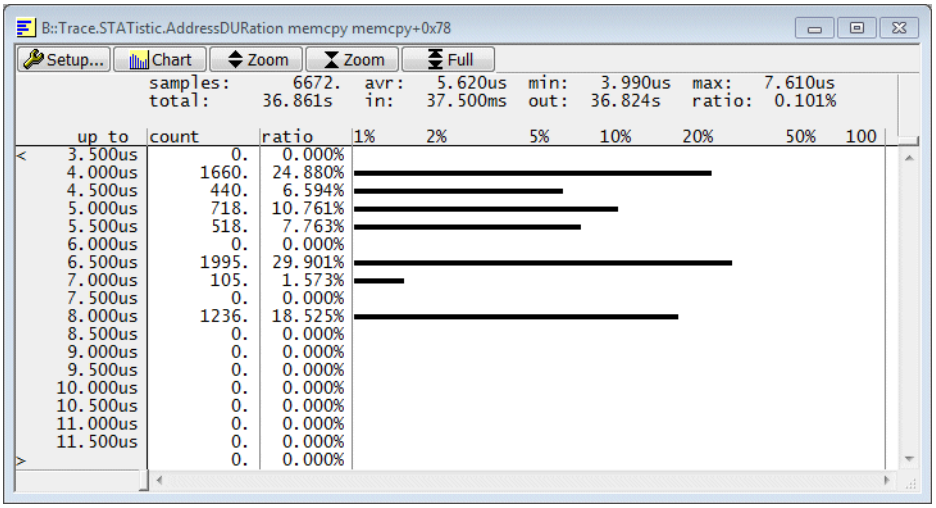
4. Display the result.



record	run	address	cycle	data	symbol	ti.back
		TRACE ENABLE				
-0000000028	1	V:00001A3E	ptrace		\\sample1\Global\memcpy+0x78	6.445us
	1	se_blr				
		TRACE ENABLE				
-0000000026	0	V:000019C6	ptrace		\\sample1\Global\memcpy	5.797ms
	0	se_mtar r8,r5				
		TRACE ENABLE				
-0000000025	0	V:00001A3E	ptrace		\\sample1\Global\memcpy+0x78	3.995us
	0	se_blr				
		TRACE ENABLE				
-0000000022	1	V:000019C6	ptrace		\\sample1\Global\memcpy	2.952ms
	1	se_mtar r8,r5				
		TRACE ENABLE				
-0000000021	1	V:00001A3E	ptrace		\\sample1\Global\memcpy+0x78	7.605us
	1	se_blr				
		TRACE ENABLE				
-0000000020	0	V:000019C6	ptrace		\\sample1\Global\memcpy	2.893ms
	0	se_mtar r8,r5				
		TRACE ENABLE				
-0000000019	0	V:00001A3E	ptrace		\\sample1\Global\memcpy+0x78	4.900us
	0	se_blr				

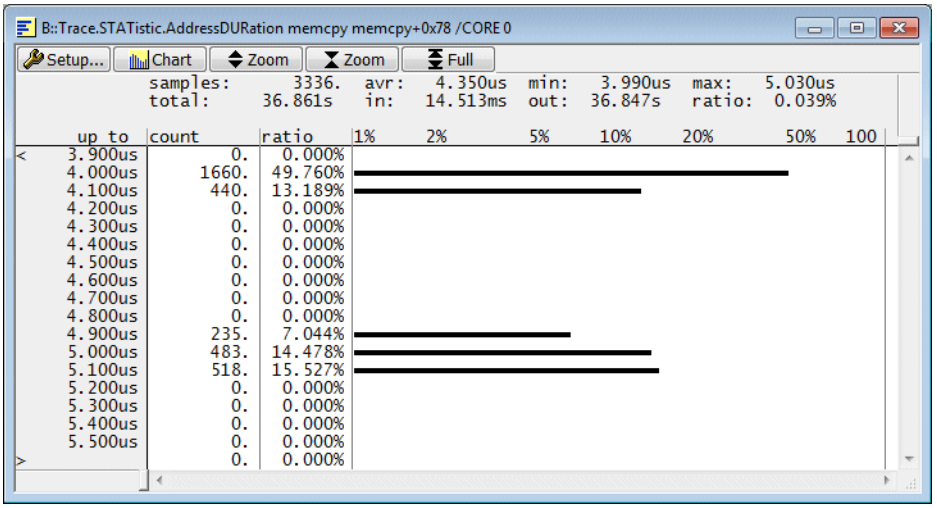
The following **Trace.STATistic** command calculates the time intervals between two program address events A and B. The entry to the function memcpy is A in this example, the exit from the function is B. The core information is discarded for this calculation.

```
Trace.STATistic.AddressDURation memcpy memcpy+0x78
```



If you need the result per core, use the following command:

```
Trace.STATistic.AddressDURation memcpy memcpy+0x78 /CORE 0
```



Resource: Limited to one instruction address range

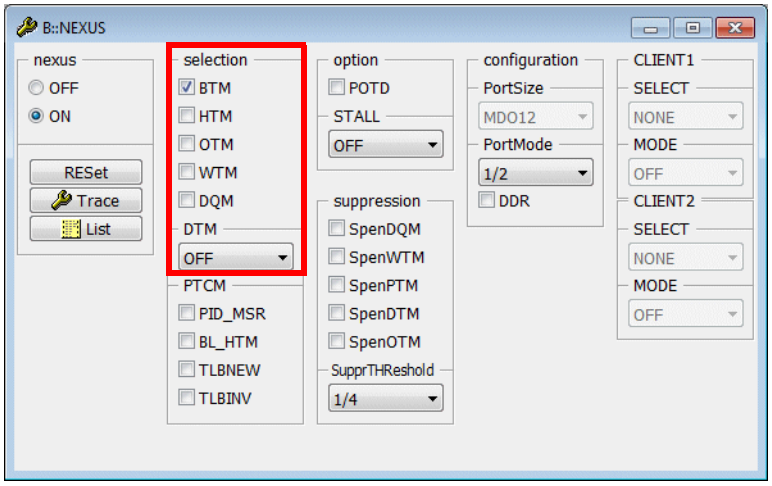
Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	Filter applies if BTM is enabled	Filter applies if DTM is enabled	Unaffected	Unaffected

Enable BTM. This filter requires that Branch History messaging is disabled.

Enable DTM if you are interested in the read/write accesses performed by the specified instruction address range.

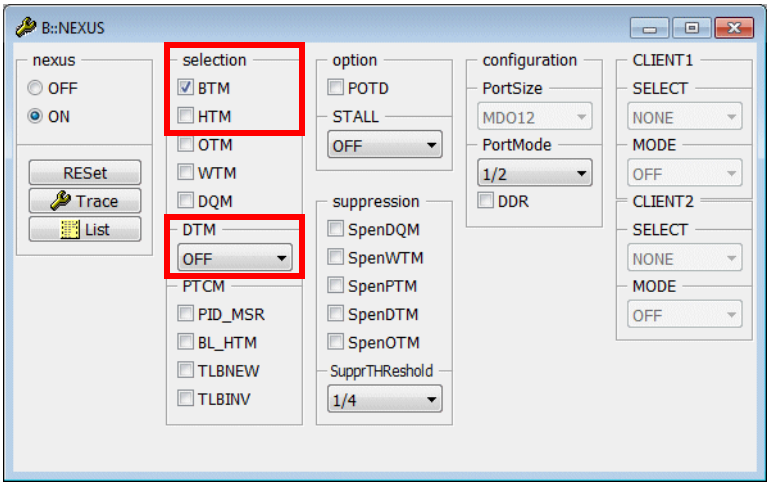
Disable message types, that are unaffected by the filter and not required for your analysis.



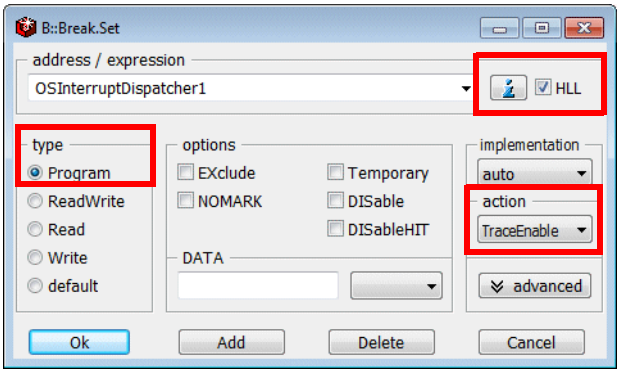
Example: Advise the NEXUS module to generate trace information for all taken branches within the function OSInterruptDispatcher1.

1. Enable Branch Trace messaging, but don't enable Indirect Branch History messaging.

   Disable Data Trace messaging.

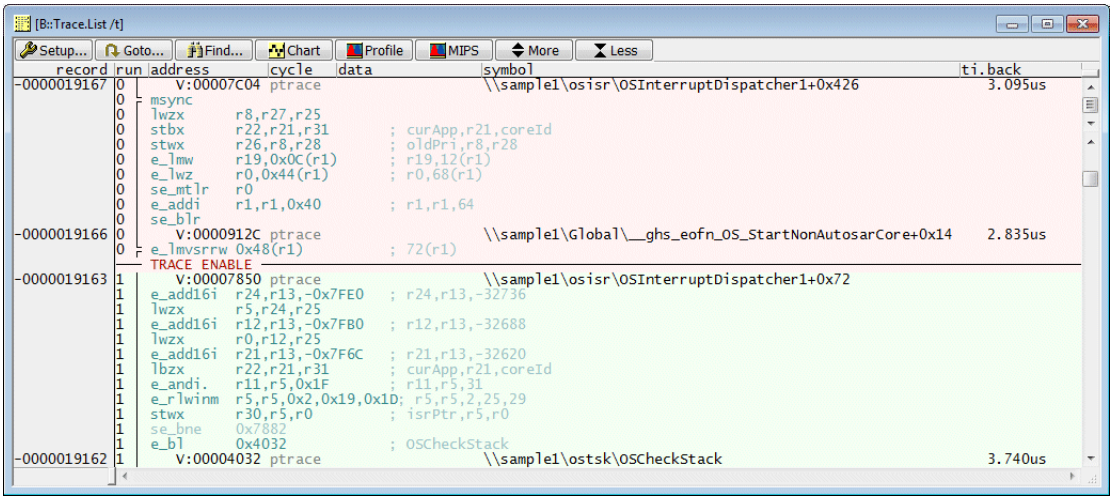


2. Set a Program breakpoint to the complete address range of the function OSInterruptDispatcher1 (HLL check box ON) and select the action TraceEnable.



3. Start the program execution and stop it.

4. Display the result.

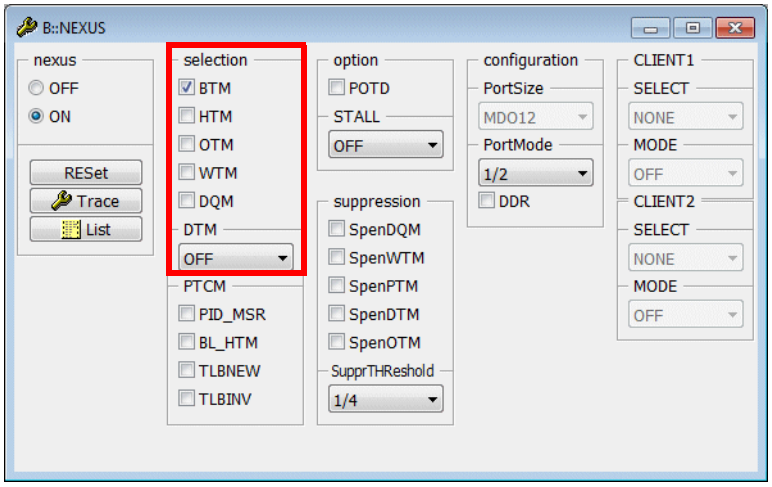


Resource: DTC Register

Controlled message types

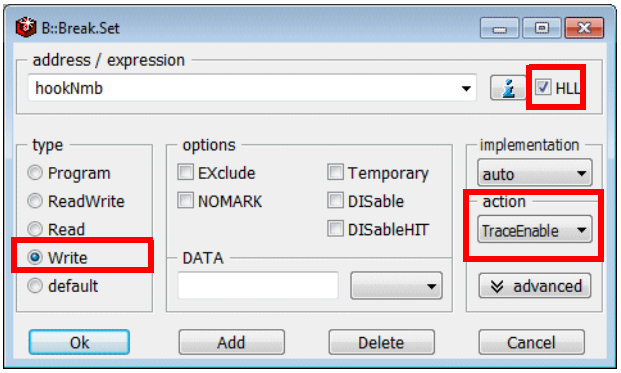
WTM	BTM	DTM	OTM	DQM
Unused	BTM is disabled by filter	DTM is enabled by filter Filter applies	Unaffected	Unaffected

Disable message types, that are unaffected by the filter and not required for your analysis.



Example: Disable Branch Trace Messaging and advise the NEXUS module to generate trace information for the write accesses to the variable hookNmb.

1. Set a Write breakpoint to the variable hookNmb and select the action TraceEnable

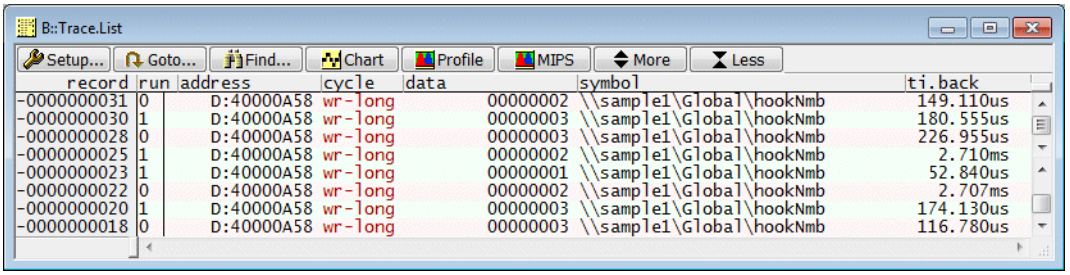


- no data value possible (limitation of DTC Register)

- accessing instruction not possible (limitation of DTC Register)

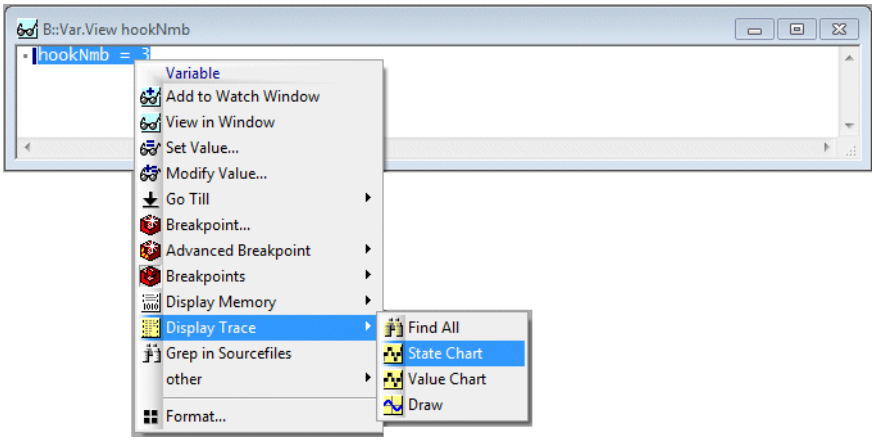
2. Start the program execution and stop it.

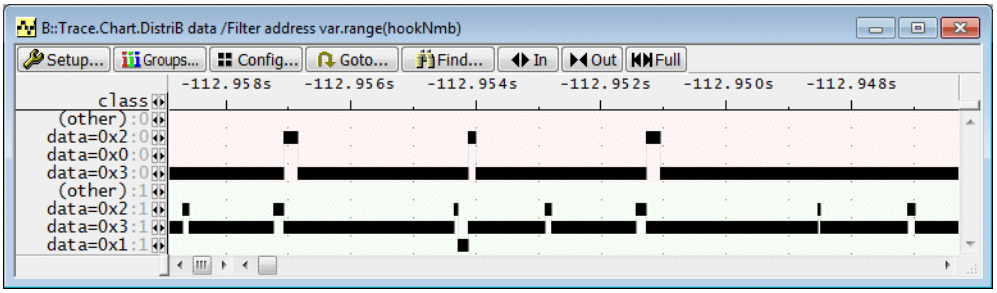
3. Display the result.



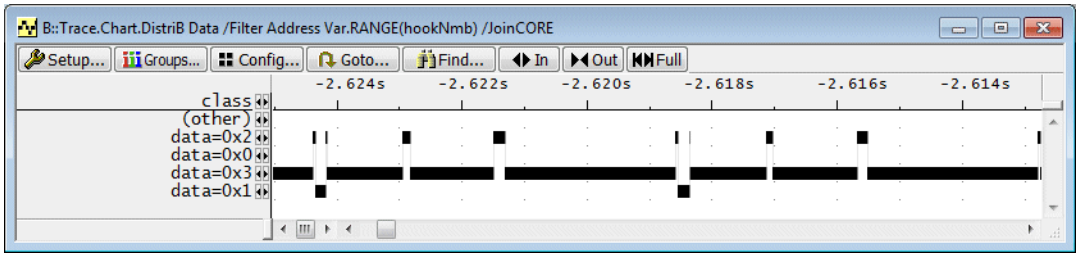
The Variable pull-down provides various ways to analyze the variable contents over the time.

```
; open a window to display the variable
Var.View hookNmb
```

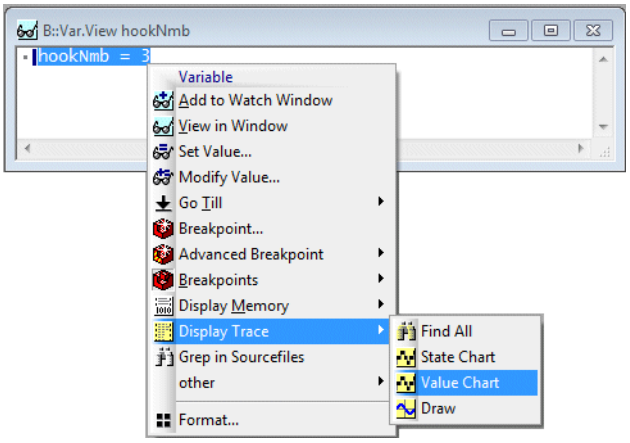


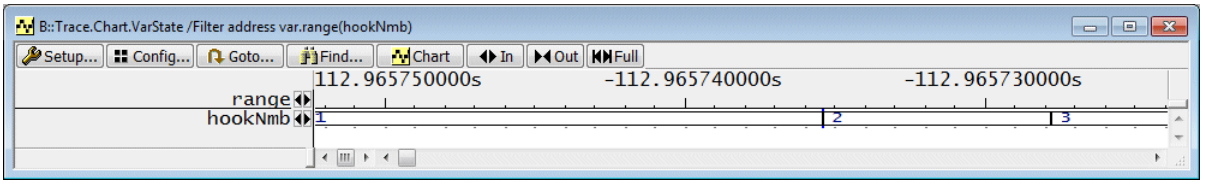


Display the value changes of a variable graphically - value changes per core
Trace.Chart.DistriB Data /Filter Address Var.RANGE(*<var>*) [/SplitCORE]

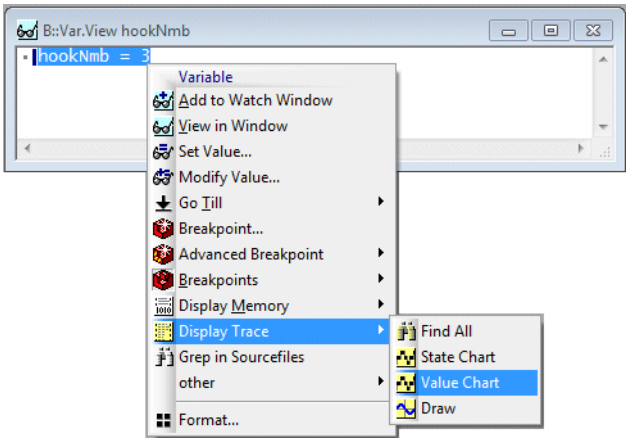


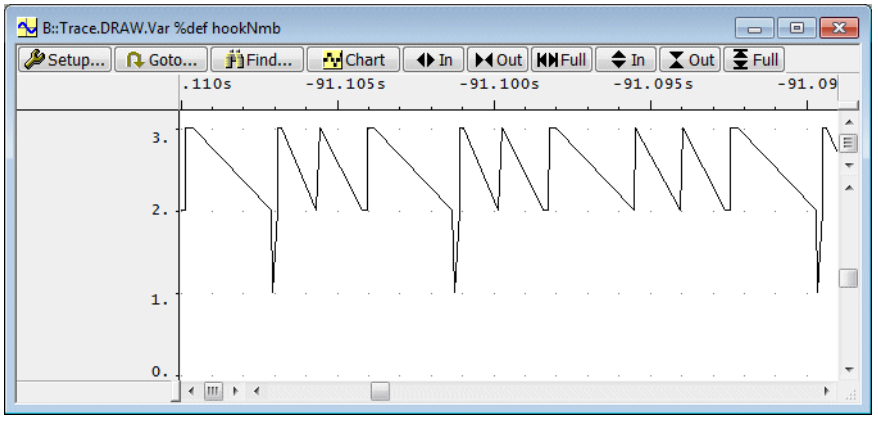
Display the value changes of a variable graphically - value changes of all cores
Trace.Chart.DistriB Data /Filter Address Var.RANGE(*<var>*) /JoinCORE





Display variable contents over the time (numerically) - the core information is discarded
Trace.Chart.VarState





Display variable contents over the time (graphically) - the core information is discarded
Trace.DRAW.Var %DEFault *<var>*

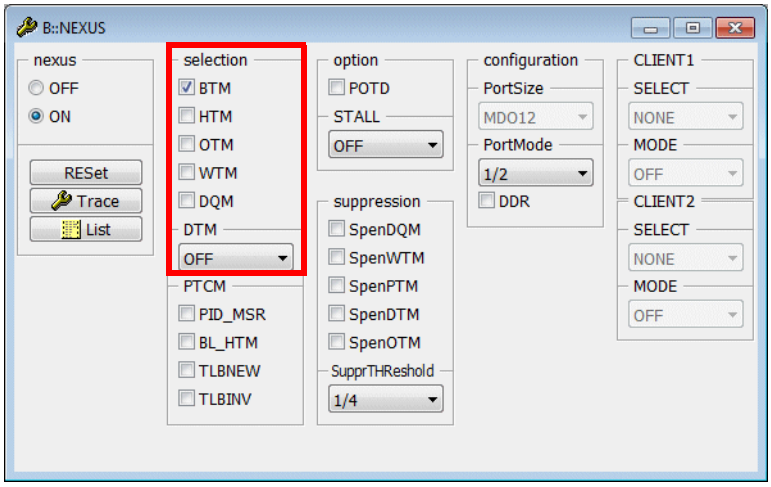
Example for TraceData

Resource: DTC Register

Controlled message types

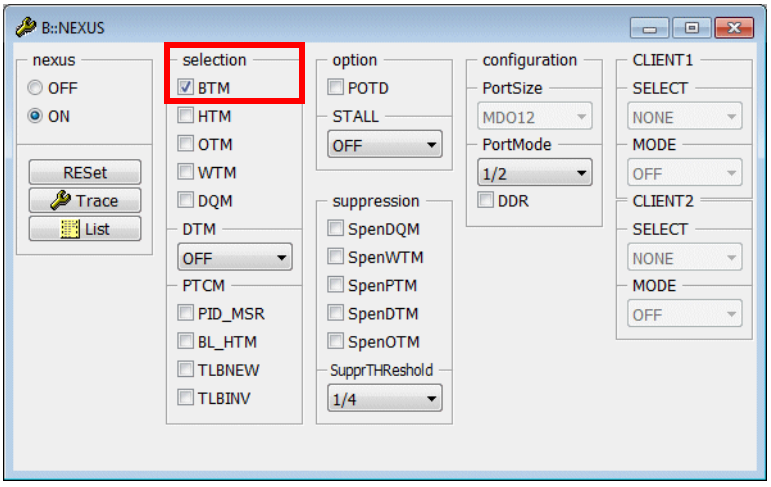
WTM	BTM	DTM	OTM	DQM
Unused	Unaffected	DTM is enabled by filter Filter applies	Unaffected	Unaffected

Disable message types that are unaffected by the filter and not required for the analysis.

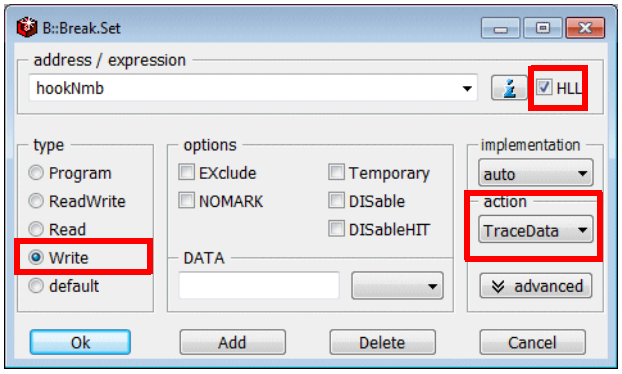


Example: Advise the NEXUS module to generate Data Trace Messages for all write accesses to the variable hookNmb and to generate trace information for all executed instructions.

1. Enable Branch Trace messaging.

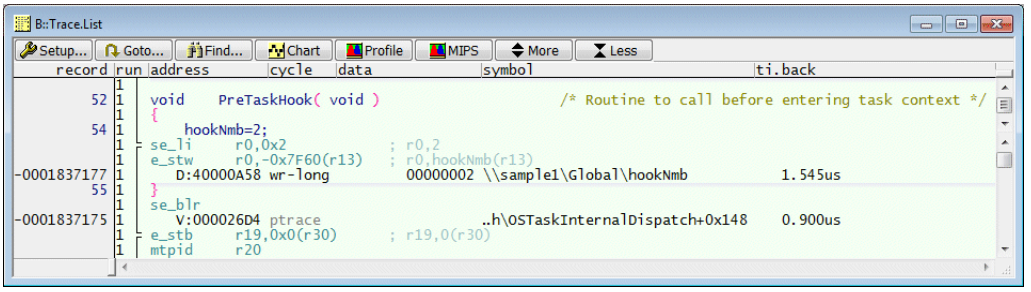


2. Set a Write breakpoint to the variable hookNmb and select the action TraceData.



3. Start the program execution and stop it.

4. Display the result.



Please be aware that in the case of a TraceData filter a correlation of the data access and the instruction is in most cases not possible.

Resource: Watchpoints

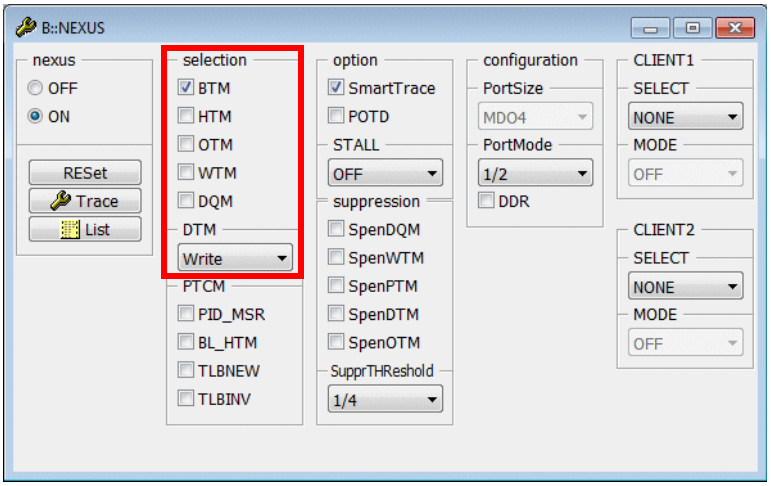
Global TraceON/Trace OFF

Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	Filter applies	Filter applies	Unaffected	Unaffected

Enable Branch Trace Messaging and Data Trace Messaging if this information is required for your analysis.

Diable messages types that are unaffected and not required for the analysis.

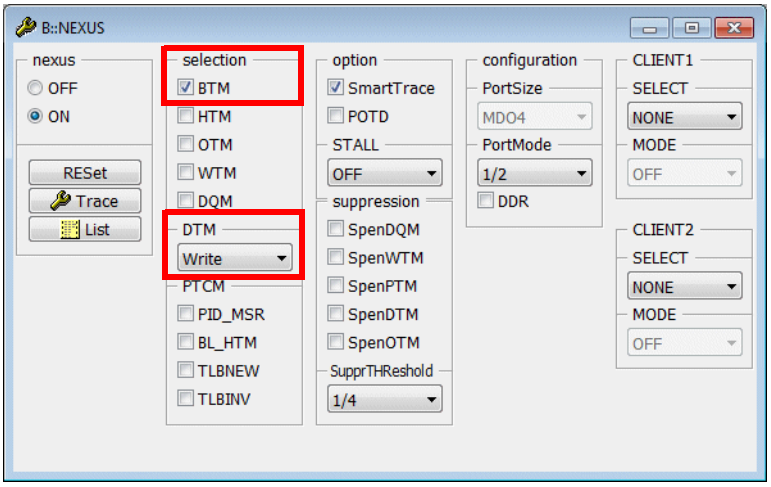


Example:

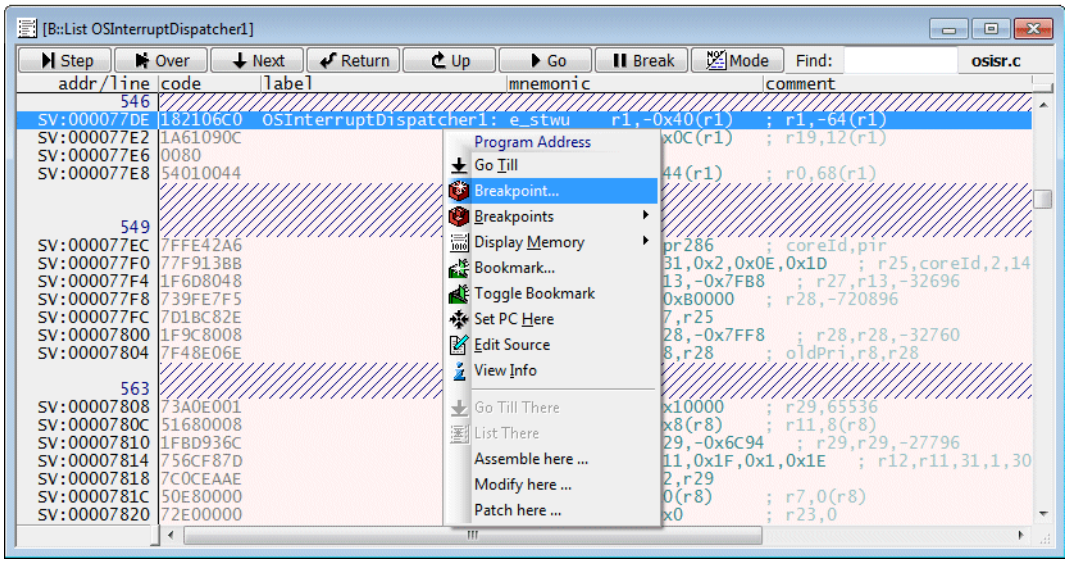
Advise the NEXUS module to start Branch Trace messaging and Data Write Messages at the entry to the function OSInterruptDispatcher1.

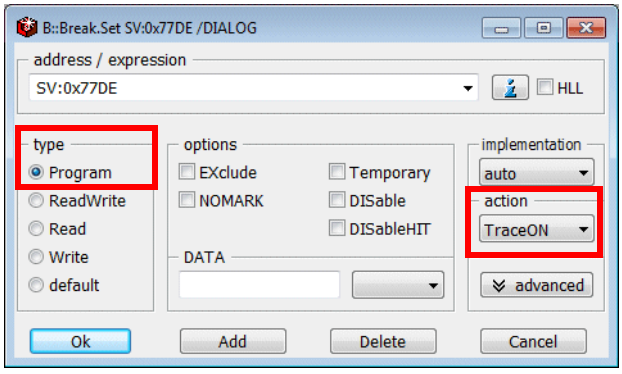
Advise the NEXUS module to stop Branch Trace messaging and Data Write Messages at the exit of the function OSInterruptDispatcher1.

1. Enable Branch Trace Messages and Data Write Messages.

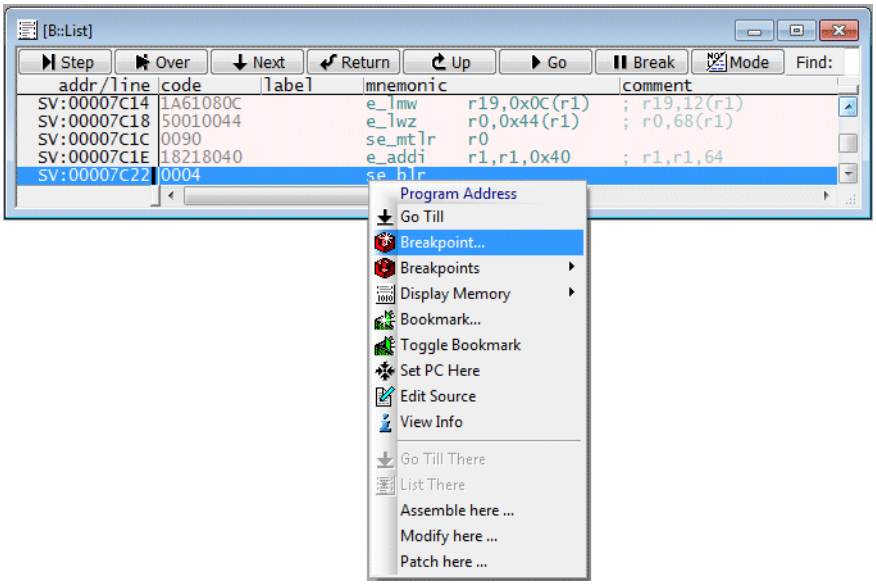


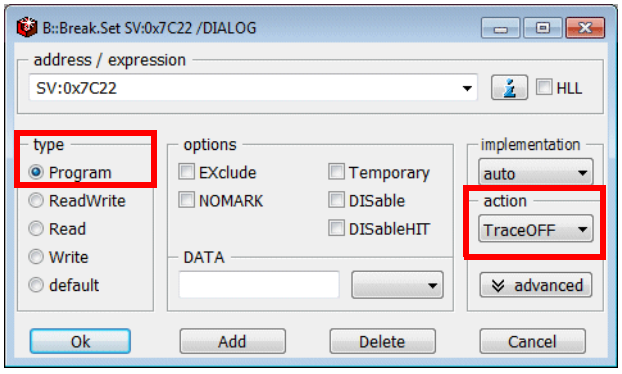
2. Set a Program breakpoint to the entry of the function OSInterruptDispatcher1 and select the action TraceON.

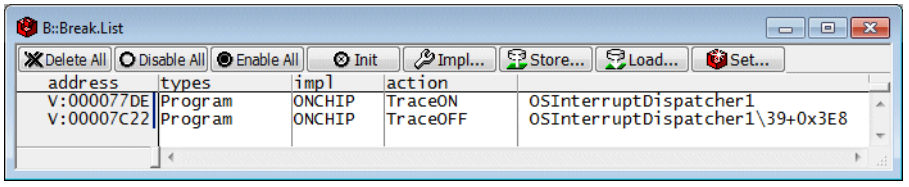




3. Set a Program breakpoint to the exit of the function OSInterruptDispatcher1 and select the action TraceOFF.

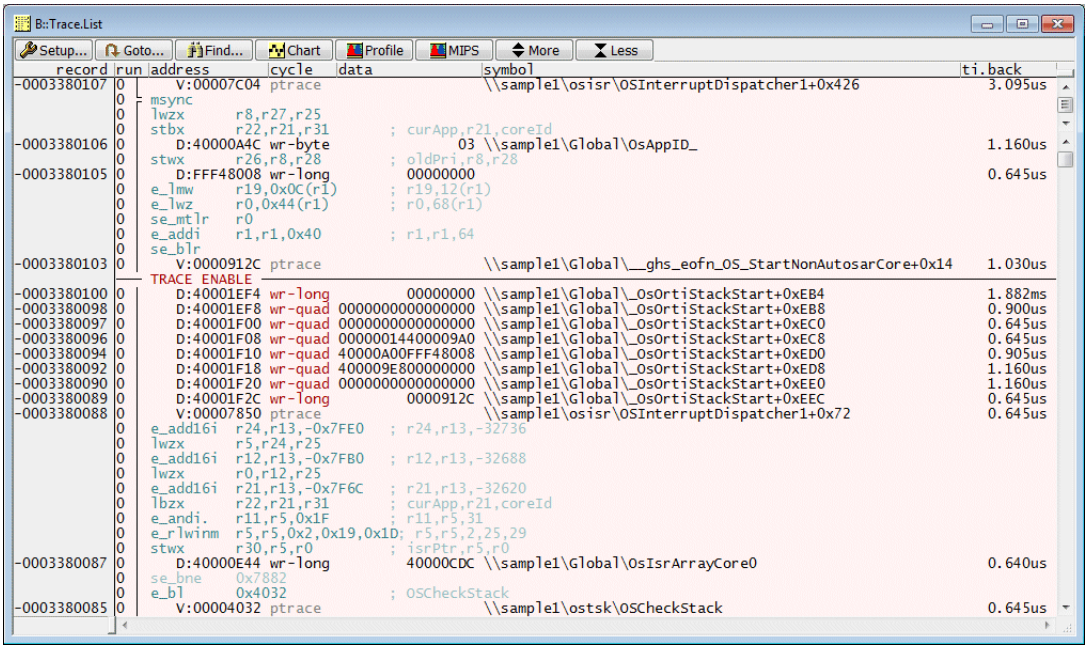






4. Start the program execution and stop it.

5. Display the result.

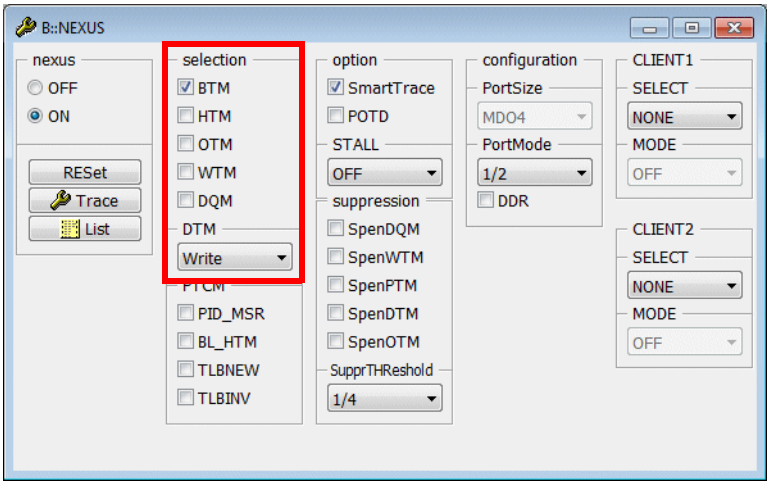


The event that switched the trace generation on is not visible in the trace.

Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	BTM is enabled by filter Filter applies	Unaffected	Unaffected	Unaffected

Diable messages types that are unaffected and not required for the analysis.



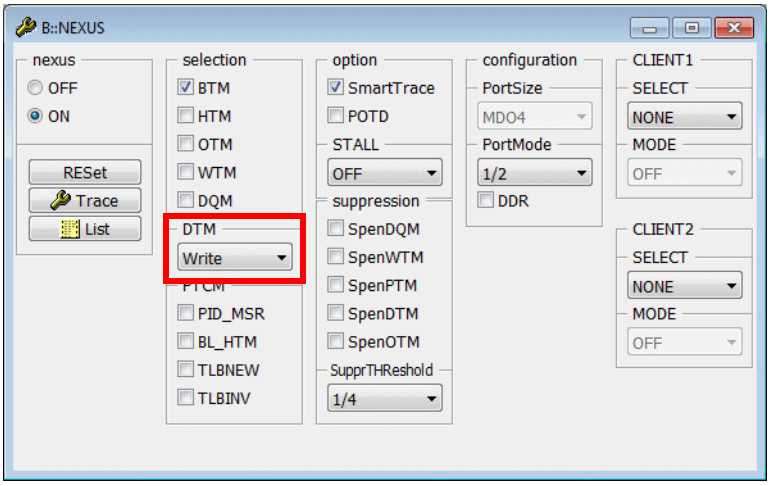
Example:

Advise the NEXUS module to generate trace information on all write accesses.

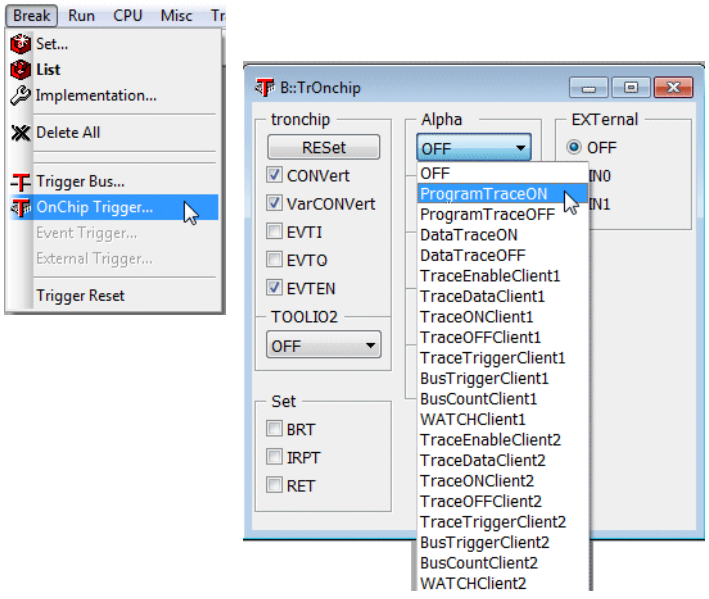
Advise the NEXUS module to start the Branch Trace messaging at the entry to the function memcpy.

Advise the NEXUS module to stop Branch Trace messaging at the exit of the function memcpy.

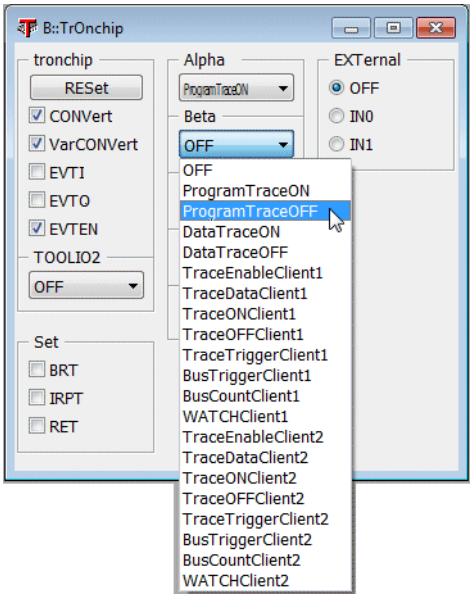
1. Enable Data Trace messaging for write accesses.



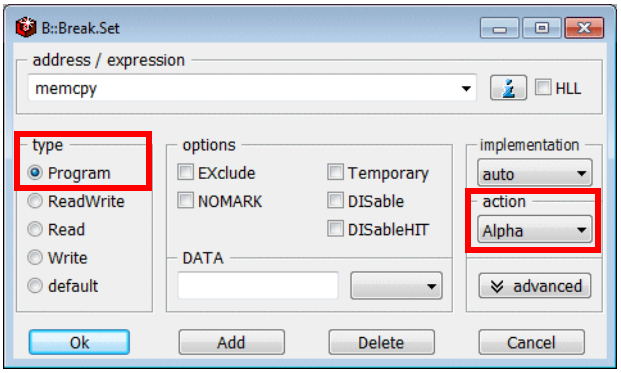
2. Open the TrOnchip window and select ProgramTraceON for Alpha.



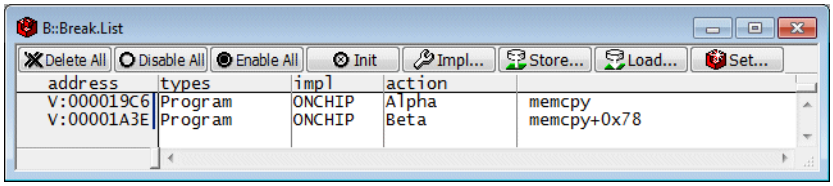
3. Select ProgramTraceOFF for Beta.



4. Set a Program breakpoint to the entry of the function memcpy and select the action Alpha.

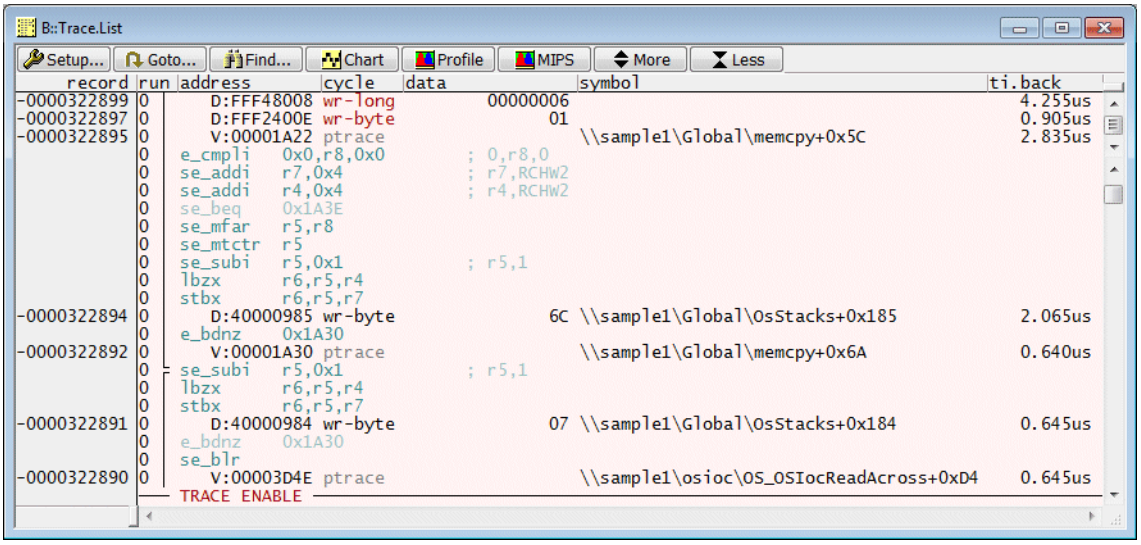


5. Set a Program breakpoint to the exit of the function at memcpy and select the action Beta.



6. Start and stop the program execution.

7. Display the result.



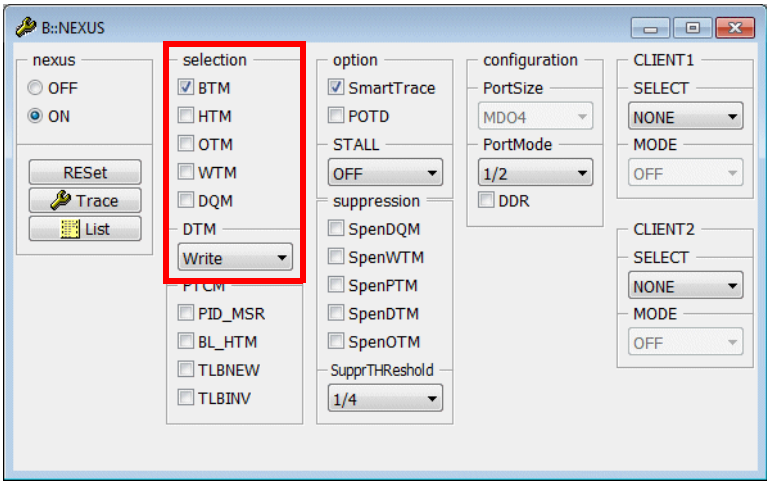
```
; default start situation
Break.Delete /ALL
TrOnchip.RESet

; messaging setup
NEXUS.BTM ON
NEXUS.DTM Write

; filter settings
TrOnchip.Alpha ProgramTraceON
TrOnchip.Beta ProgramTraceOFF
Break.Set memcpy /Program /Alpha
Break.Set memcpy+0x78 /Program /Beta

Go

…

Break

; display result
Trace.List
```

Controlled message types

WTM	BTM	DTM	OTM	DQM
Unused	Unaffected	Filter applies	Unaffected	Unaffected

Enable the Data Trace Messaging as required for the analysis.

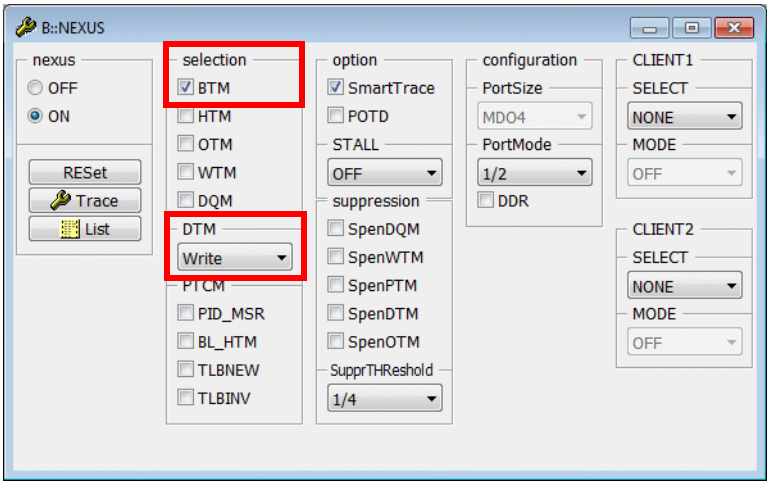
Disable messages types that are unaffected and not required for the analysis.



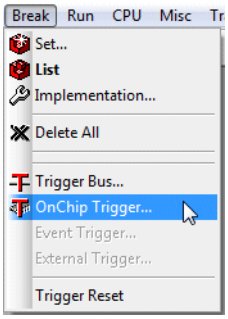
Example:

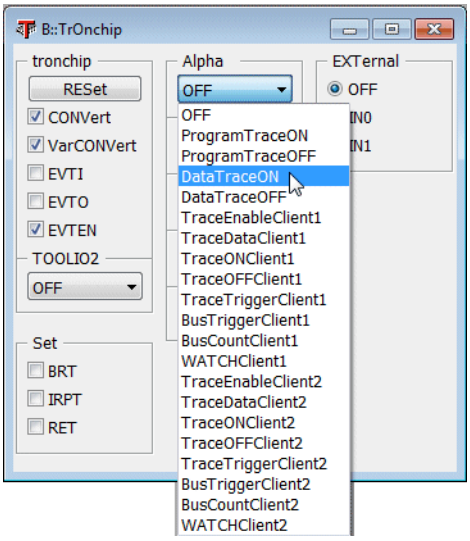
Enable Branch Trace messaging. Advise the NEXUS module to start the generation of Data Write Messages at the entry to the function OSInterruptDispatcher1. Advise the NEXUS module to stop the generation of Data Write Messages at the exit of the function OSInterruptDispatcher1.

1. Enable Branch Trace messaging and Data Trace messaging for write accesses.

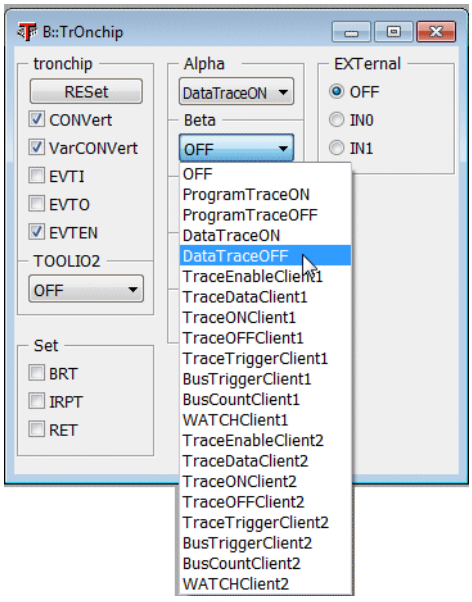


2. Open the TrOnchip window and select DataTraceON for Alpha.

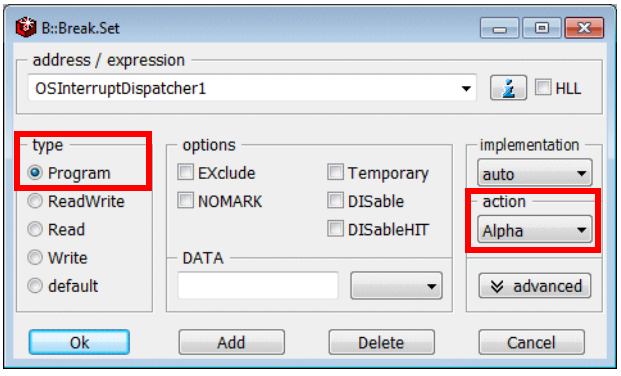




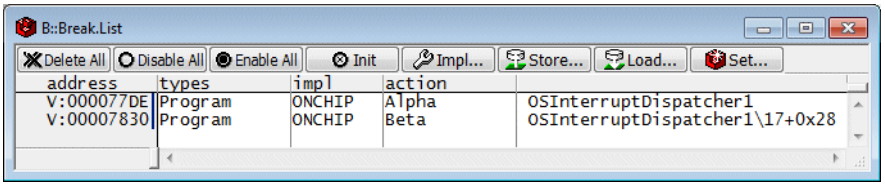
3. Select DataTraceOFF for Beta.



4. Set a Program breakpoint to the entry of the function OSInterruptDispatcher1 and select the action Alpha.

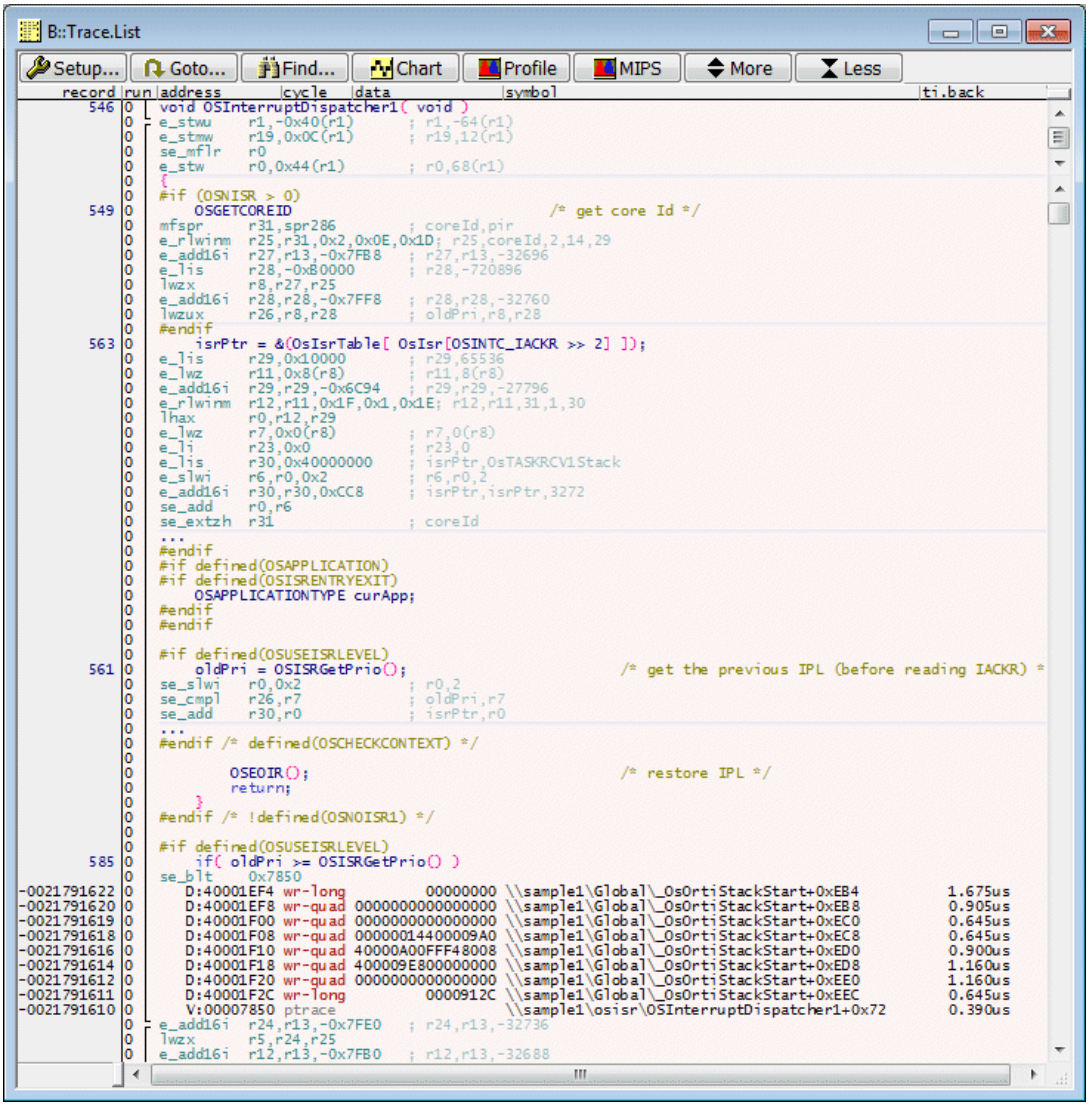


5. Set a Program breakpoint to the exit of the function OSInterruptDispatcher1and select the action Beta.



6. Start and stop the program execution.

7. Display the result.



Example for TraceTrigger

Resource: Watchpoints and logic in NEXUS Adapter (parallel trace only)

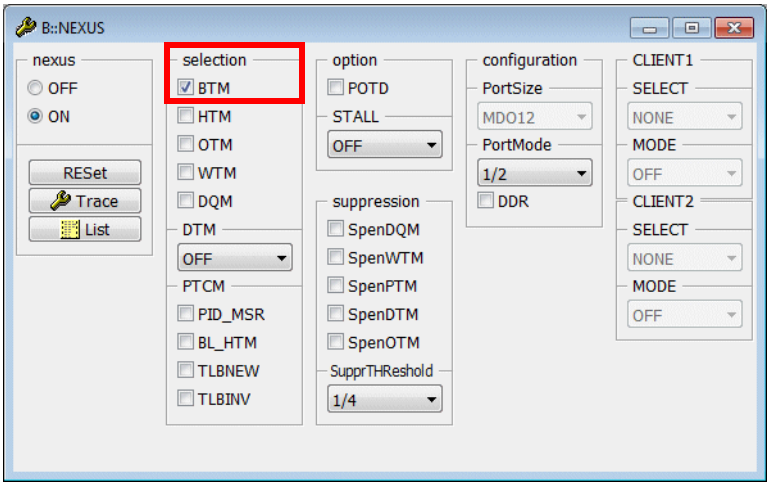
Controlled message types

WTM	BTM	DTM	OTM	DQM
Watchpoint Hit Message(s) is generated for the specified instruction(s) or data address+data value	Unaffected	Unaffected	Unaffected	Unaffected

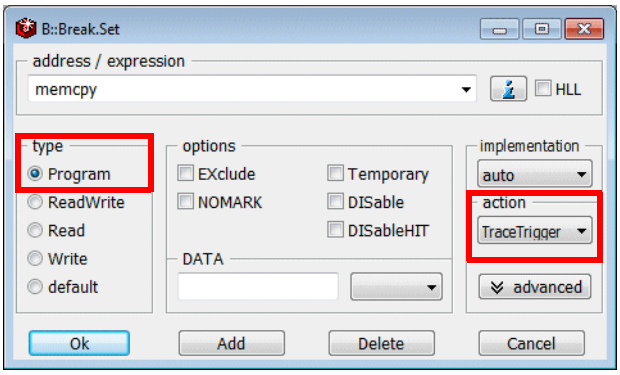
Disable messages types that are unaffected and not required for the analysis.

Example: Enable Branch Trace messaging. Advise the NEXUS module to generate a trigger for the trace if the function memcpy is entered. Use this trigger to stops the trace recording.

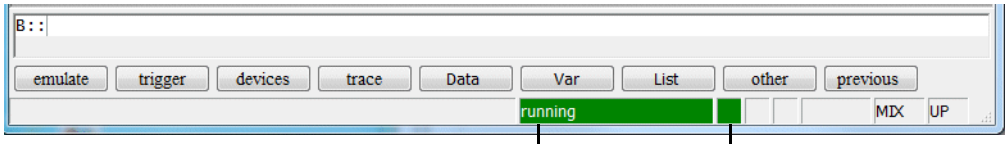
1. Enable Branch Trace messaging.



2. Set a Program breakpoint to the start address of the function memcpy and select the action TraceTrigger.

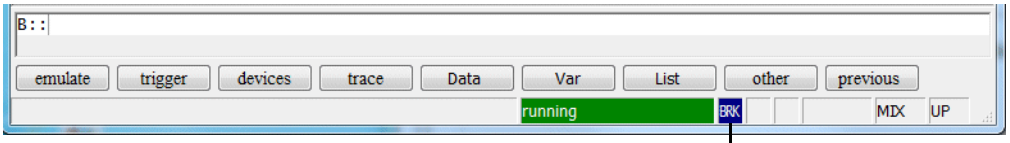


3. Start the program execution.



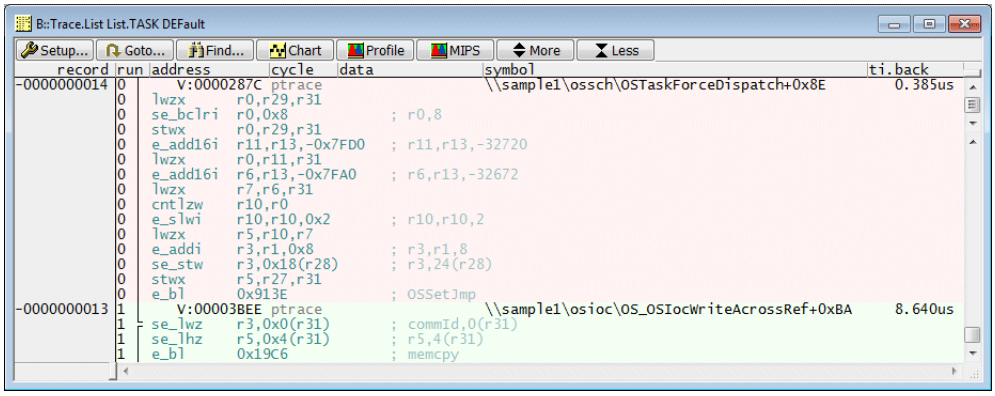
State of the program execution (running)

State of the trace recording (Arm = recording)



State of the trace recording (BRK = break by trigger, recording is stopped)

4. Display the result.



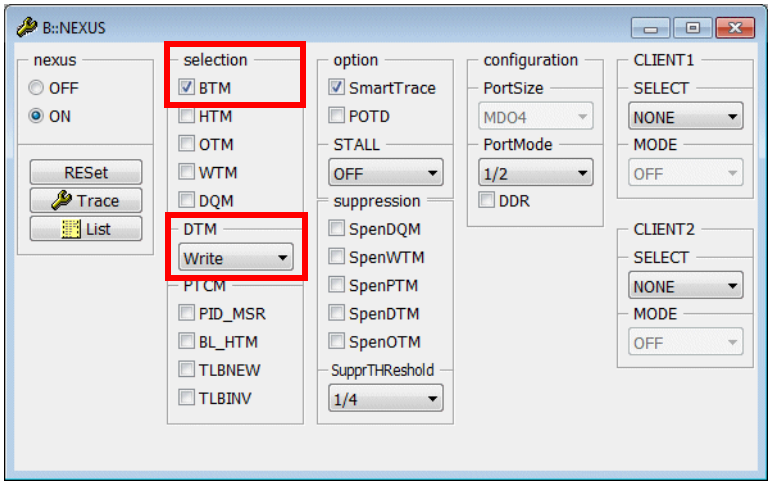
The trace generation is usually stopped before the trace information for the event that caused the trigger is exported.

Example for TraceTrigger with a Trigger Delay

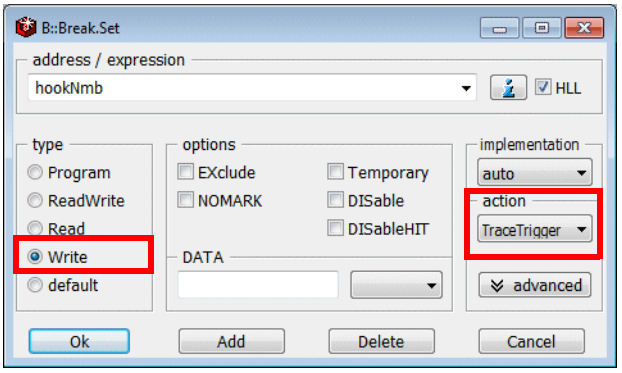
Example:

Advise the NEXUS module to generate a trigger if a write access to the variable hookNmb occurs. Advise TRACE32 to fill another 10% of the trace memory before the trace recording is stopped.

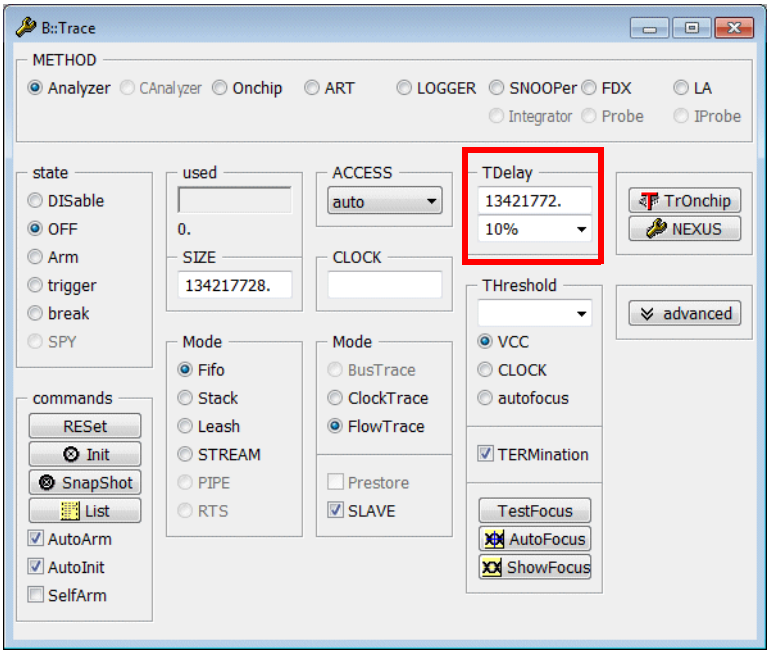
1. Enable Branch Trace messaging and Data Trace messaging for write accesses.



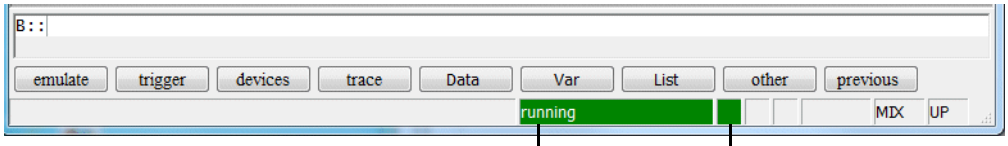
2. Set a Write breakpoint to the variable hookNmb and select the action TraceTrigger.



3. Define the trigger delay in the ***Trace Configuration*** Window.

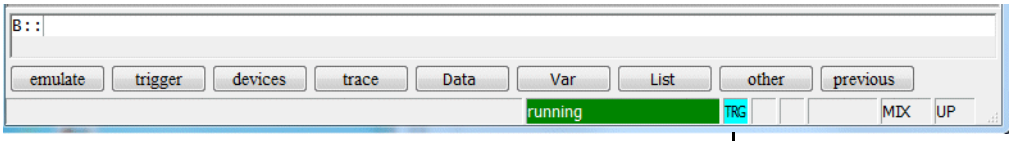


4. Start the program execution.

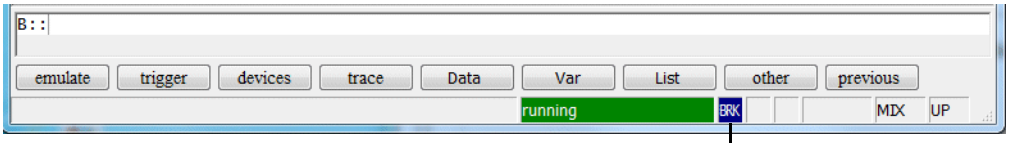


State of the
program execution
(running)

State of the
trace recording
(Arm = recording)

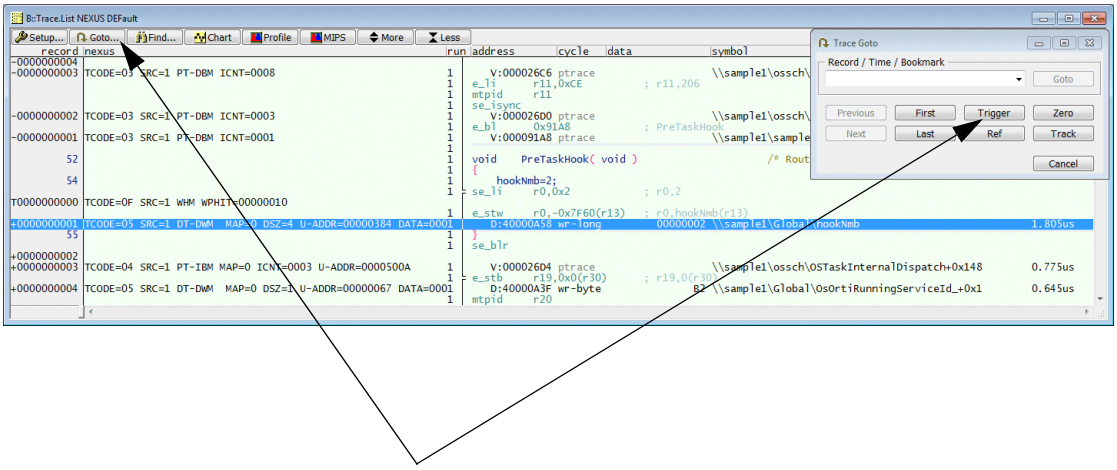


State of the
trace recording
(TRG = trigger occurred,
delay counter started)



State of the
trace recording
(BRK = delay counter elapsed,
recording is stopped)

5. Display the result.



Push the ***Trigger*** button in the ***Trace Goto*** window to find the record, where the trigger occurred (WHM message). Here the sign of the record numbers changed.
The specified event is usually exported shortly after this point.

Example for BusTrigger

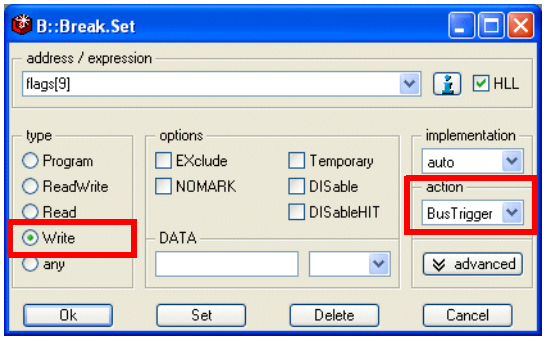
Resource: Watchpoints and logic in NEXUS Adapter (parallel trace only)

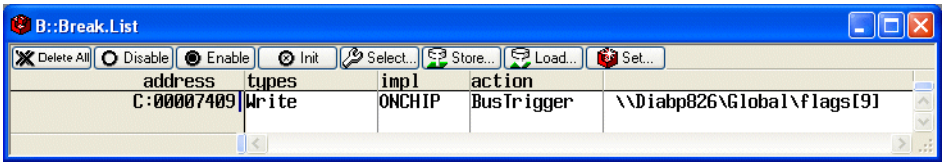
Controlled message types

WTM	BTM	DTM	OTM	DQM
Watchpoint Hit Message(s) is generated for the specified instruction(s) or data address+data value	Unaffected	Unaffected	Unaffected	Unaffected

Example: Generate a 100 ns high pulse on the trigger connector of the POWER TRACE / ETHERNET or POWER DEBUG II when 3 is writes to hookNmb.

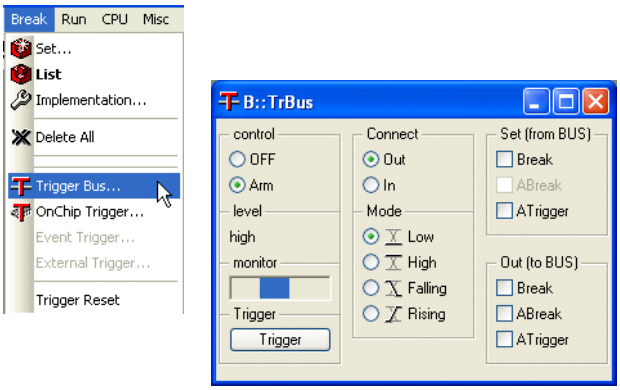
1. Set a write breakpoint to the variable flags[9] and select the action BusTrigger.





2. Start the program execution.

3. Open the ***TrBus*** window to watch the trigger.



Example for BusCount (Watchpoint)

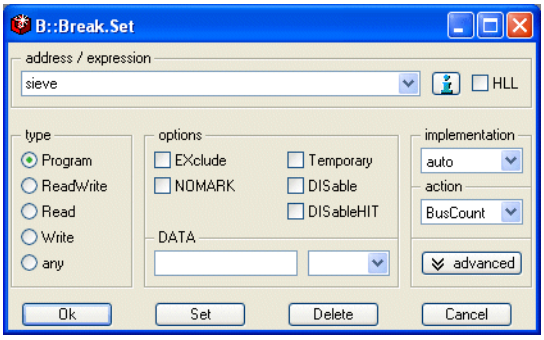
Resource: Watchpoints and logic in NEXUS Adapter (parallel trace only). Only one event possible.

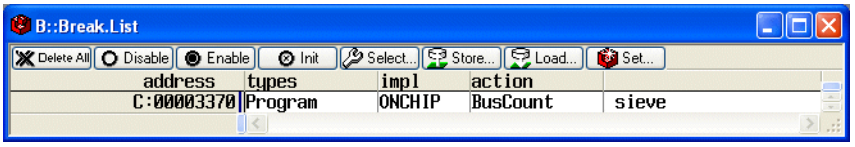
Controlled message types

WTM	BTM	DTM	OTM	DQM
Watchpoint Hit Message(s) is generated for the specified instruction(s) or data address+data value	Unaffected	Unaffected	Unaffected	Unaffected

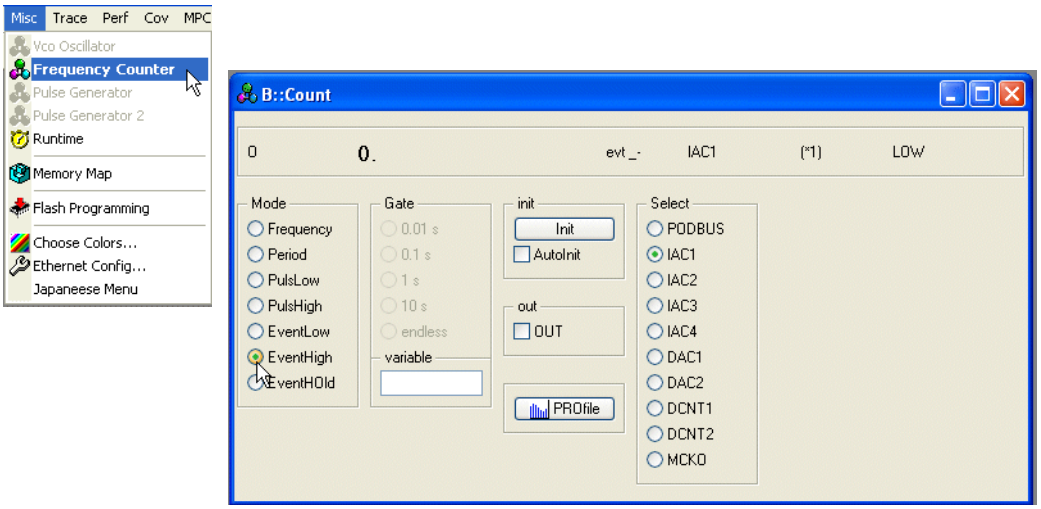
Example 1: Count how often the function sieve is called.

1. Set a program breakpoint to the start address of the function sieve and select the action BusCount.

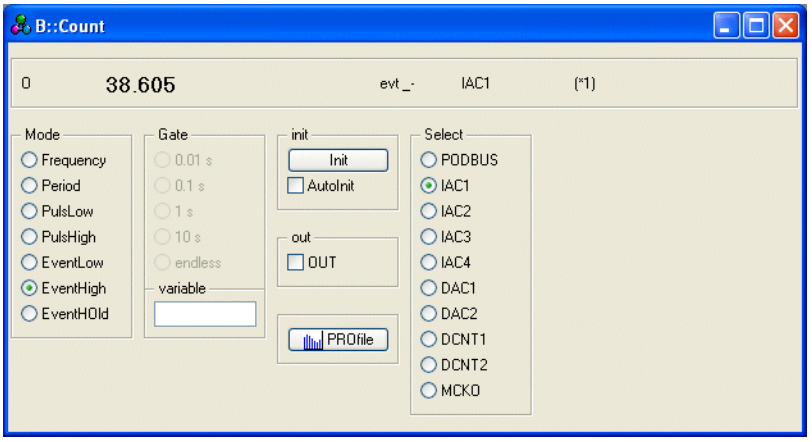




2. Open the TRACE32 counter window and select EventHigh.

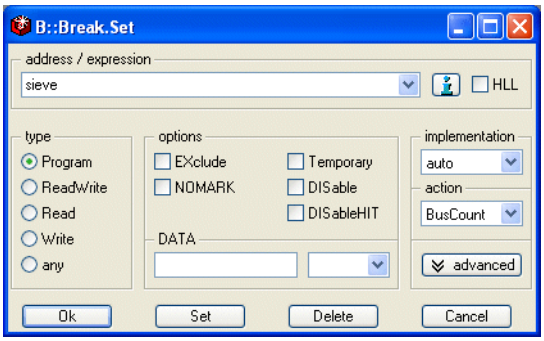


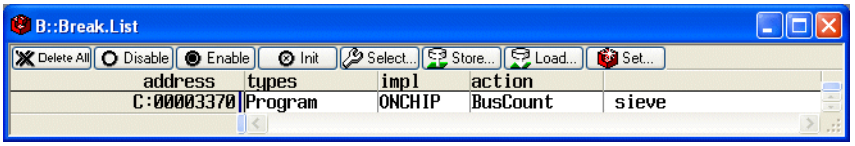
3. Start the program execution and display the result.



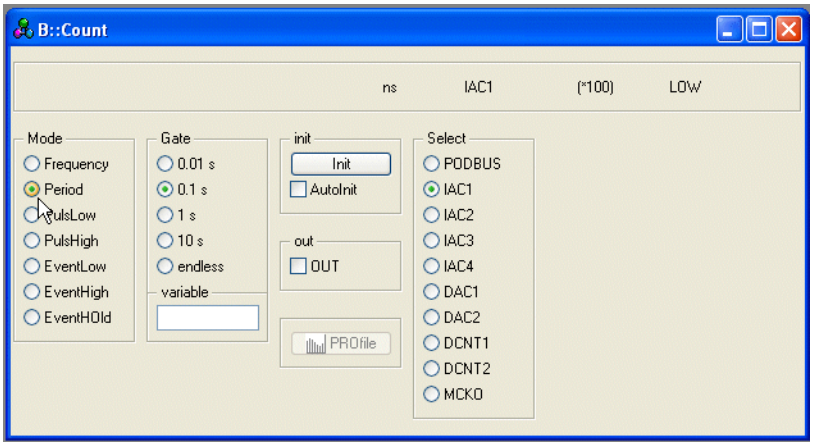
Example 2: Measure the period in which the function sieve is called.

1. Set a program breakpoint to the function sieve and select the action BusCount.

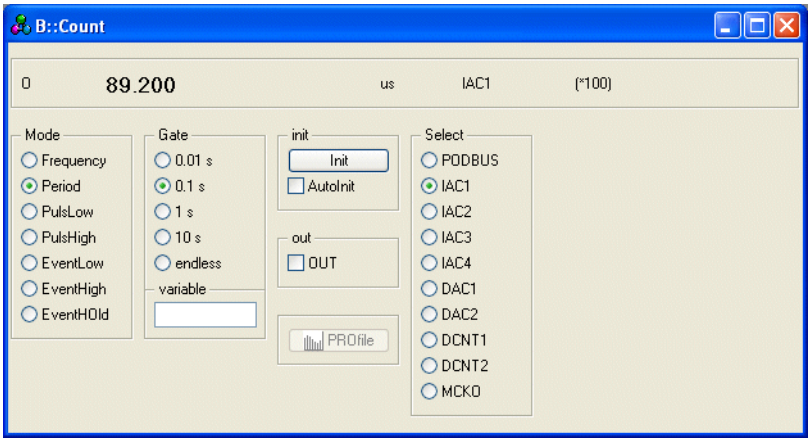




2. Open the TRACE32 counter window and select Period.

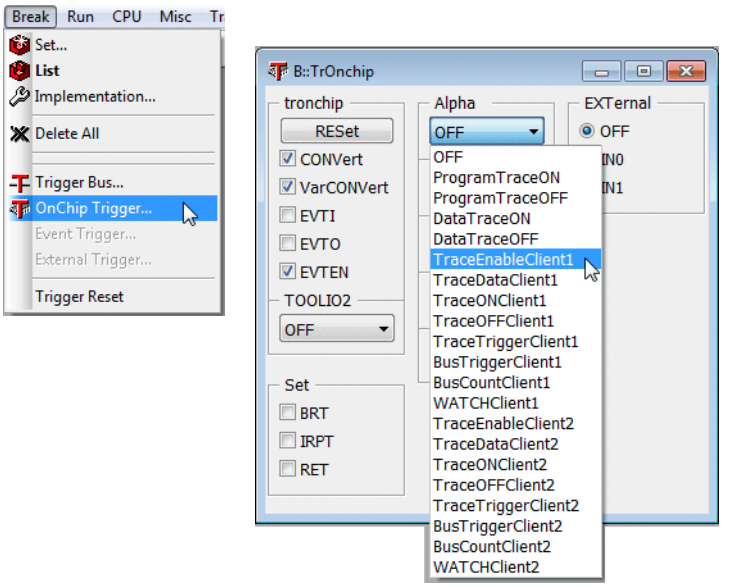


3. Start the program execution and display the result.

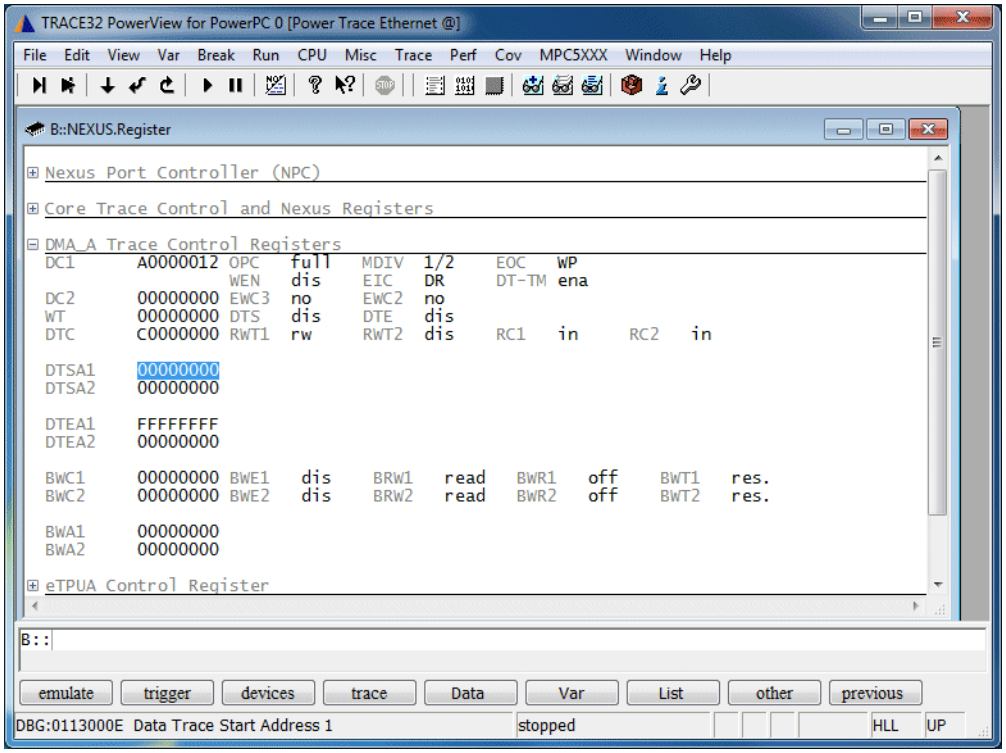


Filter and Trigger (Trace Clients)

The filter and trigger feature for the Trace Clients are provided via the ***TrOnchip*** window.



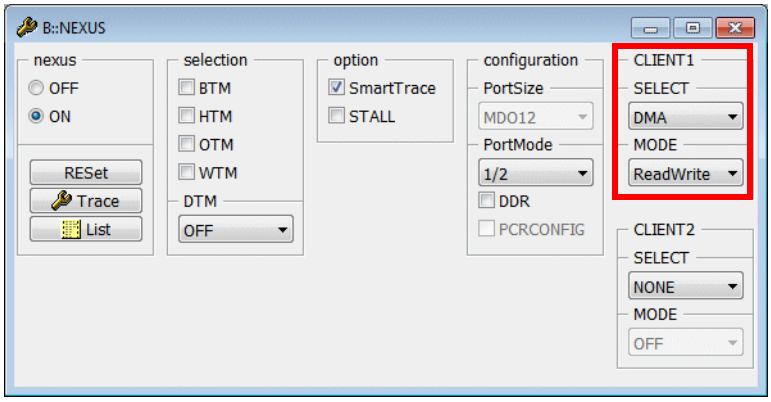
Trace Clients have their own resources in the NEXUS module. E.g. DMA client on MPC5554.



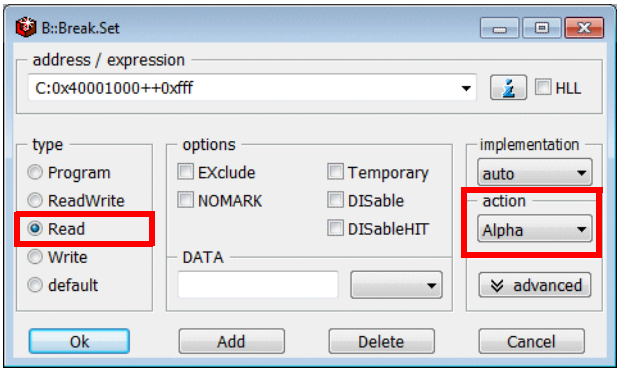
Example for TraceEnableClient1

Example MPC5554: Sample only DMA reads from address 0x40001000++0FFF.

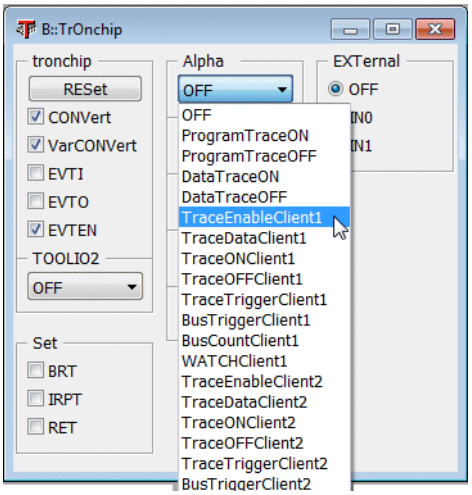
1. Select DMA a Trace Client1



2. Set a Read breakpoint to the address range 0x40001000++0FFF and select Alpha as breakpoint action.

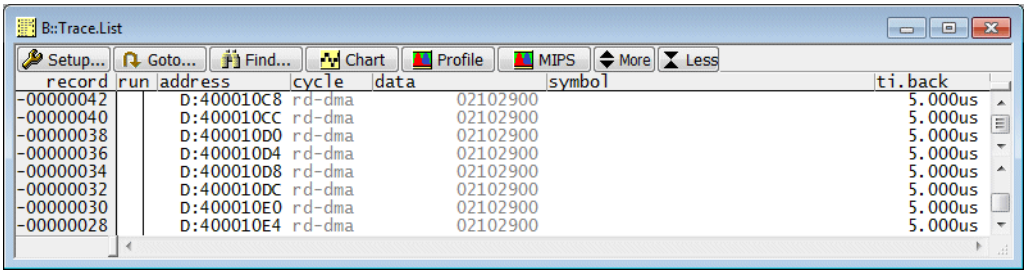


3. Select TraceEnableClient1 for Alpha in the ***TrOnchip*** window.



4. Start and stop the program.

5. Display the result.



Activate the TRACE32 OS Awareness

TRACE32 includes a configurable target-OS debugger to provide symbolic debugging of operating systems.

Since most users use an AUTOSAR operating system, this is taken as an example here.

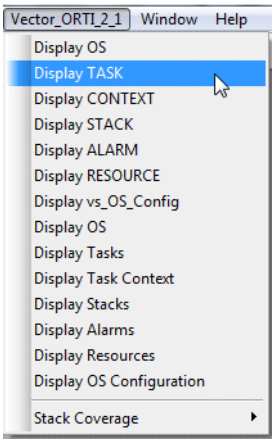
In order to provide AUTOSAR-aware tracing an ORTI file is required. The ORTI file is created by the AUTOSAR System Builder. It describes the structure and the memory mapping of the operating system objects.

Setup command:

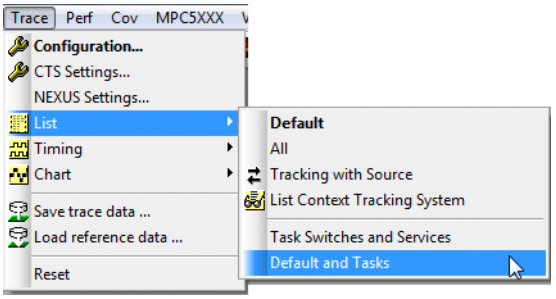
TASK.ORTI *<ORTI_file>*	Load the ORTI file into TRACE32

Loading the ORTI file results in the following:

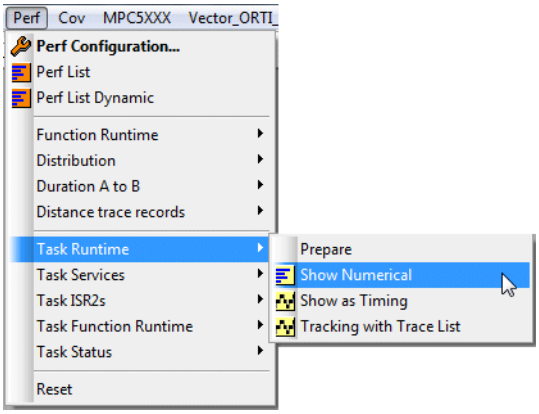
- Symbolic debugging of the OSEK OS is possible. Debug commands are provided via an ORTI menu.



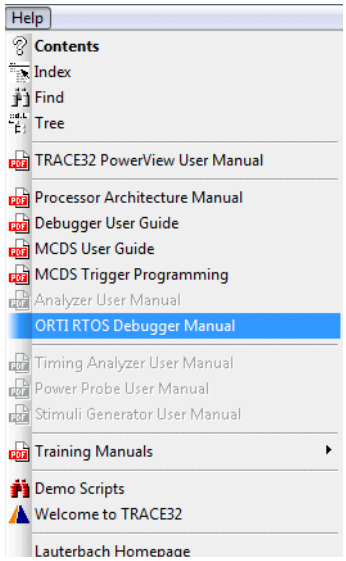
- The **Trace** menu is extended for OS-aware trace display.



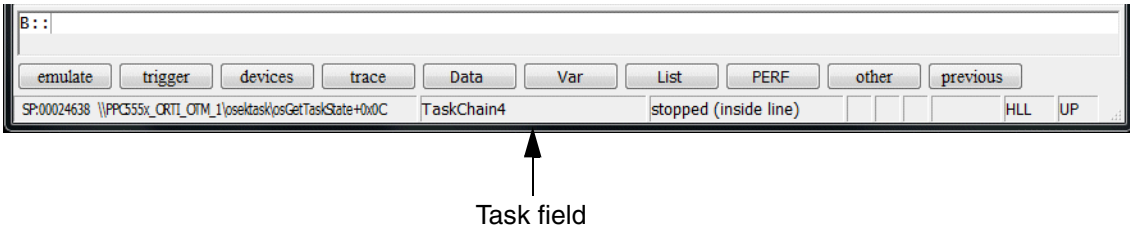
- The **Perf** menu is extended for OS-aware profiling.



- The manual of the OS Awareness for OSEK/**ORTI** is added to the **Help** menu.



- The name of the current task is displayed in the **Task** field of the TRACE32 state line.



Exporting Task Information (Overview)

There are two methods how task information can be generated by the NEXUS hardware module:

- **By generating an Ownership Trace Messages**

  This method should be used if supported by the OSEK operating system. It is the only method for NEXUS Class 2 Modules.

- **By generating trace information for a specific write access**

  This method requires a NEXUS Class 3 Module. It should be used, if the OSEK operating system does not support Ownership Trace Messages.

Exporting all Types of Task Information (OTM)

Ownership Trace Messages are generated when the OS updates

- the 8-bit Process ID register (PID0) - all compliant standards
- NEXUS PID Register (NPIDR) - IEEE-ISTO 5001-2012 compliant NEXUS module

PID0/NPIDR are updated by the OS on

- task switches
- entries and exits to service routines
- starts of ISR2 interrupt service routines and NO_ISR information

AUTOSAR OSs perform this update since 10/2010.

If you are using a IEEE-ISTO 5001-2003/2008 compliant NEXUS module and your task ID is longer the 8-bit, the PID0 register has to be updated in several steps. This requires special support from your OS. If your OS does not provide this special support, Lauterbach can provide you patch information. Please contact **support@lauterbach.com** for details.

The generation of Ownership Trace Messages has to be enabled within TRACE32.

```
NEXUS.OTM ON            ; enable the generation of Ownership Trace
                        ; Messages
```

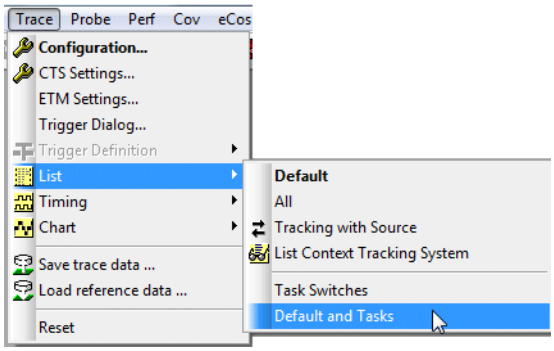
Example:

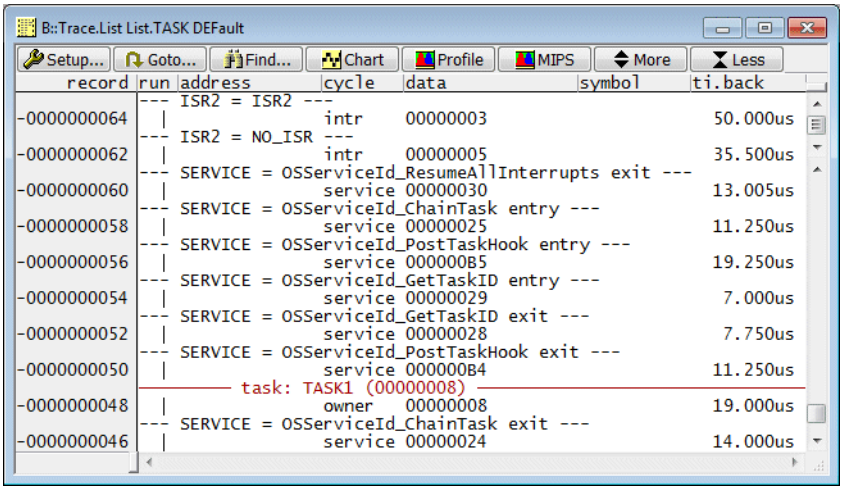
1. Advise the NEXUS hardware module to generate only Ownership Trace Messages.

```
NEXUS.BTM OFF                          ; disable the Branch Trace
                                       ; Messages

NEXUS.OTM ON                           ; enable the Ownership Trace
                                       ; Messages
```

2. Start and stop the program execution to fill the trace buffer.

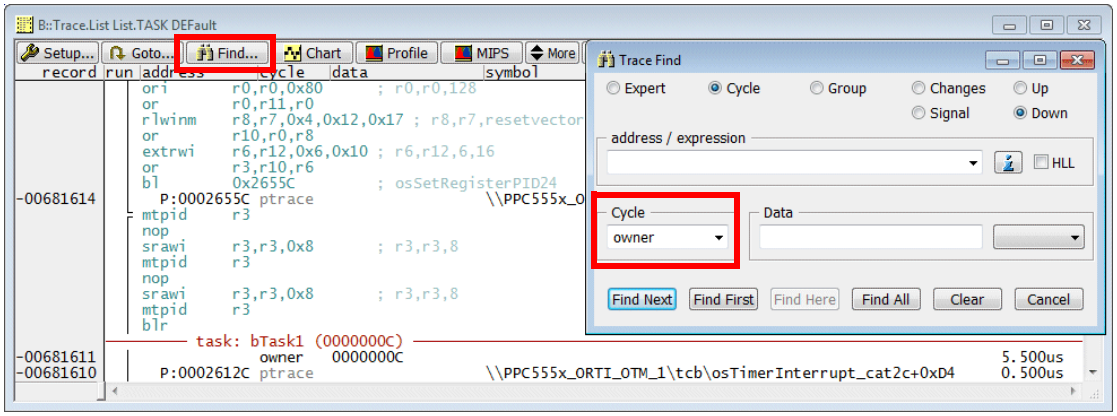
3. Display the result.



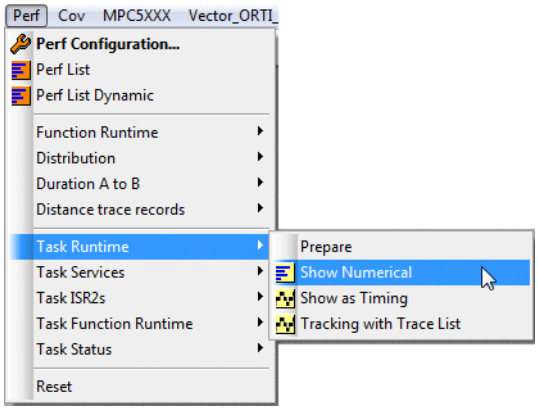


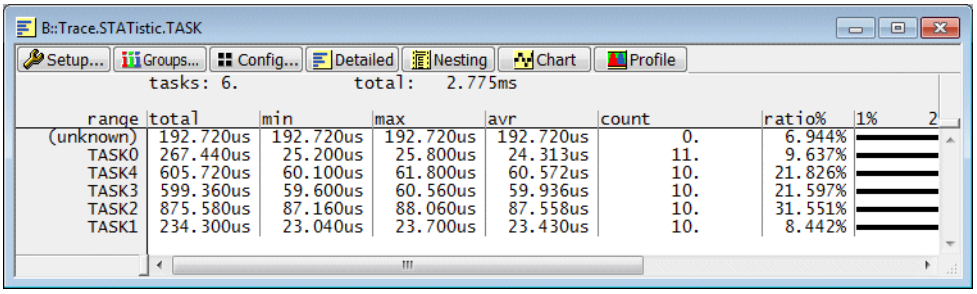
cycle types	
owner	Ownership trace message for task switches
service	Ownership trace message for entries and exits to OSEK service routines
intr	Ownership trace message for start of OSEK interrupt service routine and NO_ISR information

TRACE32 allows to search for all available cycle types e.g. owner:

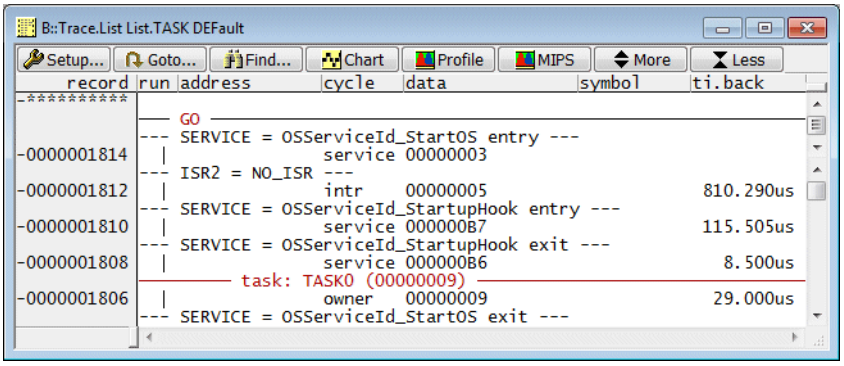


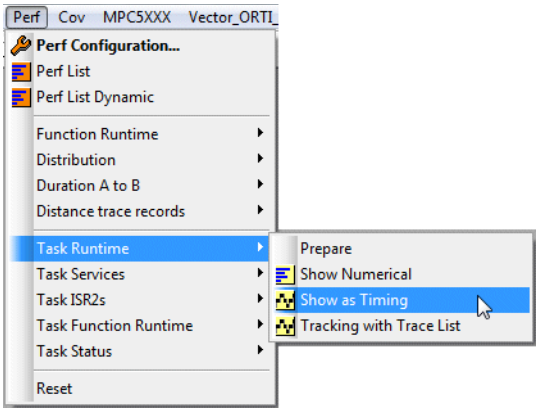
The following two commands perform a statistical analysis of the task switches:

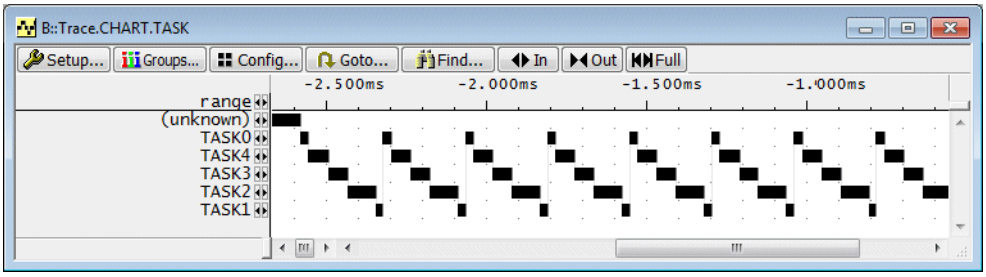




TRACE32 assigns all trace information generated before the first **task** information to the **(unknown)** task.

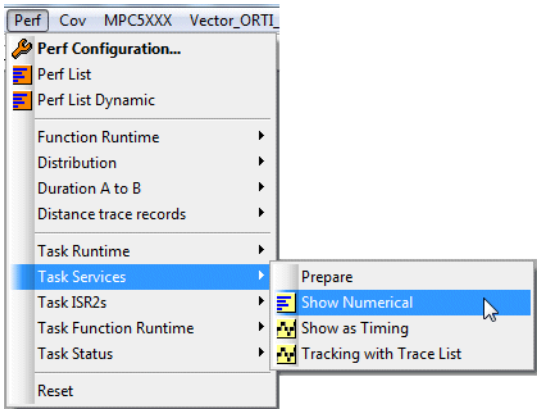




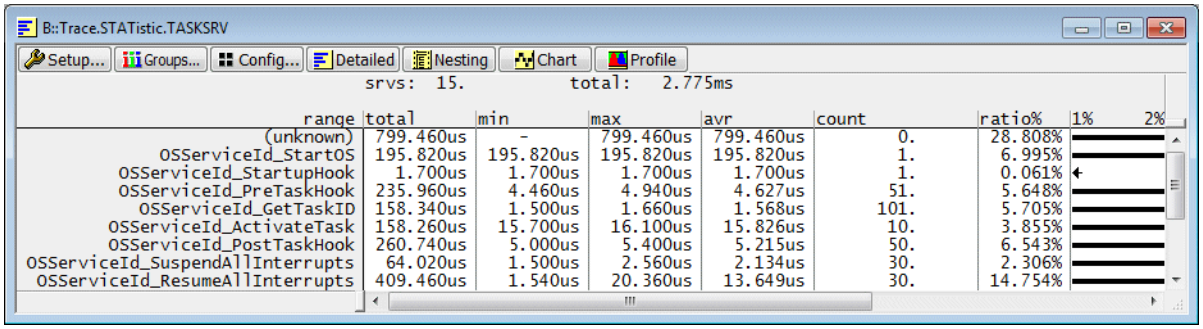


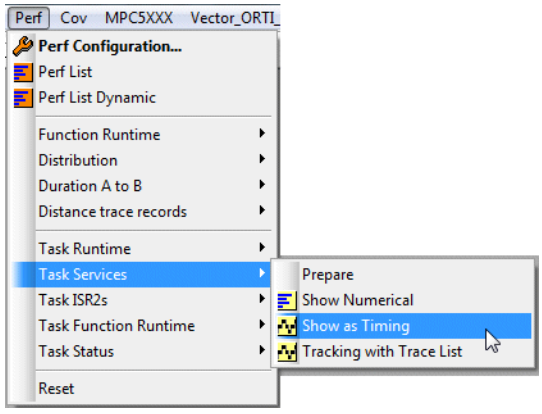
Trace.STATistic.TASK	Task runtime statistic
Trace.Chart.TASK	Task runtime time chart

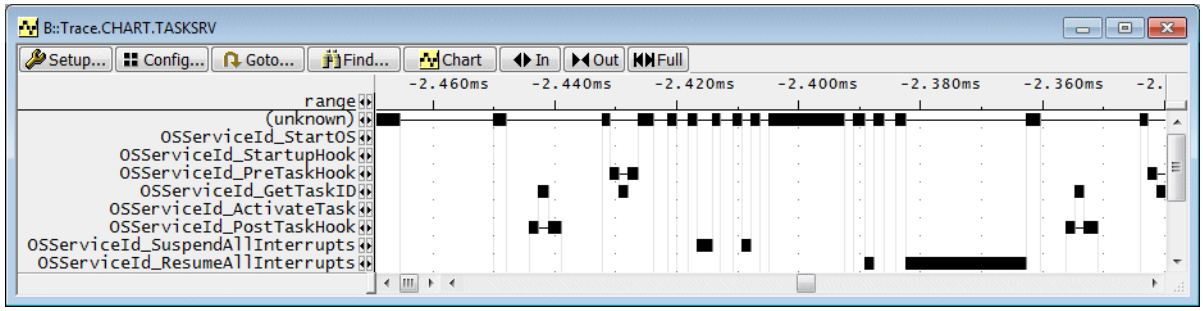
The following two commands perform a statistical analysis of the OSEK service routines:



(unknown) represents the time in which the processor/core is not in an OSEK service routine

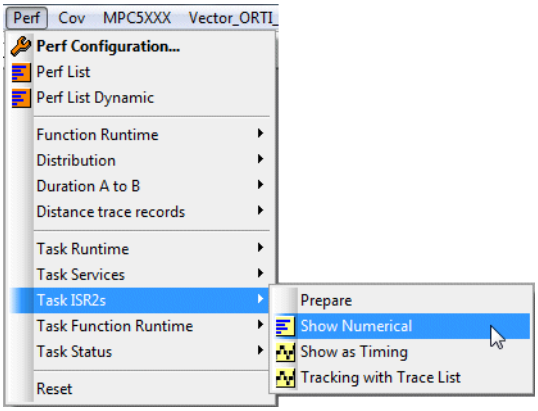


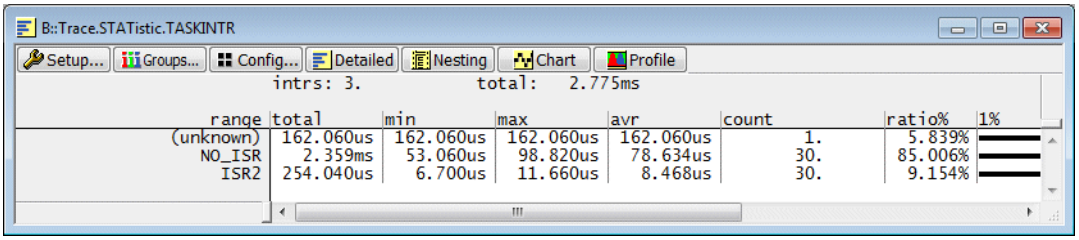




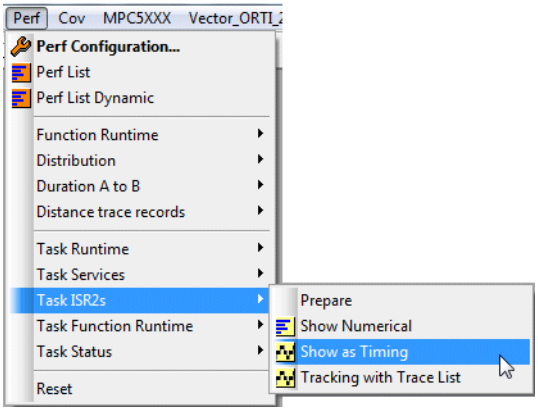
Trace.STATistic.TASKSRV	Statistic on service routines
Trace.Chart.TASKSRV	Time chart on service routines

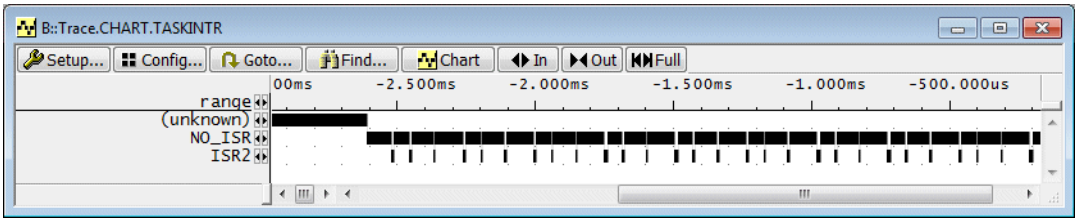
The following two commands perform a statistical analysis of the OSEK interrupt service routines:





TRACE32 assigns all trace information generated before the first **intr** information to **(unknown)**.

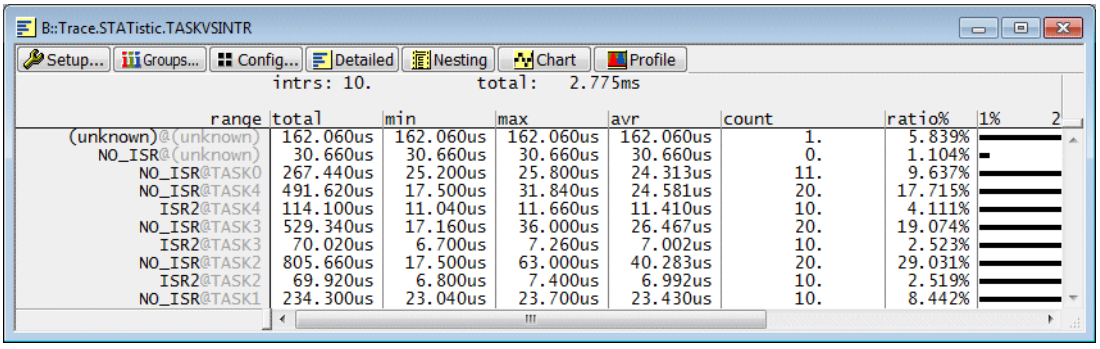




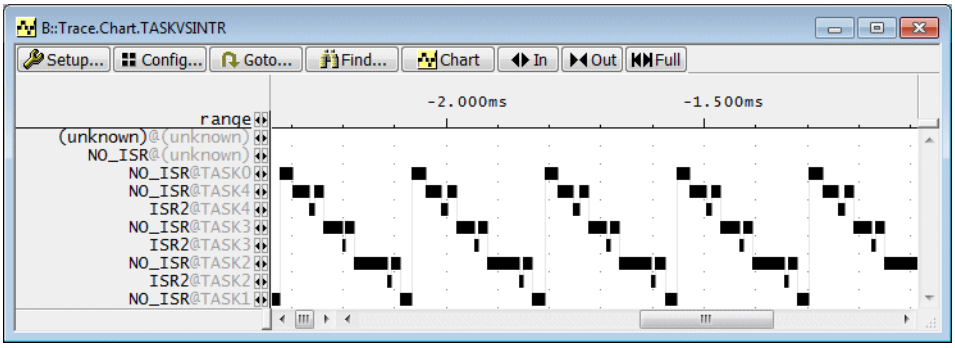
Trace.STATistic.TASKINTR	Statistic on interrupt service routines
Trace.Chart.TASKINTR	Time chart on interrupt service routines

The following command allows to perform a statistical analysis of the OSEK interrupt service routines related to the active tasks.

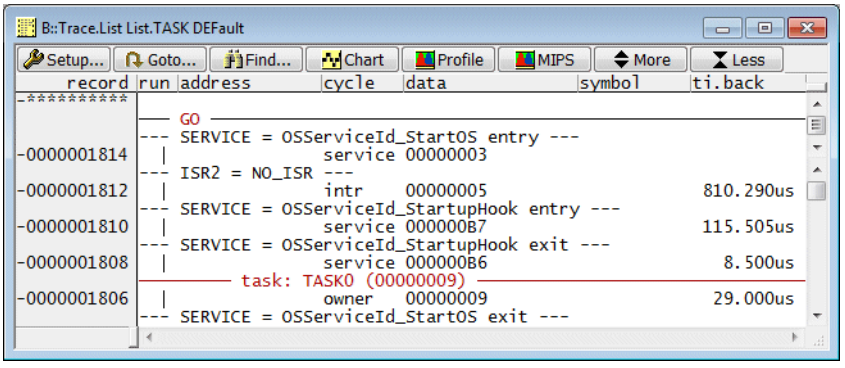
Trace.STATistic.TASKVSINTR	Task-related statistic on interrupt service routines
Trace.Chart.TASKVSINTR	Time-chart for task related interrupt service routines



range	total	min	max	avr	count	ratio%	1% 2
(unknown)@(unknown)	162.060us	162.060us	162.060us	162.060us	1.	5.839%	
NO_ISR@(unknown)	30.660us	30.660us	30.660us	30.660us	0.	1.104%	
NO_ISR@TASK0	267.440us	25.200us	25.800us	24.313us	11.	9.637%	
NO_ISR@TASK4	491.620us	17.500us	31.840us	24.581us	20.	17.715%	
ISR2@TASK4	114.100us	11.040us	11.660us	11.410us	10.	4.111%	
NO_ISR@TASK3	529.340us	17.160us	36.000us	26.467us	20.	19.074%	
ISR2@TASK3	70.020us	6.700us	7.260us	7.002us	10.	2.523%	
NO_ISR@TASK2	805.660us	17.500us	63.000us	40.283us	20.	29.031%	
ISR2@TASK2	69.920us	6.800us	7.400us	6.992us	10.	2.519%	
NO_ISR@TASK1	234.300us	23.040us	23.700us	23.430us	10.	8.442%	



intr information that was generated before the first **task** information is assigned to the **@(unknown)** task.



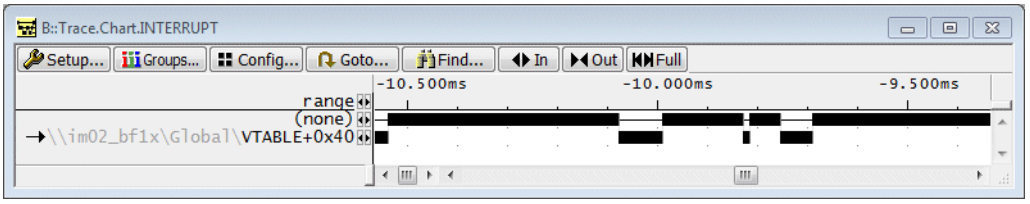
Exporting all Types of Task Information and all Instructions (OTM)

General setup:

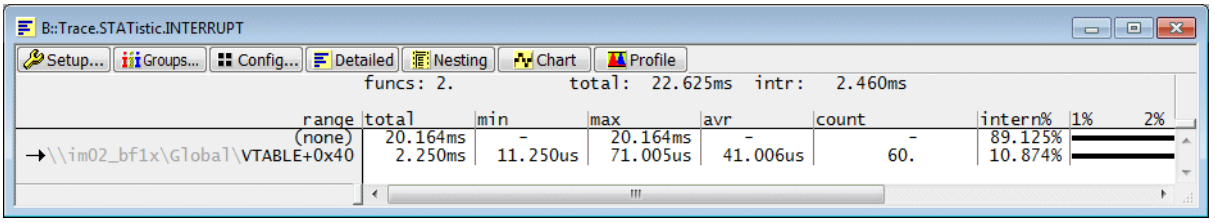
```
NEXUS.BTM ON                                  ; enable the Branch Trace
                                              ; Messages

NEXUS.OTM ON                                  ; enable the Ownership Trace
                                              ; Messages

Trace.STATistic.InterruptIsFunction ON        ; advise TRACE32 to regard the
                                              ; time between interrupt entry
                                              ; and exit as function
```

Statistic Analysis of Interrupts

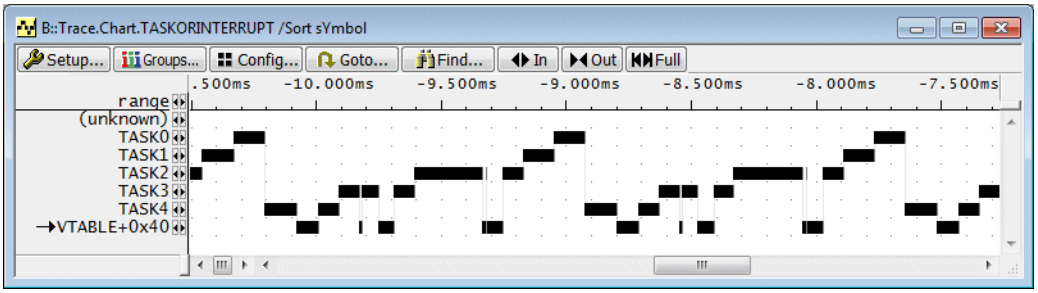
Trace.Chart.INTERRUPT Interrupt time chart



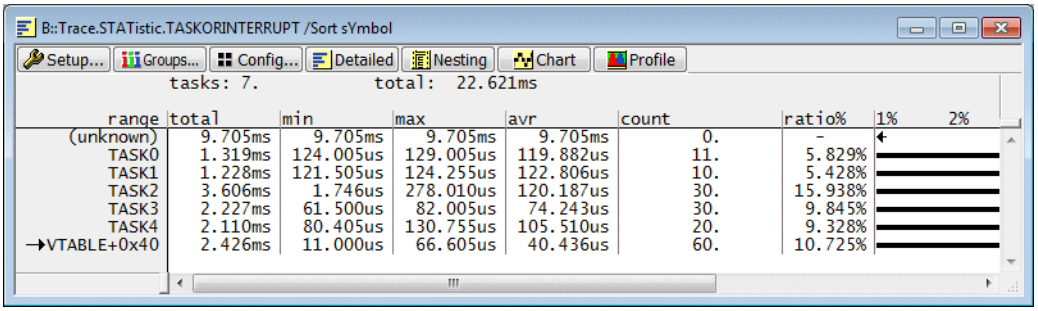
Trace.STATistic.INTERRUPT Interrupt statistic



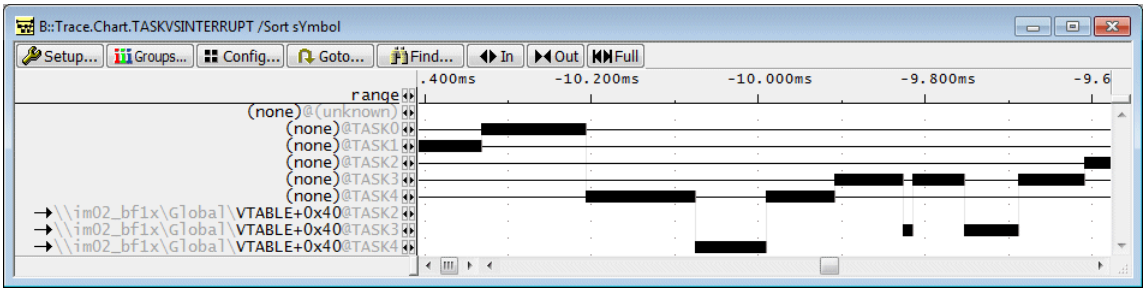
Trace.Chart.TASKORINTERRUPT Time chart of interrupts and tasks



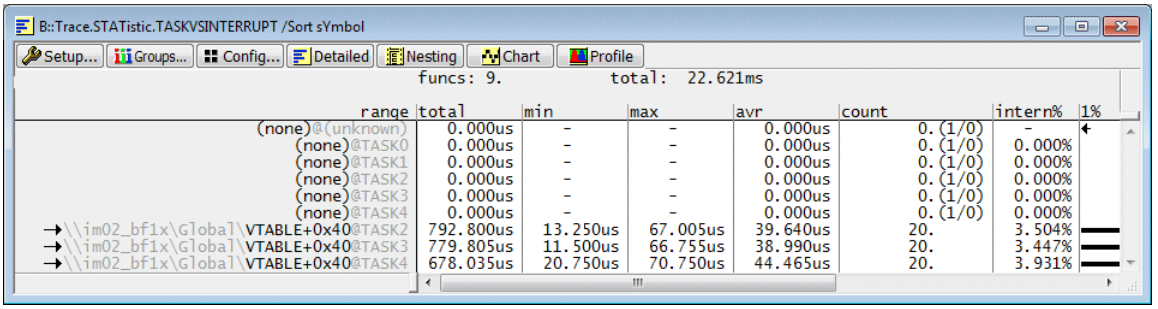
Trace.STATistic.TASKORINTERRUPT Statistic of interrupts and tasks



Trace.Chart.TASKVSINTERRUPT Time chart interrupts, task-related



Trace.STATistic.TASKVSINTERRUPT Statistic of interrupts, task-related



range	total	min	max	avr	count	intern%	1%
(none)@(unknown)	0.000us	-	-	0.000us	0.(1/0)	-	
(none)@TASK0	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK1	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK2	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK3	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK4	0.000us	-	-	0.000us	0.(1/0)	0.000%	
\\im02_bf1x\Global\VTABLE+0x40@TASK2	792.800us	13.250us	67.005us	39.640us	20.	3.504%	
\\im02_bf1x\Global\VTABLE+0x40@TASK3	779.805us	11.500us	66.755us	38.990us	20.	3.447%	
\\im02_bf1x\Global\VTABLE+0x40@TASK4	678.035us	20.750us	70.750us	44.465us	20.	3.931%	

Task Switches

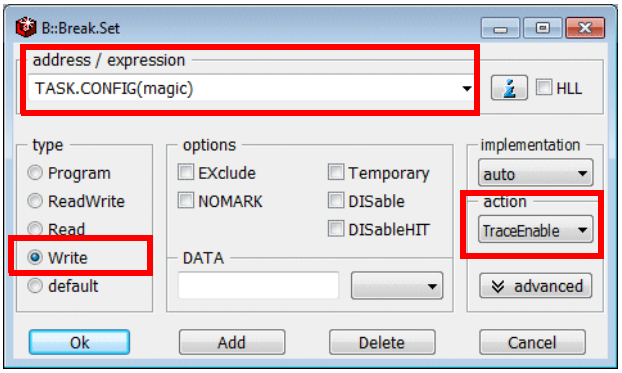
Each operating system has a variable that contains the information which task is currently running. One way to export task switch information is to advise the NEXUS hardware module to generate trace information when a write access to this variable occurs.

The address of this variable is provided by the TRACE32 function **TASK.CONFIG(magic)**.

```
PRINT TASK.CONFIG(magic)                ; print the address of the variable
                                        ; that holds the task identifier
```

Example: Advise the NEXUS hardware module to generate only trace information on task switches.

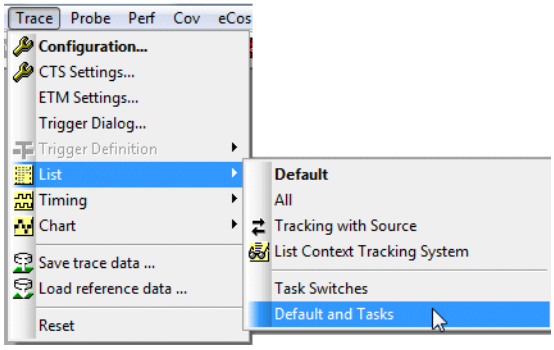
1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic) and select the trace action TraceEnable.

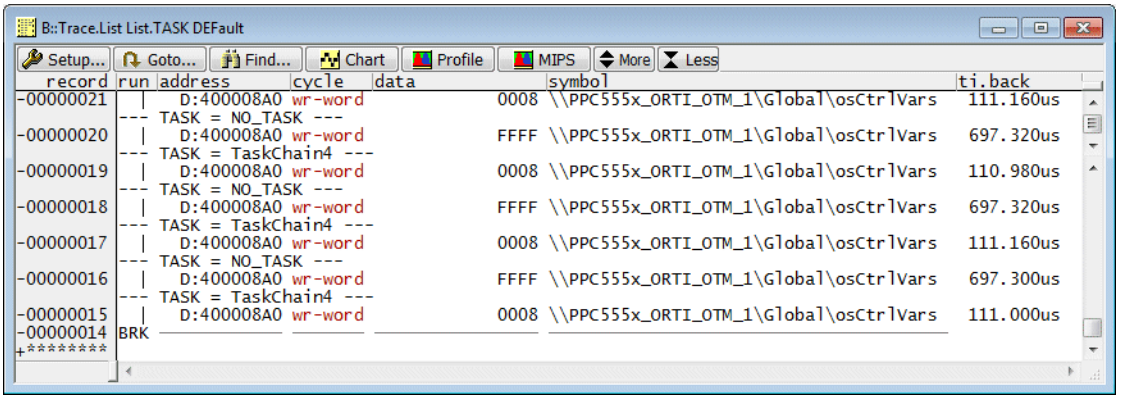



```
Break.Set TASK.CONFIG(magic) /Write /TraceEnable
```

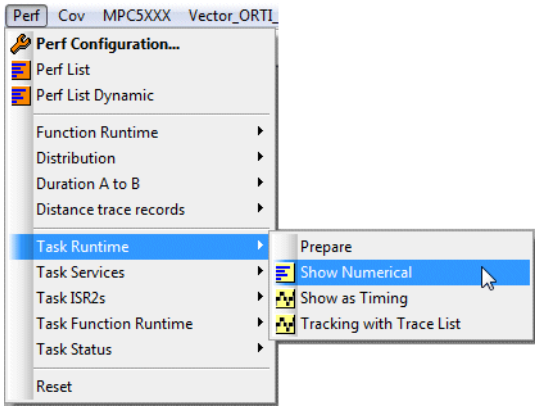
2. Start and stop the program execution to fill the trace buffer

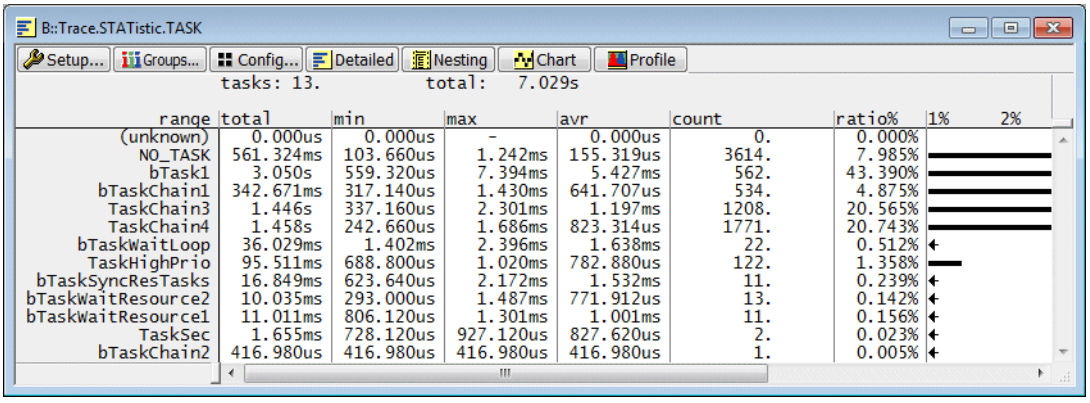
3. Display the result.

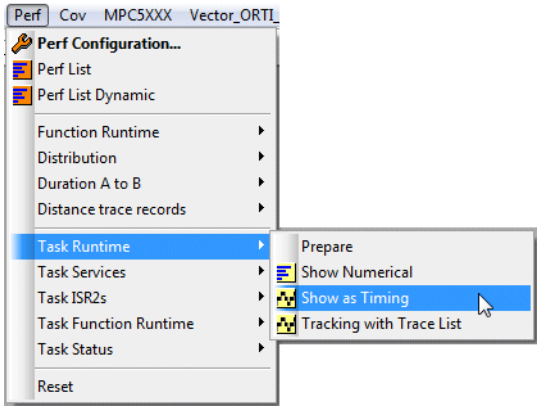


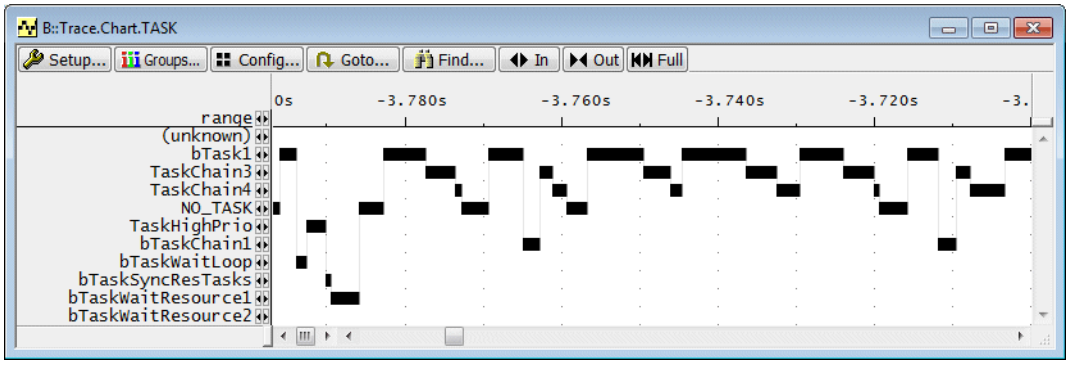


The following two commands perform a statistical analysis of the task switches:









Trace.STATistic.TASK	Task runtime statistic
Trace.Chart.TASK	Task runtime time chart

OSEK Service Routines

The time spent in OSEK service routines can be evaluated.

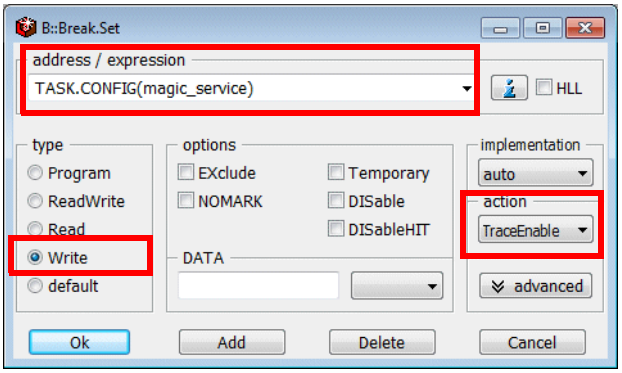
OSEK writes information on the entries and exits to OSEK service routines to a defined variable. One way to export information on OSEK service routines is to advise the NEXUS hardware module to generate trace information when a write access to this variable occurs.

The address of this variable is provided by the TRACE32 function **TASK.CONFIG(magic_service)**.

```
PRINT TASK.CONFIG(magic_service)      ; print the address of the variable
                                      ; that holds the service
                                      ; information
```

Example: Advise the NEXUS hardware module to generate only trace information for entries and exits to OSEK service routines.

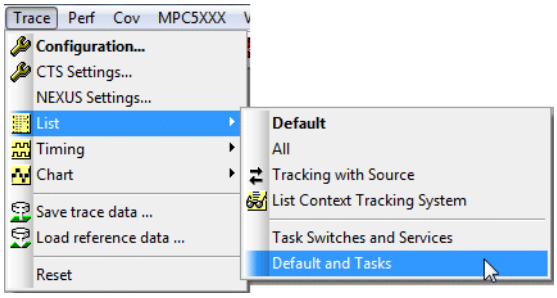
1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic_service) and select the trace action TraceEnable.

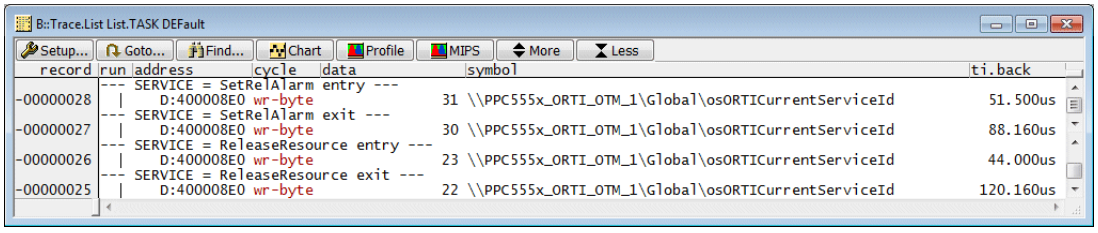



```
Break.Set TASK.CONFIG(magic_service) /Write /TraceEnable
```

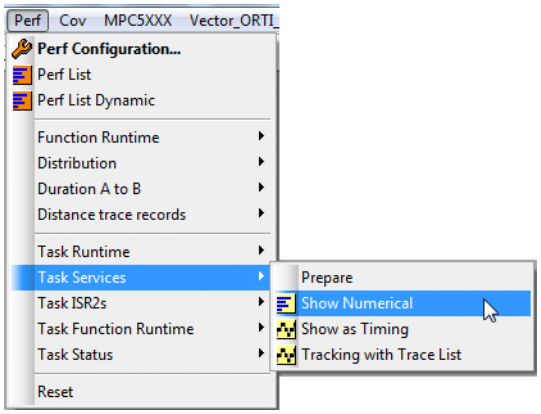
2. Start and stop the program execution to fill the trace buffer

3. Display the result.

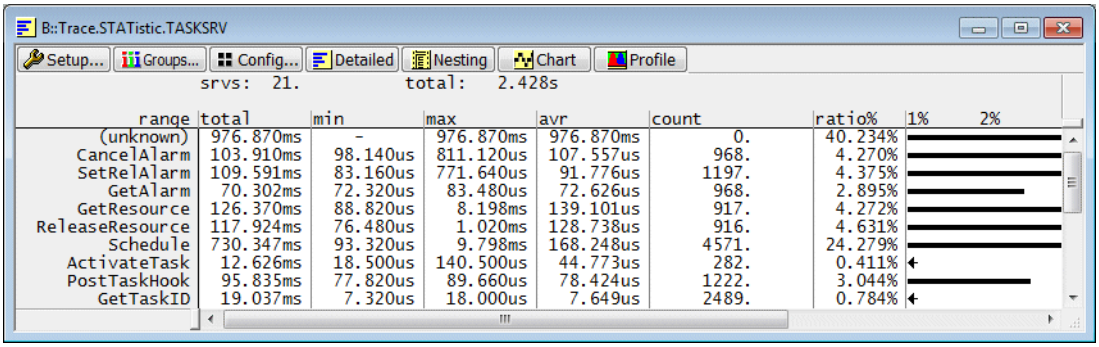




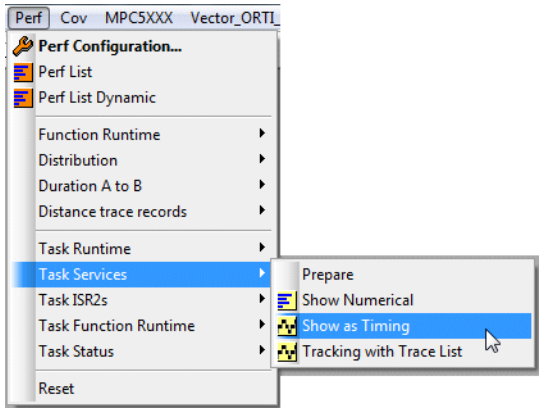
The following two commands perform a statistical analysis of the OSEK service routines:

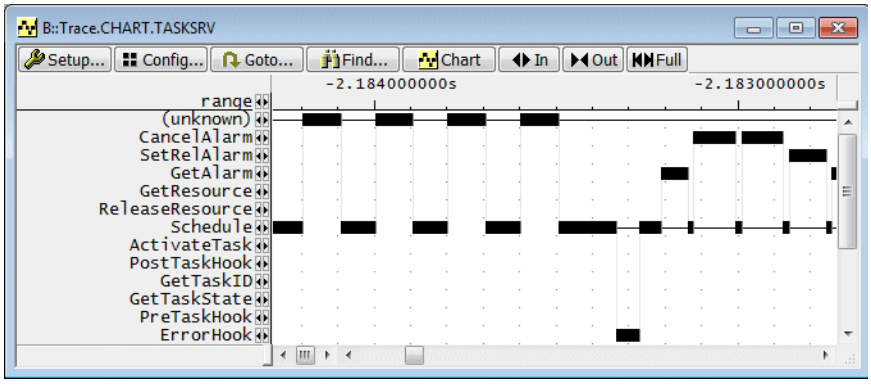


(unknown) represents the time in which the processor/core is not in an OSEK service routine



range	total	min	max	avr	count	ratio%	1% 2%
(unknown)	976.870ms	-	976.870ms	976.870ms	0.	40.234%	
CancelAlarm	103.910ms	98.140us	811.120us	107.557us	968.	4.270%	
SetRelAlarm	109.591ms	83.160us	771.640us	91.776us	1197.	4.375%	
GetAlarm	70.302ms	72.320us	83.480us	72.626us	968.	2.895%	
GetResource	126.370ms	88.820us	8.198ms	139.101us	917.	4.272%	
ReleaseResource	117.924ms	76.480us	1.020ms	128.738us	916.	4.631%	
Schedule	730.347ms	93.320us	9.798ms	168.248us	4571.	24.279%	
ActivateTask	12.626ms	18.500us	140.500us	44.773us	282.	0.411%	
PostTaskHook	95.835ms	77.820us	89.660us	78.424us	1222.	3.044%	
GetTaskID	19.037ms	7.320us	18.000us	7.649us	2489.	0.784%	





Trace.STATistic.TASKSRV	Statistic on service routines
Trace.Chart.TASKSRV	Time chart on service routines

OSEK ISR2s

The time spent in OSEK interrupt service routine can be evaluated.

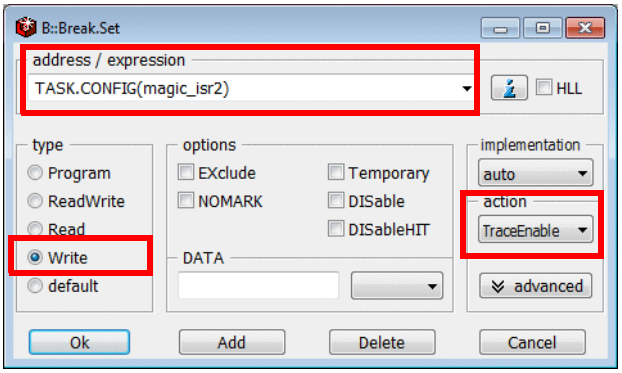
OSEK writes information on the start of an interrupt service routine to a defined variable as well as the information NO_ISR. One way to export information on OSEK interrupt service routine is to advise the NEXUS hardware module to generate trace information when a write access to this variable occurs.

The address of this variable is provided by the TRACE32 function **TASK.CONFIG(magic_isr2)**.

```
PRINT TASK.CONFIG(magic_isr2)          ; print the address of the variable
                                       ; that holds the interrupt service
                                       ; information
```

Example: Advise the NEXUS hardware module to generate only trace information on the start of an interrupt service routine as well as on the information NO_ISR.

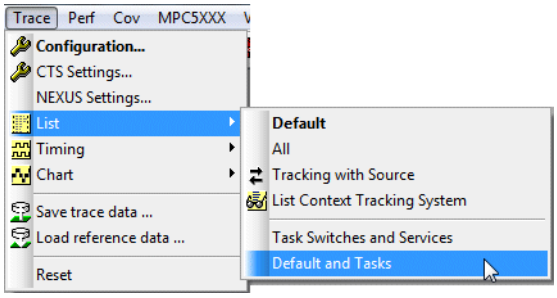
1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic_isr2) and select the trace action TraceEnable.

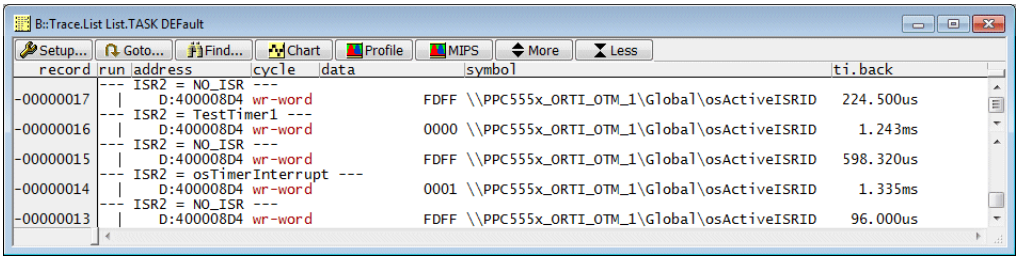



```
Break.Set TASK.CONFIG(magic_isr2) /Write /TraceEnable
```

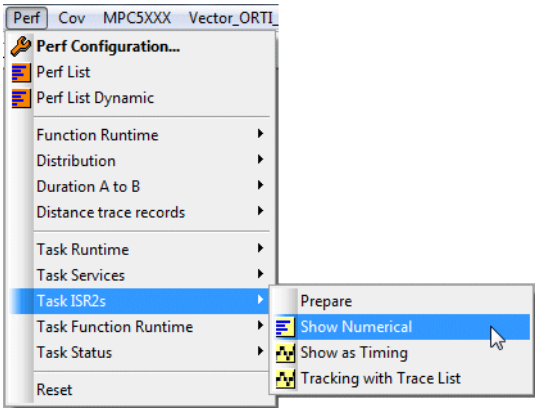
2. Start and stop the program execution to fill the trace buffer

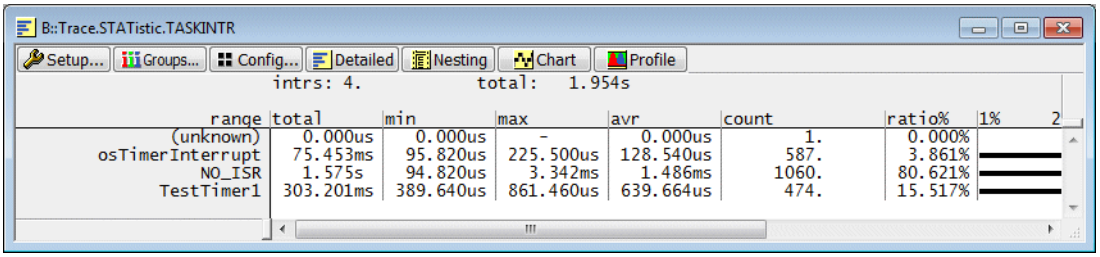
3. Display the result.

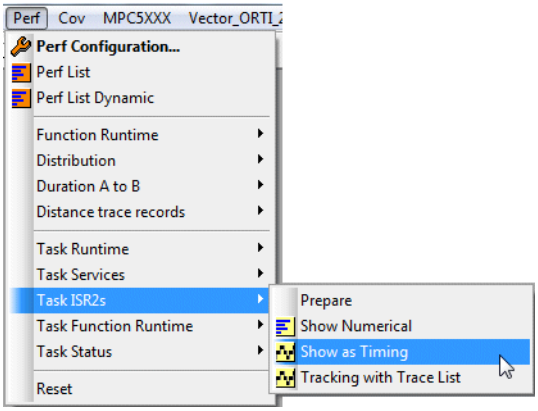


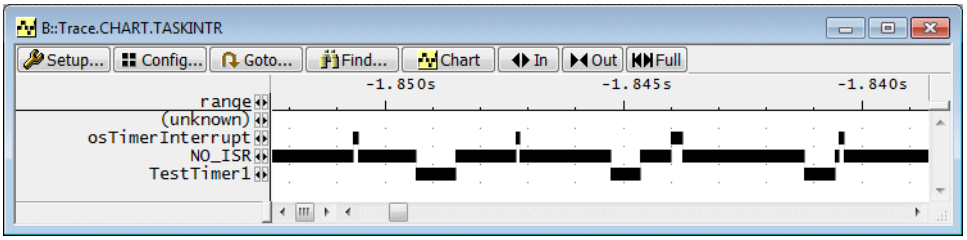


The following two commands perform a statistical analysis of the OSEK interrupt service routines:









Trace.STATistic.TASKINTR	Statistic on interrupt service routines
Trace.Chart.TASKINTR	Time chart on interrupt service routines

OSEK interrupt service routines that occur in multiple tasks can be displayed per task, if the following information is available:

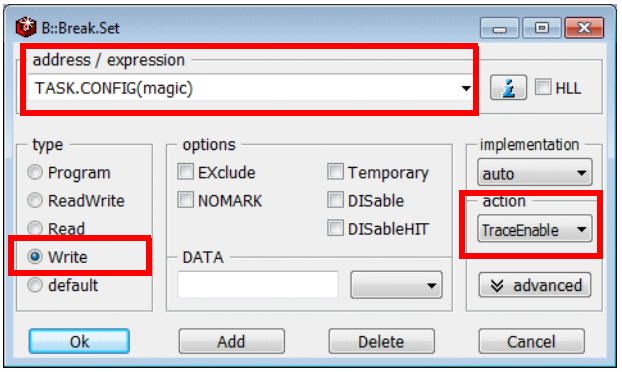
- Task switch information
- ISR2 start and NO_ISR information

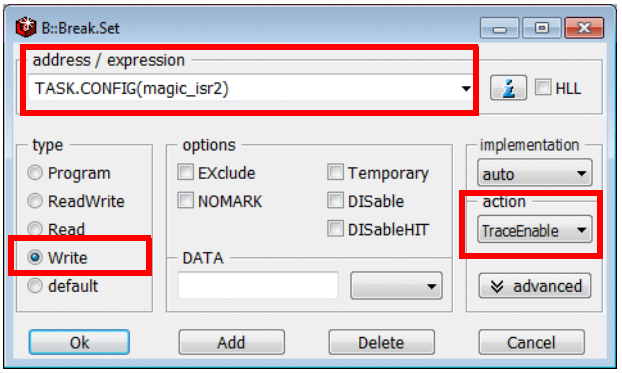
Example:

1. Advise the NEXUS hardware module to generate only trace information

   - on task switches

   - on the start of an interrupt service routine as well as on the information NO_ISR

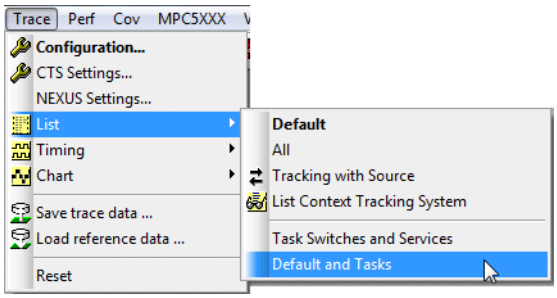


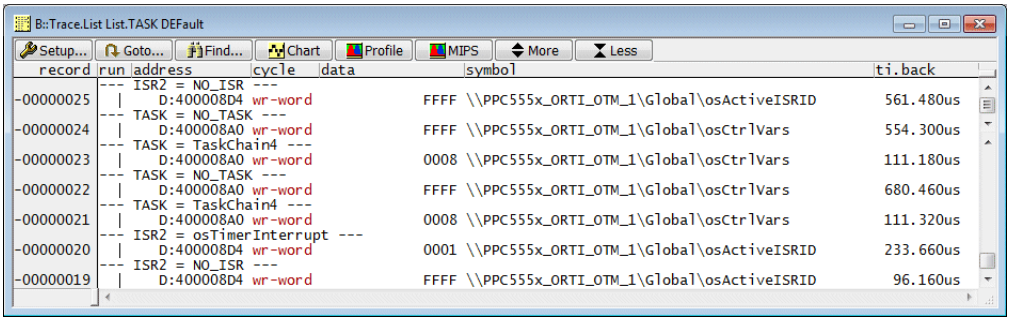



```
Break.Set TASK.CONFIG(magic) /Write /TraceEnable

Break.Set TASK.CONFIG(magic_isr2) /Write /TraceEnable
```

2. Start and stop the program execution to fill the trace buffer.

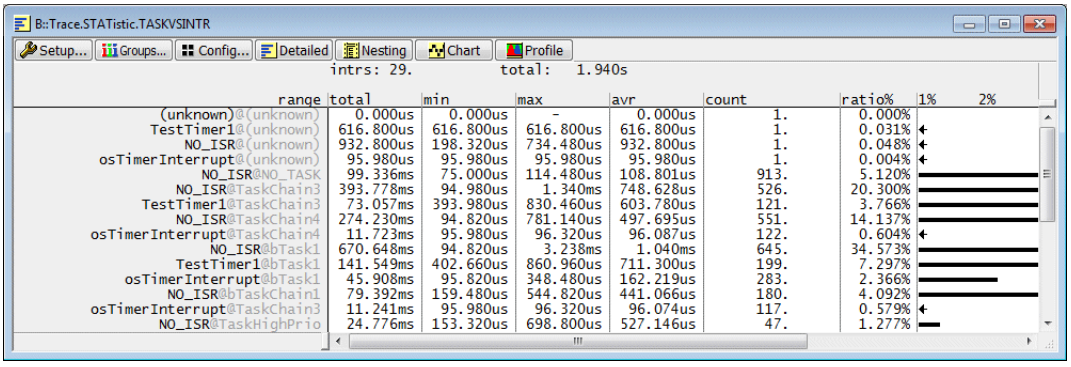
3. Display the result.

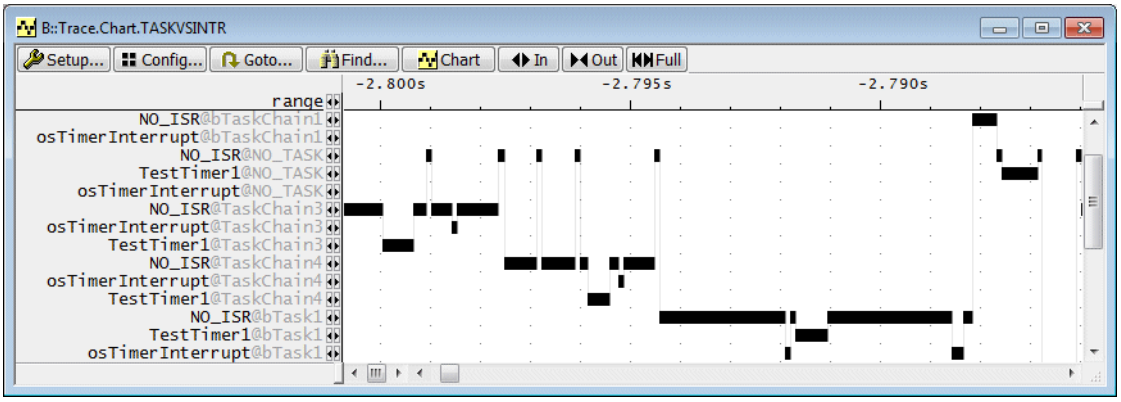




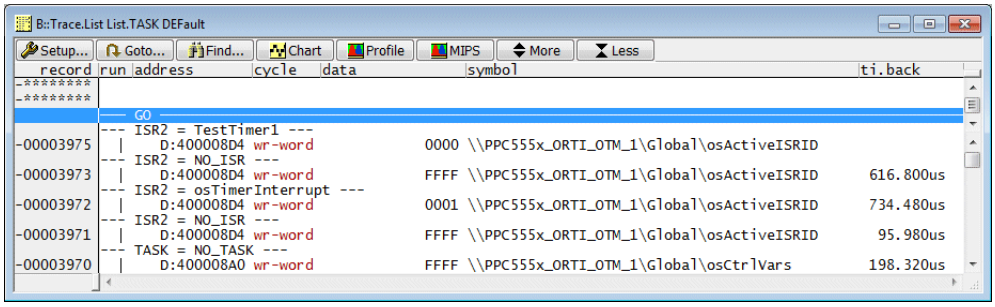
The following command allows to perform a statistical analysis of the OSEK interrupt service routines related to the active tasks.

Trace.STATistic.TASKVSINTR	Task-related statistic on interrupt service routines
Trace.Chart.TASKVSINTR	Time-chart on task related interrupt service routines





ISR2 information that was generated before the first TASK information is assigned to the **@(unknown)** task.



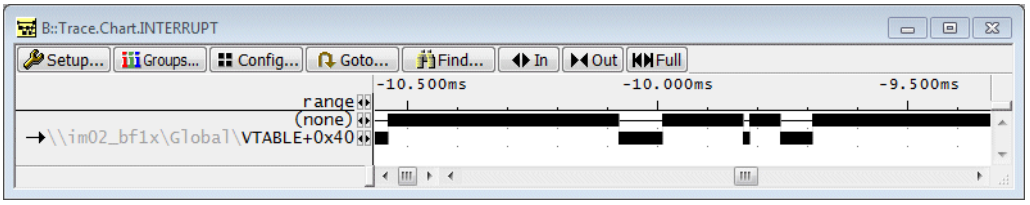
Exporting Task Switches and all Instructions (Write Access)

General setup:

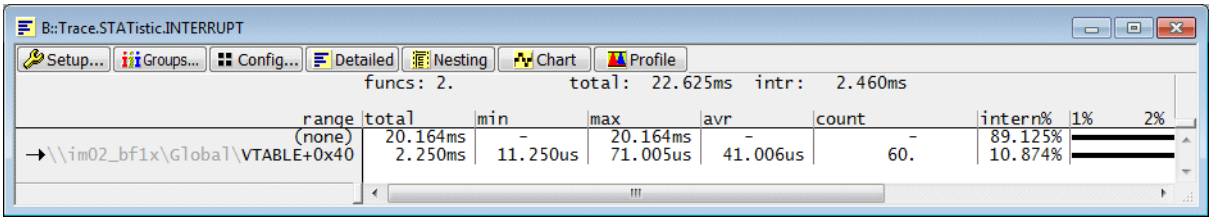
```
Break.Set TASK.CONFIG(magic) /Write /TraceData

; advise TRACE32 to regard the time between interrupt entry
; and exit as function
Trace.STATistic.InterruptIsFunction ON
```

Statistic Analysis of Interrupts

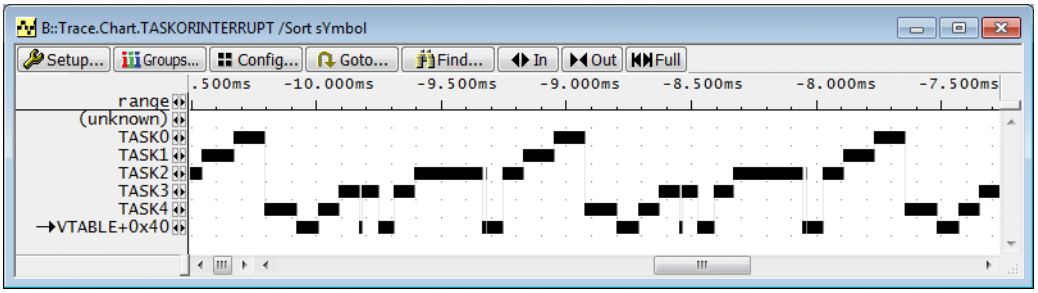
Trace.Chart.INTERRUPT Interrupt time chart



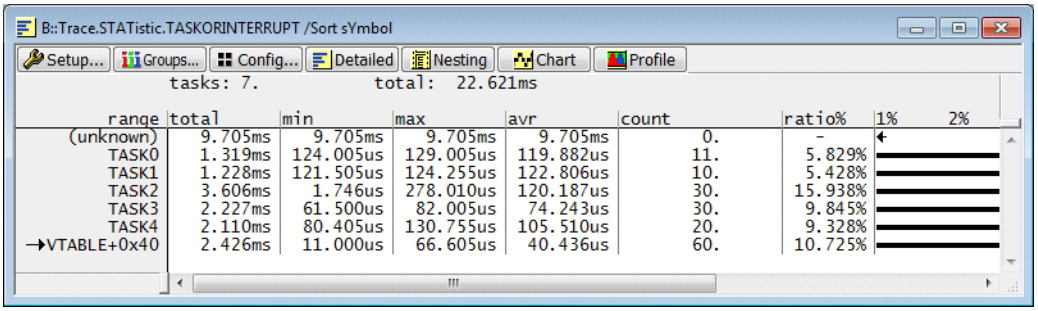
Trace.STATistic.INTERRUPT Interrupt statistic



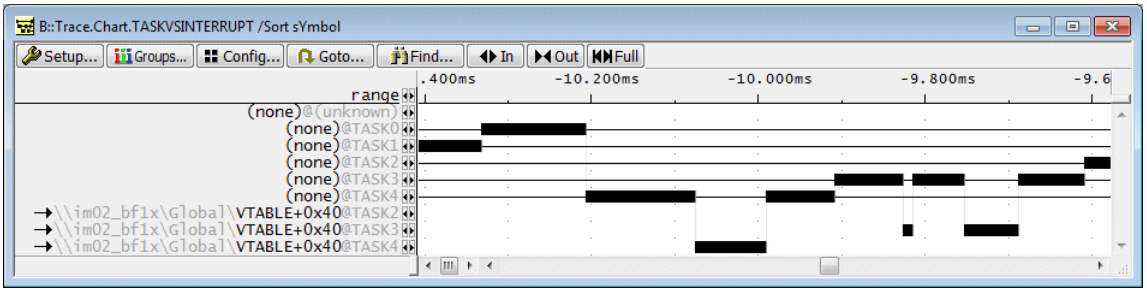
Trace.Chart.TASKORINTERRUPT Time chart of interrupts and tasks



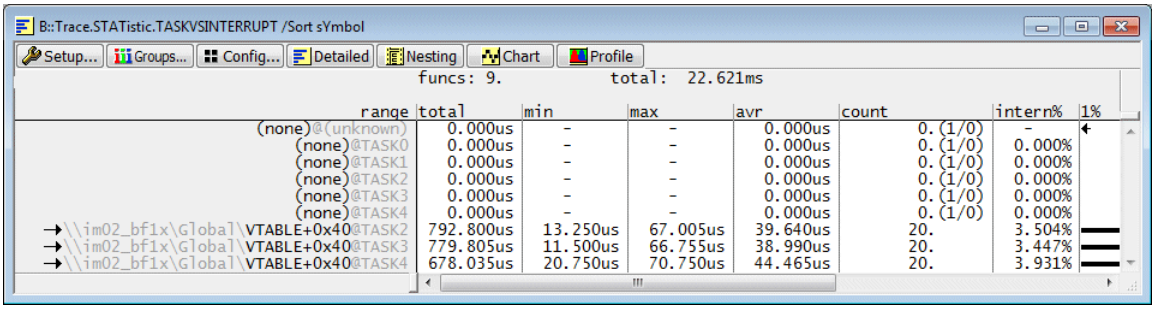
Trace.STATistic.TASKORINTERRUPT Statistic of interrupts and tasks



Trace.Chart.TASKVSINTERRUPT Time chart interrupts, task-related



Trace.STATistic.TASKVSINTERRUPT Statistic of interrupts, task-related



range	total	min	max	avr	count	intern%	1%
(none)@(unknown)	0.000us	-	-	0.000us	0.(1/0)	-	
(none)@TASK0	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK1	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK2	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK3	0.000us	-	-	0.000us	0.(1/0)	0.000%	
(none)@TASK4	0.000us	-	-	0.000us	0.(1/0)	0.000%	
\\im02_bf1x\Global\VTABLE+0x40@TASK2	792.800us	13.250us	67.005us	39.640us	20.	3.504%	
\\im02_bf1x\Global\VTABLE+0x40@TASK3	779.805us	11.500us	66.755us	38.990us	20.	3.447%	
\\im02_bf1x\Global\VTABLE+0x40@TASK4	678.035us	20.750us	70.750us	44.465us	20.	3.931%	

Belated Trace Analysis (OS)

The TRACE32 Instruction Set Simulator can be used for a belated OS-aware trace evaluation. To set up the TRACE32 Instruction Set Simulator for belated OS-aware trace evaluation proceed as follows:

1. Save the trace information for the belated evaluation to a file.

```
Trace.SAVE belated__orti.ad
```

2. Set up the TRACE32 Instruction Set Simulator for a belated OS-aware trace evaluation (here OSEK on a MPC5553):

```
SYStem.CPU MPC5553                          ; select the target CPU

SYStem.Up                                   ; establish the
                                            ; communication between
                                            ; TRACE32 and the TRACE32
                                            ; Instruction Set
                                            ; Simulator

Trace.LOAD belated_orti.ad                  ; load the trace file

Data.Load.ELF my_app.out /NoCODE /GHS       ; load the symbol and
                                            ; debug information

TASK.ORTI my_orti.ort                       ; load the ORTI file

Trace.List List.TASK DEFault                ; display the trace
                                            ; listing
```

Exporting all Types of Task Information (OTM)

Ownership Trace Messages are generated when the OS updates

- the 8-bit Process ID register (PID0) - IEEE-ISTO 5001-2003 compliant NEXUS module
- NEXUS PID Register (NPIDR) - IEEE-ISTO 5001-2008 compliant NEXUS module and subsequent standards

PID0 respectively NPIDR is updated on

- task switches
- entries and exits to OSEK service routines
- start of OSEK interrupt service routines and start of NO_ISR code

The ORTI standard support task-aware tracing via OTMs since October/2010.

If you are using a IEEE-ISTO 5001-2003 compliant NEXUS Class 2 module and your task ID is longer the 8-bit, the PID0 register has to be updated in several steps. This requires special support from your OSEK system. If your OSEK system does not provide this special support, Lauterbach can provide you patch information. Please contact support@lauterbach.com for details.

The generation of Ownership Trace Messages has to be enabled within TRACE32.

```
NEXUS.OTM ON            ; enable the generation of Ownership Trace
                        ; Messages
```

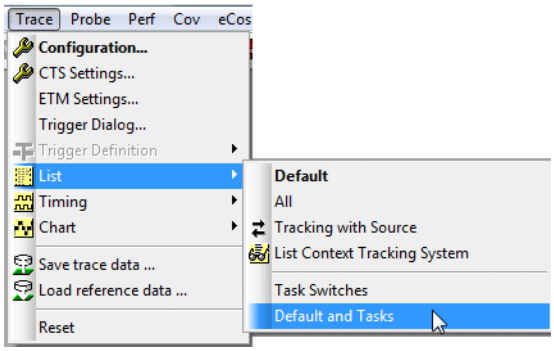
Example:

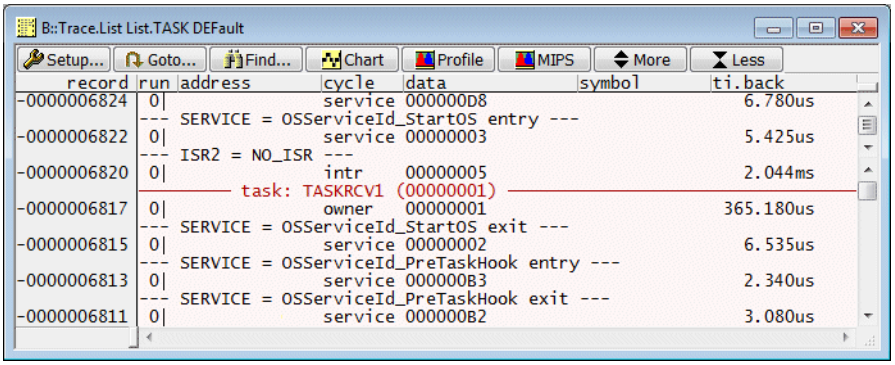
1. Advise the NEXUS hardware module to generate only Ownership Trace Messages.

```
NEXUS.BTM OFF                          ; disable the Branch Trace
                                       ; messaging

NEXUS.OTM ON                           ; enable the Ownership Trace
                                       ; Messages
```

2. Start and stop the program execution to fill the trace buffer.

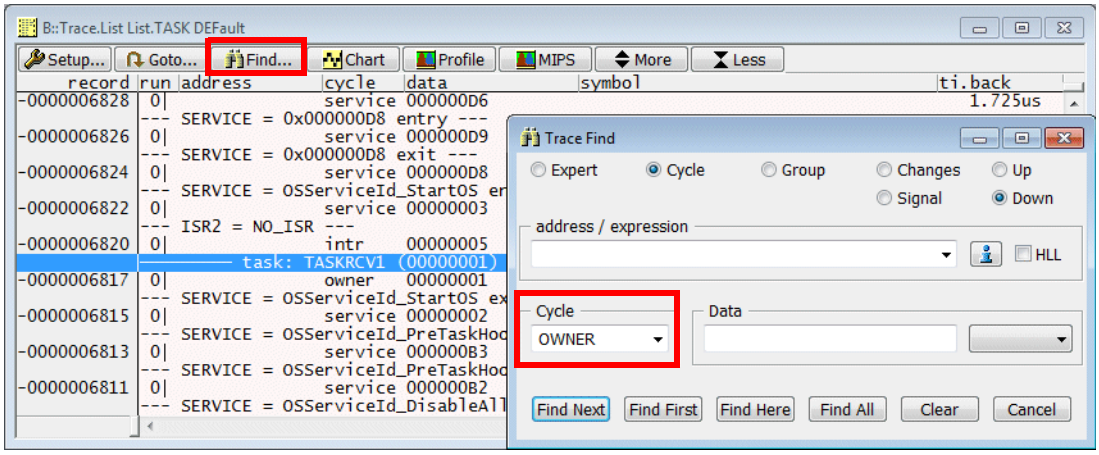
3. Display the result.



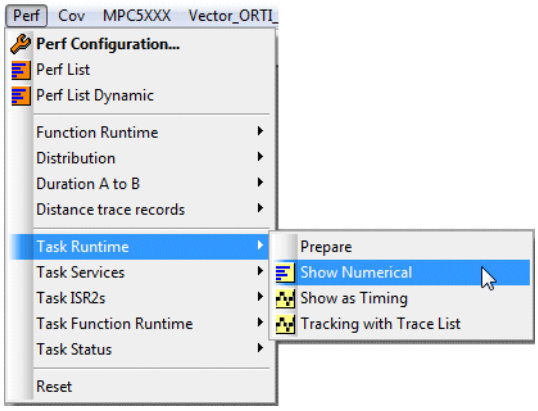


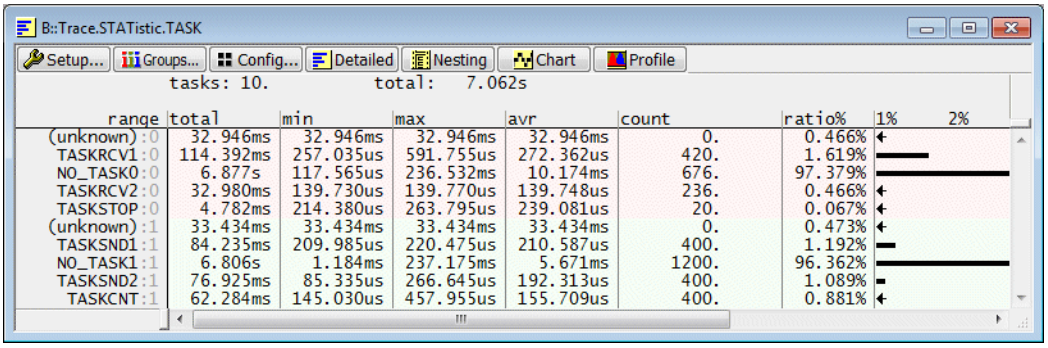
cycle types	
owner	Ownership trace message for task switches
service	Ownership trace message for entries and exits to OSEK service routines
intr	Ownership trace message for start of OSEK interrupt service routine and start of NO_ISR code

TRACE32 allows to search for all available cycle types e.g. owner:

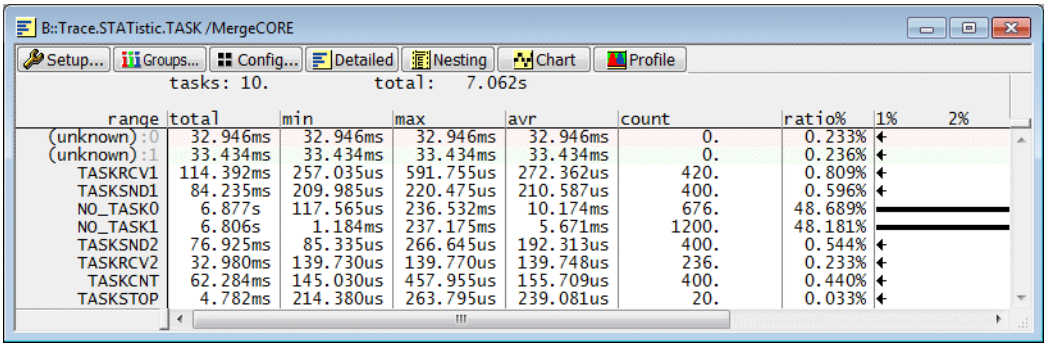


The following commands perform a statistical analysis of the task switches:





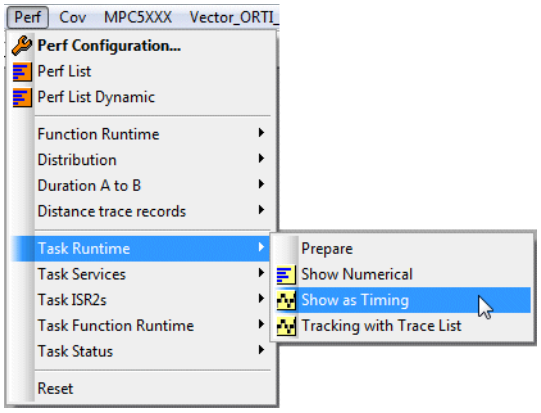
Trace.STATistic.TASK [/SplitCORE] Task runtime statistic, result per core

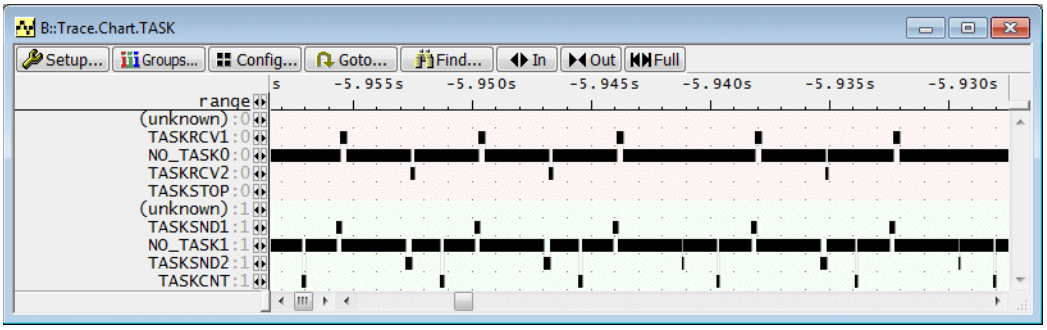


Trace.STATistic.TASK /MergeCORE Task runtime statistic, results of all cores merged

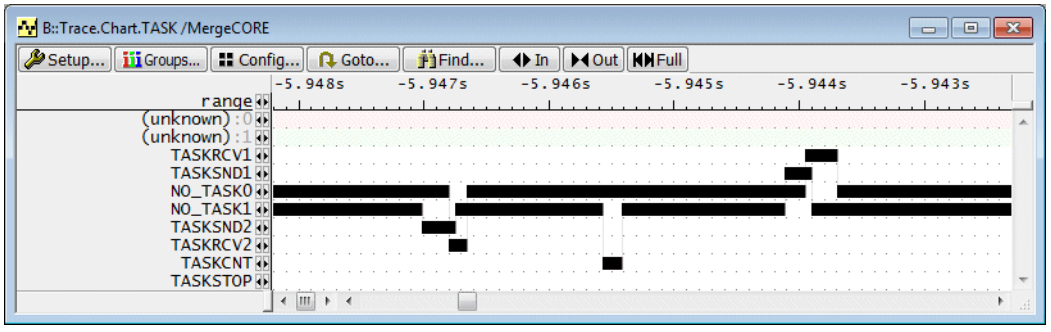
TRACE32 assigns all trace information generated before the first **task** information to the **(unknown)** tasks. The **(unknown)** tasks are always displayed per core.

The following commands display a time-chart of the task run-times:



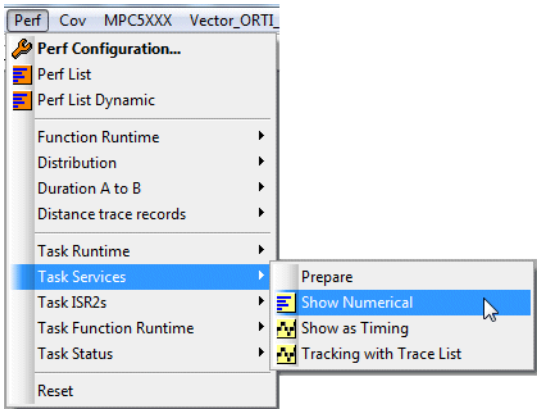


Trace.Chart.TASK [/SplitCORE] Task runtime time chart, result per core

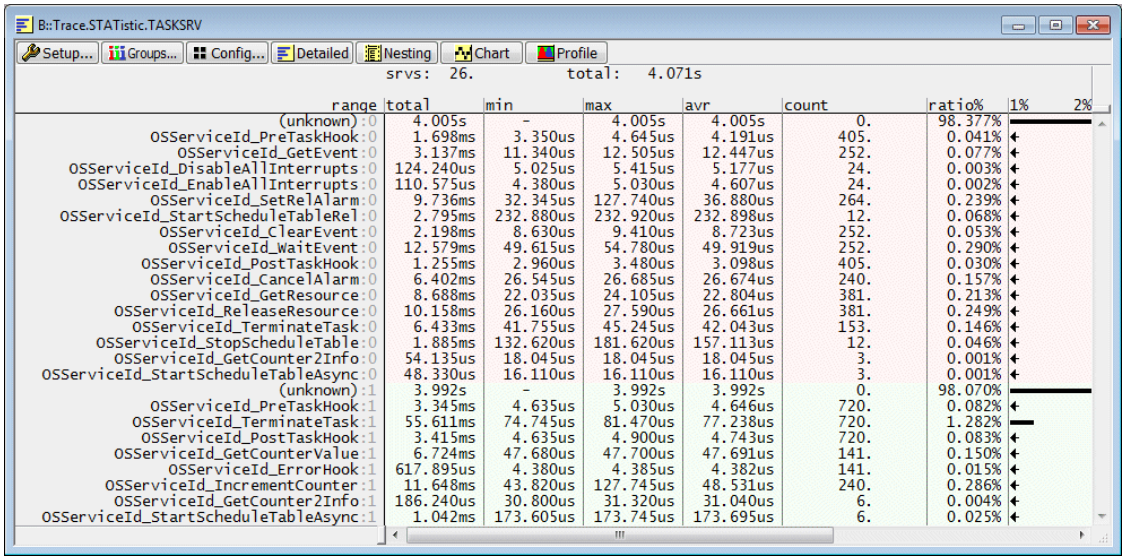


Trace.Chart.TASK /MergeCORE Task runtime time chart, results of all cores merged

The following commands perform a statistical analysis of the OSEK service routines:

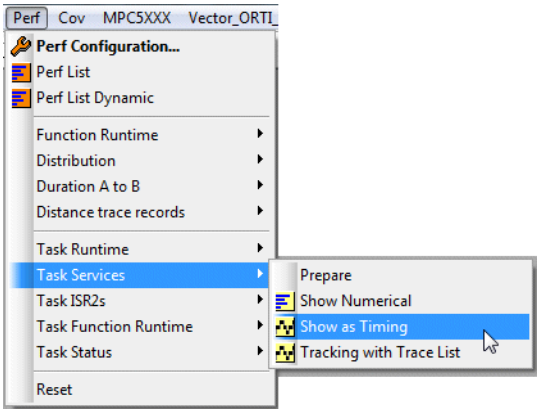


(unknown) represents the time in which the processor/core is not in an OSEK service routine



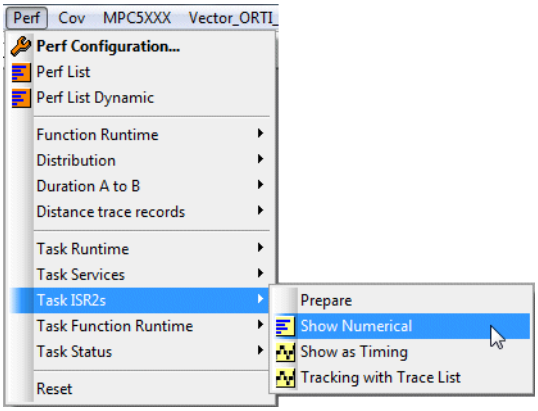
range	total	min	max	avr	count	ratio%	1% 2%
(unknown) :0	4.005s	-	4.005s	4.005s	0.	98.377%	
OSServiceId_PreTaskHook :0	1.698ms	3.350us	4.645us	4.191us	405.	0.041%	
OSServiceId_GetEvent :0	3.137ms	11.340us	12.505us	12.447us	252.	0.077%	
OSServiceId_DisableAllInterrupts :0	124.240us	5.025us	5.415us	5.177us	24.	0.003%	
OSServiceId_EnableAllInterrupts :0	110.575us	4.380us	5.030us	4.607us	24.	0.002%	
OSServiceId_SetRelAlarm :0	9.736ms	32.345us	127.740us	36.880us	264.	0.239%	
OSServiceId_StartScheduleTableRel :0	2.795ms	232.880us	232.920us	232.898us	12.	0.068%	
OSServiceId_ClearEvent :0	2.198ms	8.630us	9.410us	8.723us	252.	0.053%	
OSServiceId_WaitEvent :0	12.579ms	49.615us	54.780us	49.919us	252.	0.290%	
OSServiceId_PostTaskHook :0	1.255ms	2.960us	3.480us	3.098us	405.	0.030%	
OSServiceId_CancelAlarm :0	6.402ms	26.545us	26.685us	26.674us	240.	0.157%	
OSServiceId_GetResource :0	8.688ms	22.035us	24.105us	22.804us	381.	0.213%	
OSServiceId_ReleaseResource :0	10.158ms	26.160us	27.590us	26.661us	381.	0.249%	
OSServiceId_TerminateTask :0	6.433ms	41.755us	45.245us	42.043us	153.	0.146%	
OSServiceId_StopScheduleTable :0	1.885ms	132.620us	181.620us	157.113us	12.	0.046%	
OSServiceId_GetCounter2Info :0	54.135us	18.045us	18.045us	18.045us	3.	0.001%	
OSServiceId_StartScheduleTableAsync :0	48.330us	16.110us	16.110us	16.110us	3.	0.001%	
(unknown) :1	3.992s	-	3.992s	3.992s	0.	98.070%	
OSServiceId_PreTaskHook :1	3.345ms	4.635us	5.030us	4.646us	720.	0.082%	
OSServiceId_TerminateTask :1	55.611ms	74.745us	81.470us	77.238us	720.	1.282%	
OSServiceId_PostTaskHook :1	3.415ms	4.635us	4.900us	4.743us	720.	0.083%	
OSServiceId_GetCounterValue :1	6.724ms	47.680us	47.700us	47.691us	141.	0.150%	
OSServiceId_ErrorHook :1	617.895us	4.380us	4.385us	4.382us	141.	0.015%	
OSServiceId_IncrementCounter :1	11.648ms	43.820us	127.745us	48.531us	240.	0.286%	
OSServiceId_GetCounter2Info :1	186.240us	30.800us	31.320us	31.040us	6.	0.004%	
OSServiceId_StartScheduleTableAsync :1	1.042ms	173.605us	173.745us	173.695us	6.	0.025%	

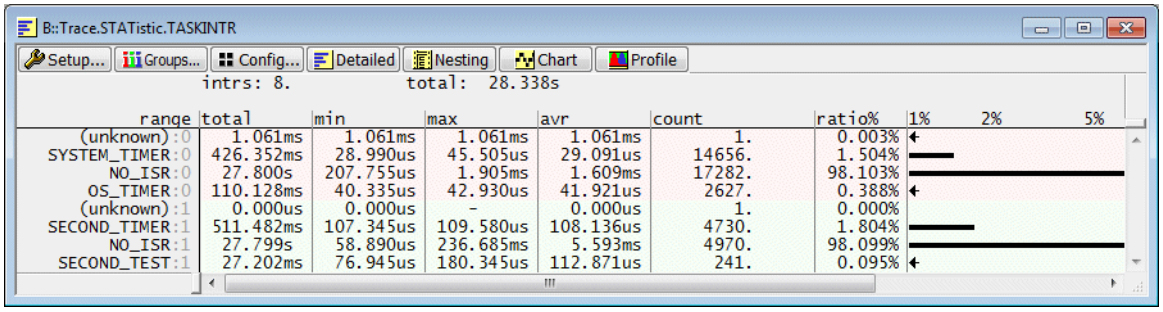
Trace.STATistic.TASKSRV [/SplitCORE] Statistic on service routines, result per core



Trace.Chart.TASKSRV [/SplitCORE] Time chart on service routines, result per core

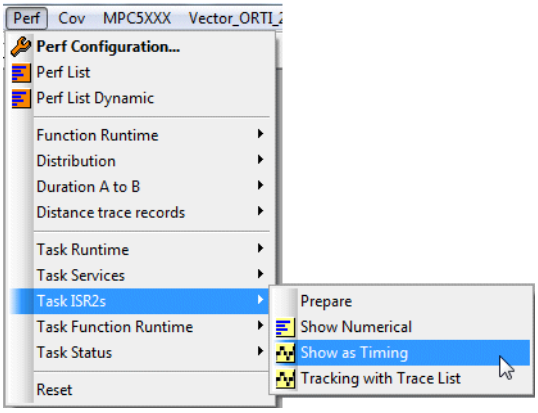
The following commands perform a statistical analysis of the OSEK interrupt service routines:

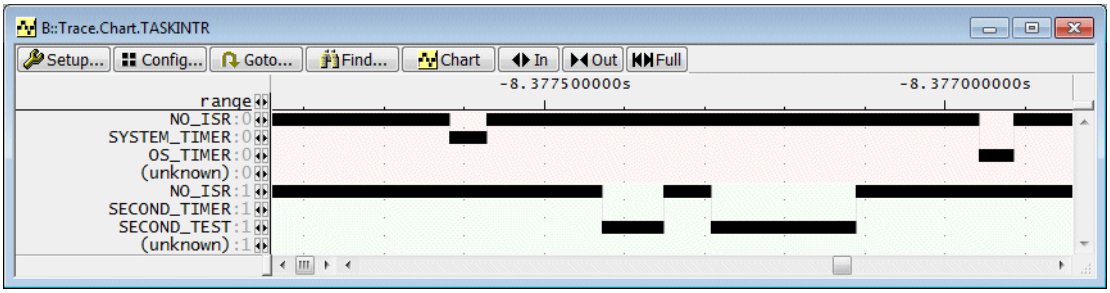




Trace.STATistic.TASKINTR [/SplitCORE] Statistic on interrupt service routines, result per core

TRACE32 assigns all trace information generated before the first **intr** information to **(unknown)**.

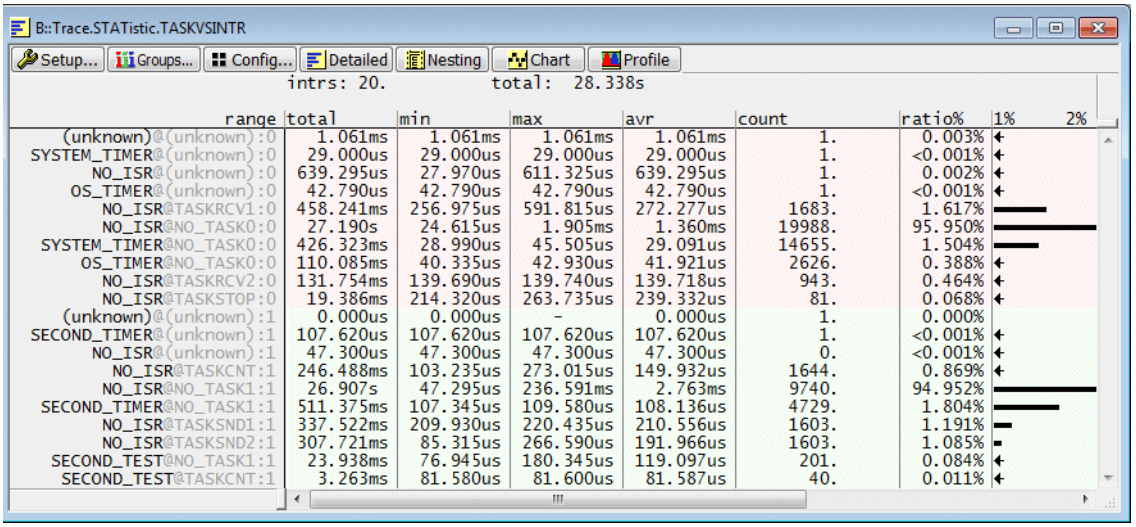




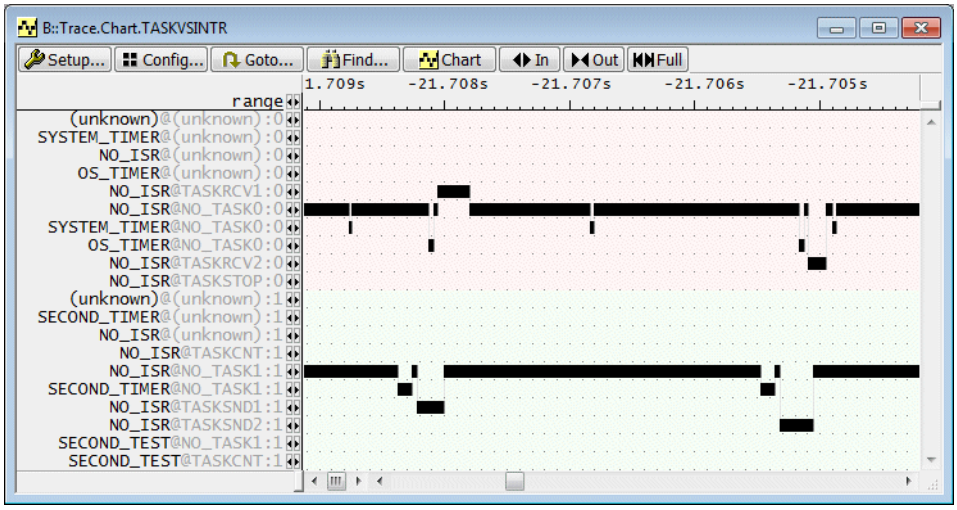
Trace.Chart.TASKINTR [/SplitCORE] Time chart on interrupt service routines, result per core

The following commands allow to perform a statistical analysis of the OSEK interrupt service routines related to the active tasks.

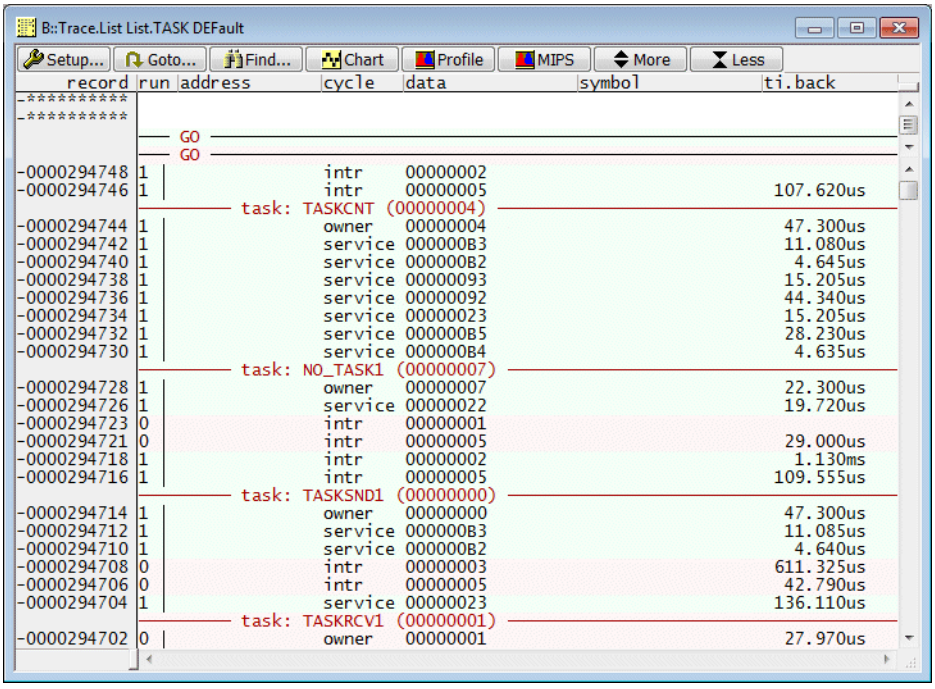
Trace.STATistic.TASKVSINTR [/SplitCORE]	Task-related statistic on interrupt service routines, result per core
Trace.Chart.TASKVSINTR [/SplitCORE]	Time-chart for task related interrupt service routines, result per core



range	total	min	max	avr	count	ratio%	1% 2%
(unknown)@(unknown):0	1.061ms	1.061ms	1.061ms	1.061ms	1.	0.003%	
SYSTEM_TIMER@(unknown):0	29.000us	29.000us	29.000us	29.000us	1.	<0.001%	
NO_ISR@(unknown):0	639.295us	27.970us	611.325us	639.295us	1.	0.002%	
OS_TIMER@(unknown):0	42.790us	42.790us	42.790us	42.790us	1.	<0.001%	
NO_ISR@TASKRCV1:0	458.241ms	256.975us	591.815us	272.277us	1683.	1.617%	
NO_ISR@NO_TASK0:0	27.190s	24.615us	1.905ms	1.360ms	19988.	95.950%	
SYSTEM_TIMER@NO_TASK0:0	426.323ms	28.990us	45.505us	29.091us	14655.	1.504%	
OS_TIMER@NO_TASK0:0	110.085ms	40.335us	42.930us	41.921us	2626.	0.388%	
NO_ISR@TASKRCV2:0	131.754ms	139.690us	139.740us	139.718us	943.	0.464%	
NO_ISR@TASKSTOP:0	19.386ms	214.320us	263.735us	239.332us	81.	0.068%	
(unknown)@(unknown):1	0.000us	0.000us	-	0.000us	1.	0.000%	
SECOND_TIMER@(unknown):1	107.620us	107.620us	107.620us	107.620us	1.	<0.001%	
NO_ISR@(unknown):1	47.300us	47.300us	47.300us	47.300us	0.	<0.001%	
NO_ISR@TASKCNT:1	246.488ms	103.235us	273.015us	149.932us	1644.	0.869%	
NO_ISR@NO_TASK1:1	26.907s	47.295us	236.591ms	2.763ms	9740.	94.952%	
SECOND_TIMER@NO_TASK1:1	511.375ms	107.345us	109.580us	108.136us	4729.	1.804%	
NO_ISR@TASKSND1:1	337.522ms	209.930us	220.435us	210.556us	1603.	1.191%	
NO_ISR@TASKSND2:1	307.721ms	85.315us	266.590us	191.966us	1603.	1.085%	
SECOND_TEST@NO_TASK1:1	23.938ms	76.945us	180.345us	119.097us	201.	0.084%	
SECOND_TEST@TASKCNT:1	3.263ms	81.580us	81.600us	81.587us	40.	0.011%	



intr information that was generated before the first **task** information is assigned to the **@(unknown)** task.



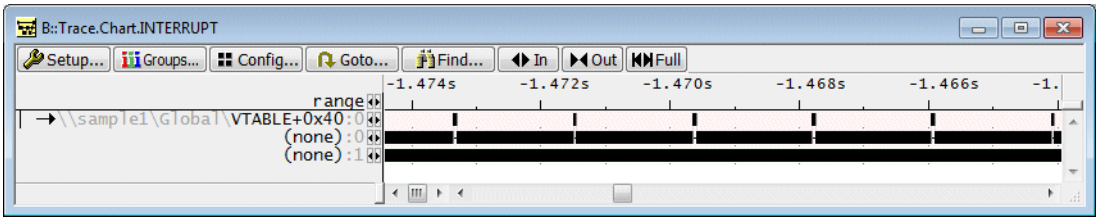
Exporting all Types of Task Information and all Instructions (OTM)

General setup:

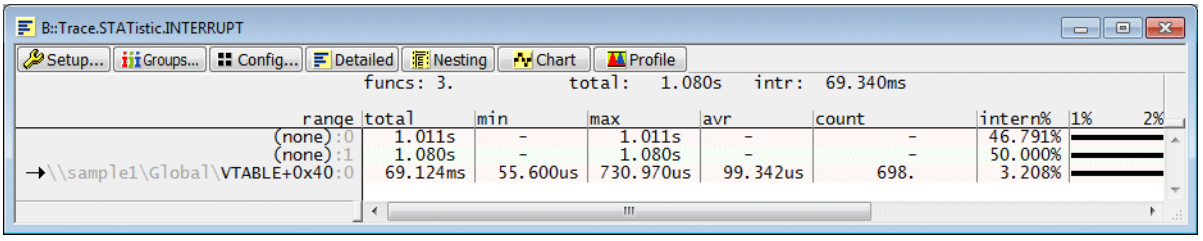
```
NEXUS.BTM ON                                  ; enable the Branch Trace
                                              ; Messages

NEXUS.OTM ON                                  ; enable the Ownership Trace
                                              ; Messages

Trace.STATistic.InterruptIsFunction ON        ; advise TRACE32 to regard the
                                              ; time between interrupt entry
                                              ; and exit as function
```

Statistic Analysis of Interrupts

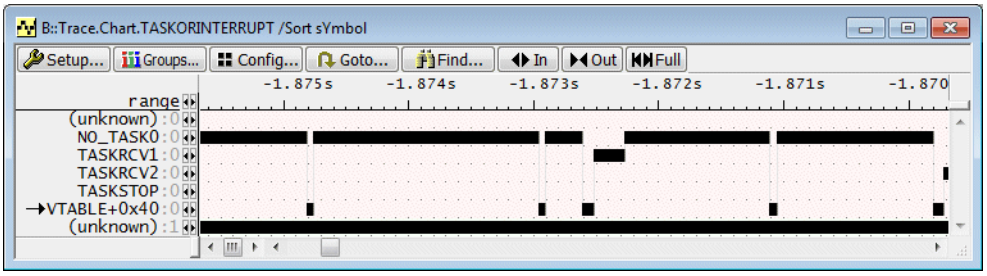
Trace.Chart.INTERRUPT [**/SplitCORE**]	Interrupt time chart (default), results split up per core
Trace.Chart.INTERRUPT /CORE *<n>*	Interrupt time chart for specified core



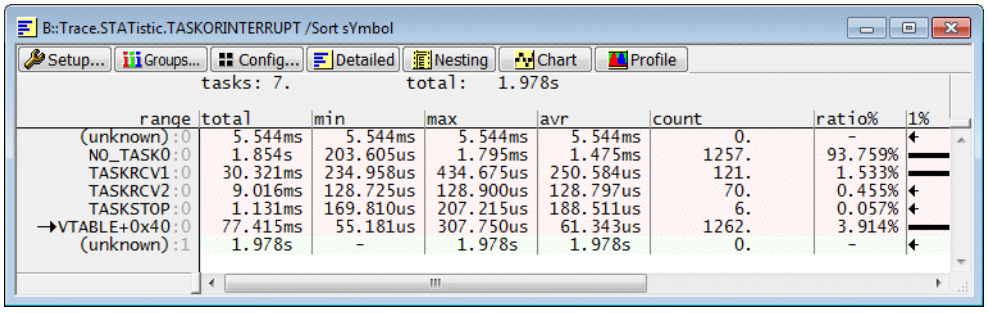
Trace.STATistic.INTERRUPT [**/SplitCORE**]	Interrupt statistic (default), results split up per core
Trace.STATistic.INTERRUPT /CORE *<n>*	Interrupt statistic for specified core



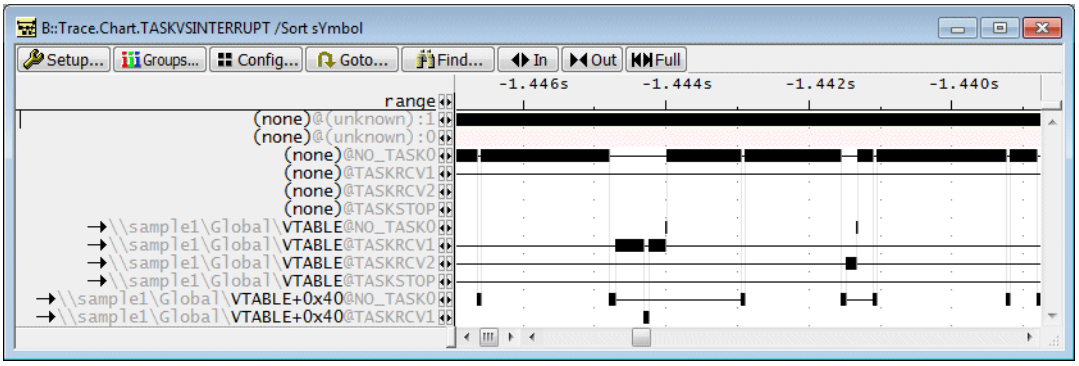
Trace.Chart.TASKORINTERRUPT [**/SplitCORE**]	Time chart for interrupts and tasks (default), results split up per core
Trace.Chart.TASKORINTERRUPT /CORE *<n>*	Time chart for interrupts and tasks for specified core



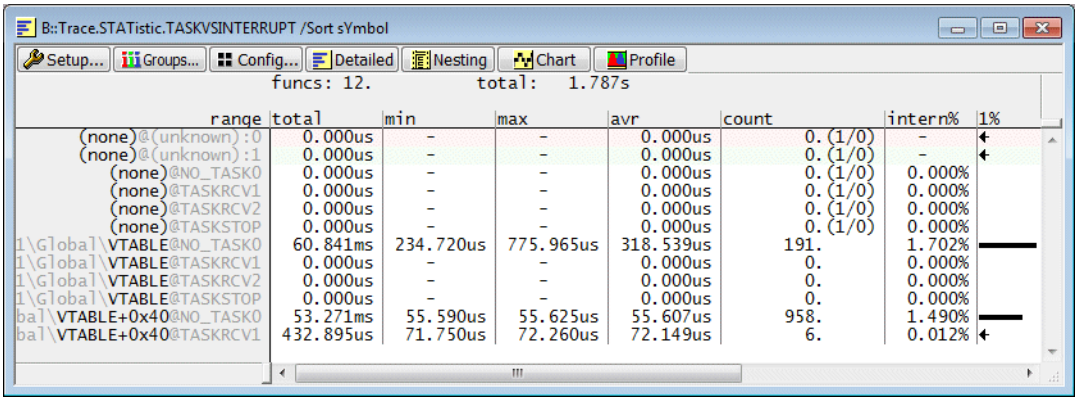
Trace.STATistic.TASKORINTERRUPT [**/SplitCORE**]	Statistic for interrupts and tasks (default), results split up per core
Trace.STATistic.TASKORINTERRUPT /CORE *<n>*	Statistic for interrupts and tasks for specified core



Trace.Chart.TASKVSINTERRUPT [**/SplitCORE**]	Interrupt time chart, task-related (default), results split up per core
Trace.Chart.TASKVSINTERRUPT /CORE *<n>*	Interrupt time chart task-related, for specified core



Trace.STATistic.TASKVSINTERRUPT [**/SplitCORE**]	Interrupt statistic, task-related (default), results split up per core
Trace.STATistic.TASKVSINTERRUPT /CORE *<n>*	Interrupt statistic, task-related, for specified core



Task Switches

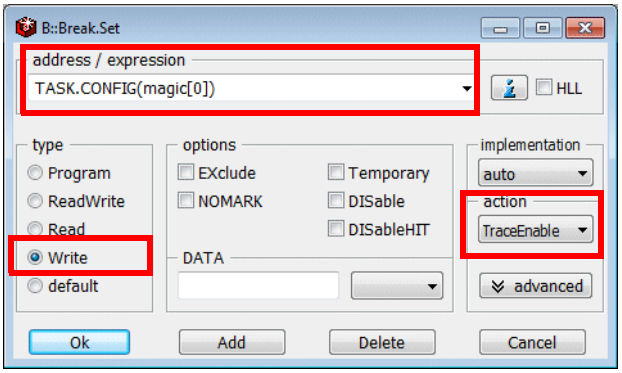
An SMP operating system has **one variable per core** that contains the information which task is currently running. One way to export task switch information is to advise the NEXUS hardware module to generate trace information when a write access to one of these variables occurs.

The address of these variables is provided by the TRACE32 functions **TASK.CONFIG(magic[**<*core*>**])**.

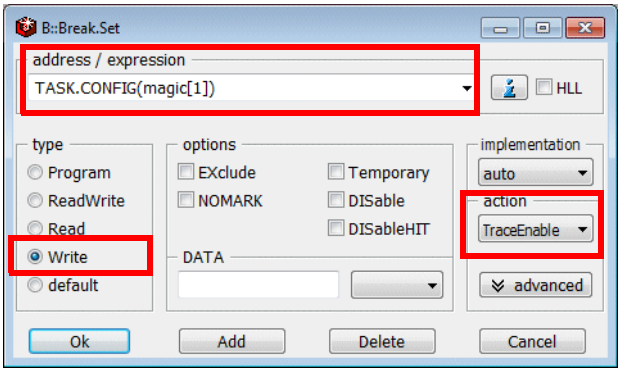
```
PRINT TASK.CONFIG(magic[0])           ; print the address of the variable
                                      ; that holds the task identifier
                                      ; for core 0

PRINT TASK.CONFIG(magic[1])           ; print the address of the variable
                                      ; that holds the task identifier
                                      ; for core 1

…

PRINT TASK.CONFIG(magic[n])           ; print the address of the variable
                                      ; that holds the task identifier
                                      ; for core n
```

Example: Advise the NEXUS hardware module to generate only trace information on task switches for a dual-core chip.

1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic[0]) and select the trace action TraceEnable.



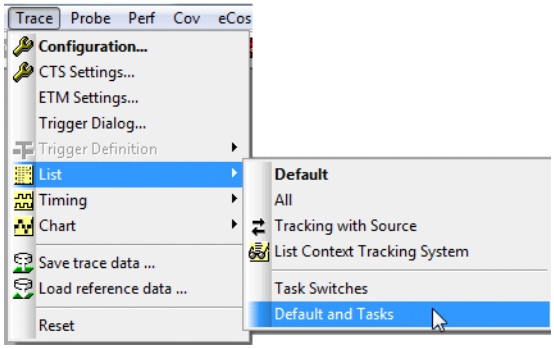
2. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic[1]) and select the trace action TraceEnable.

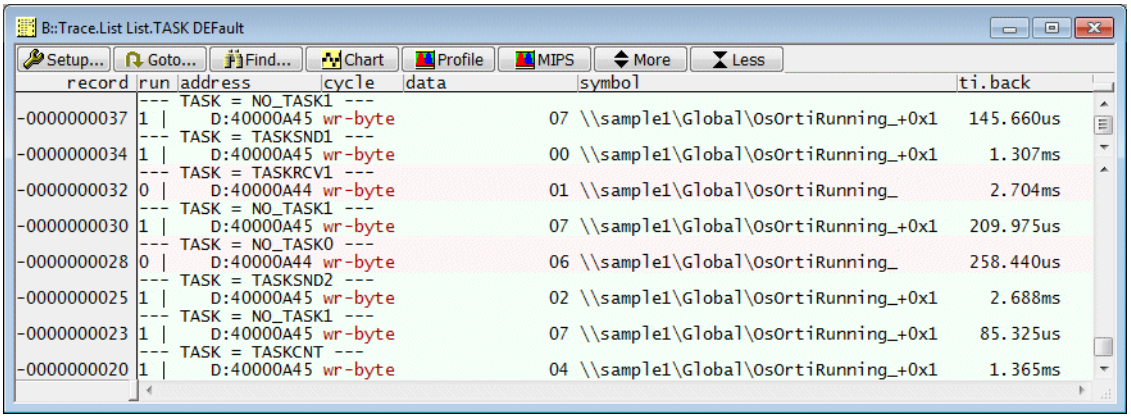



```
Break.Set TASK.CONFIG(magic[0]) /Write /TraceEnable

Break.Set TASK.CONFIG(magic[1]) /Write /TraceEnable
```

3. Start and stop the program execution to fill the trace buffer

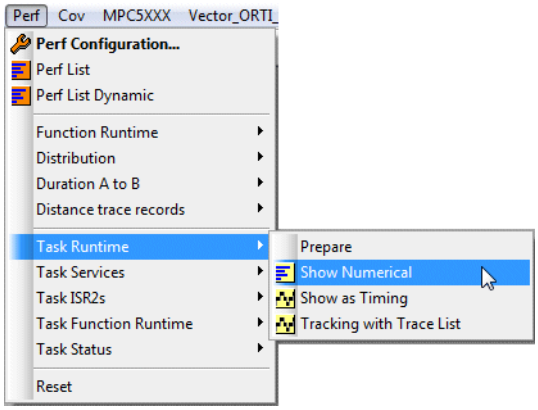
4. Display the result.

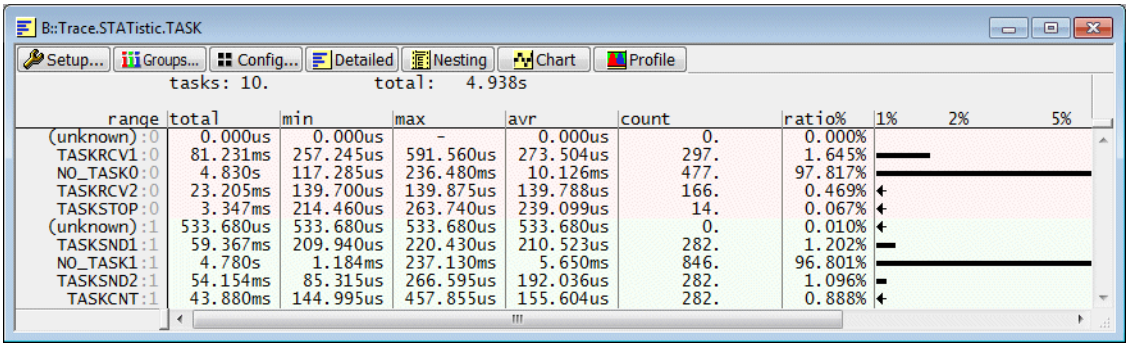




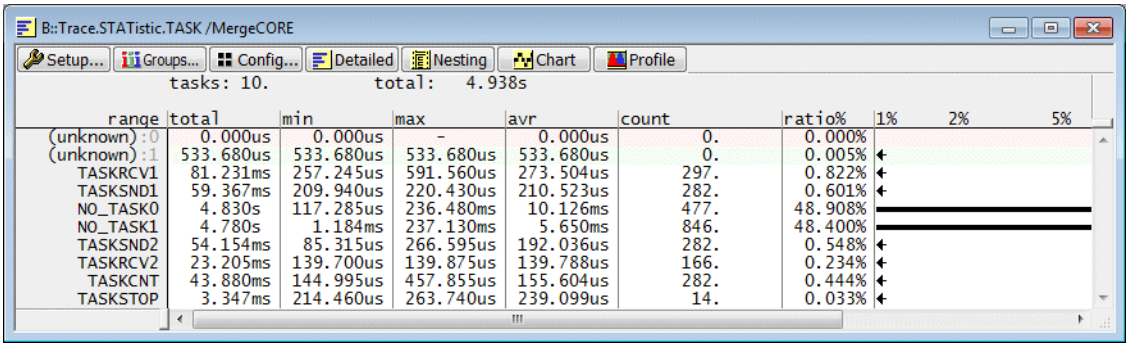
record	run	address	cycle	data	symbol	ti.back
	--- TASK = NO_TASK1 ---					
-0000000037	1 |	D:40000A45 wr-byte		07	\\sample1\Global\OsOrtiRunning_+0x1	145.660us
	--- TASK = TASKSND1 ---					
-0000000034	1 |	D:40000A45 wr-byte		00	\\sample1\Global\OsOrtiRunning_+0x1	1.307ms
	--- TASK = TASKRCV1 ---					
-0000000032	0 |	D:40000A44 wr-byte		01	\\sample1\Global\OsOrtiRunning_	2.704ms
	--- TASK = NO_TASK1 ---					
-0000000030	1 |	D:40000A45 wr-byte		07	\\sample1\Global\OsOrtiRunning_+0x1	209.975us
	--- TASK = NO_TASK0 ---					
-0000000028	0 |	D:40000A44 wr-byte		06	\\sample1\Global\OsOrtiRunning_	258.440us
	--- TASK = TASKSND2 ---					
-0000000025	1 |	D:40000A45 wr-byte		02	\\sample1\Global\OsOrtiRunning_+0x1	2.688ms
	--- TASK = NO_TASK1 ---					
-0000000023	1 |	D:40000A45 wr-byte		07	\\sample1\Global\OsOrtiRunning_+0x1	85.325us
	--- TASK = TASKCNT ---					
-0000000020	1 |	D:40000A45 wr-byte		04	\\sample1\Global\OsOrtiRunning_+0x1	1.365ms

The following commands perform a statistical analysis of the task switches:



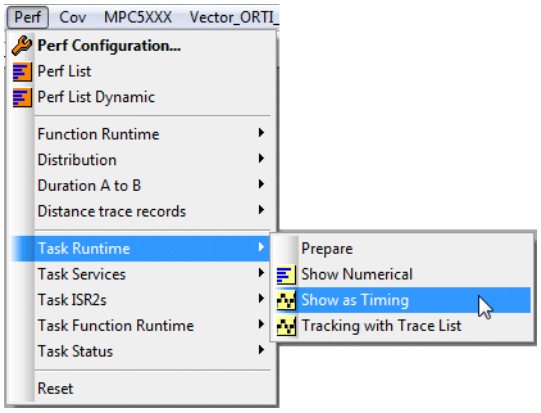


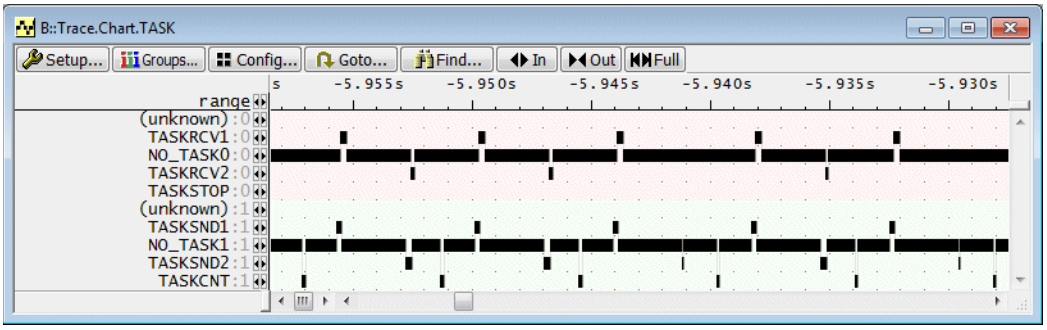
Trace.STATistic.TASK [/SplitCORE] Task runtime statistic, result per core



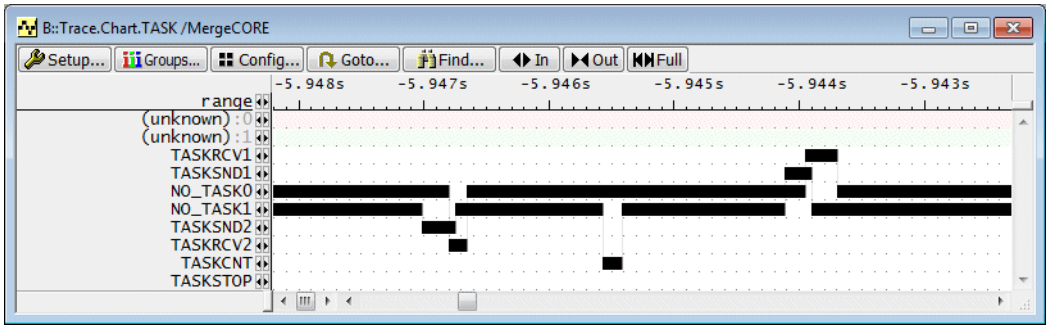
Trace.STATistic.TASK /MergeCORE Task runtime statistic, results of all cores merged

The following commands display a time-chart of the task run-times:





Trace.Chart.TASK [/SplitCORE] Task runtime time chart, result per core



Trace.Chart.TASK /MergeCORE Task runtime time chart, results of all cores merged

The time spent in OSEK service routines can be evaluated.

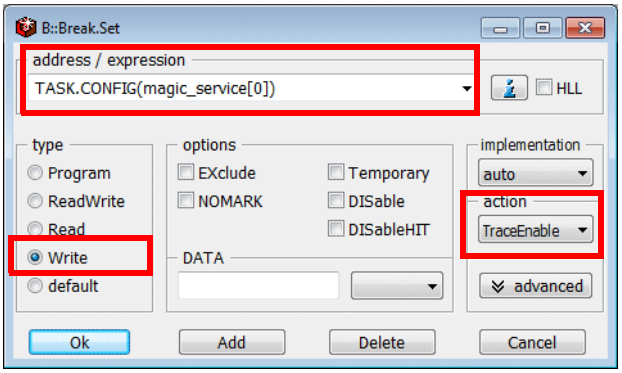
OSEK writes information on the entries and exits to OSEK service routines to a defined variable per core. One way to export information on OSEK service routines is to advise the NEXUS hardware module to generate trace information when a write access to one of these variables occurs.

The address of these variables is provided by the TRACE32 functions **TASK.CONFIG(magic_service[**<*core*>**])**.

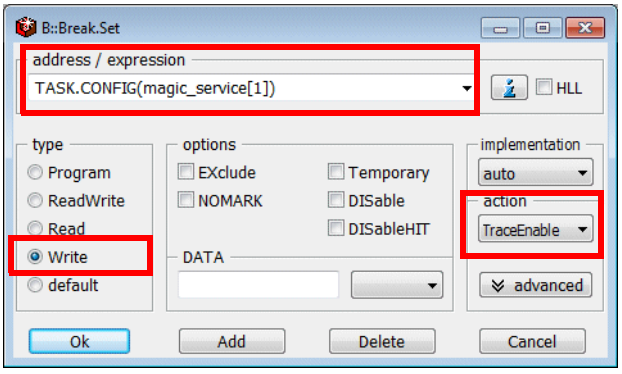
```
PRINT TASK.CONFIG(magic_service[0])      ; print the address of the
                                         ; variable that holds the
                                         ; service information for core 0

PRINT TASK.CONFIG(magic_service[1])      ; print the address of the
                                         ; variable that holds the
                                         ; service information for core 1

…

PRINT TASK.CONFIG(magic_service[n])      ; print the address of the
                                         ; variable that holds the
                                         ; service information for core n
```

Example: Advise the NEXUS hardware module to generate only trace information for entries and exits to OSEK service routines for a dual-core chip.

1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic_service[0]) and select the trace action TraceEnable.



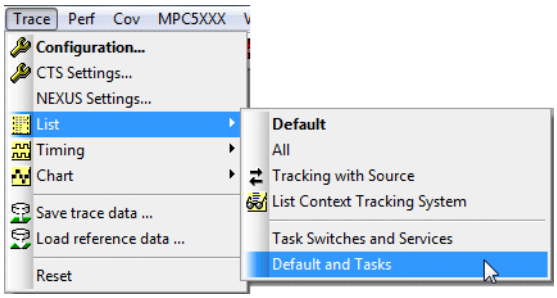
2. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic_service[1]) and select the trace action TraceEnable.

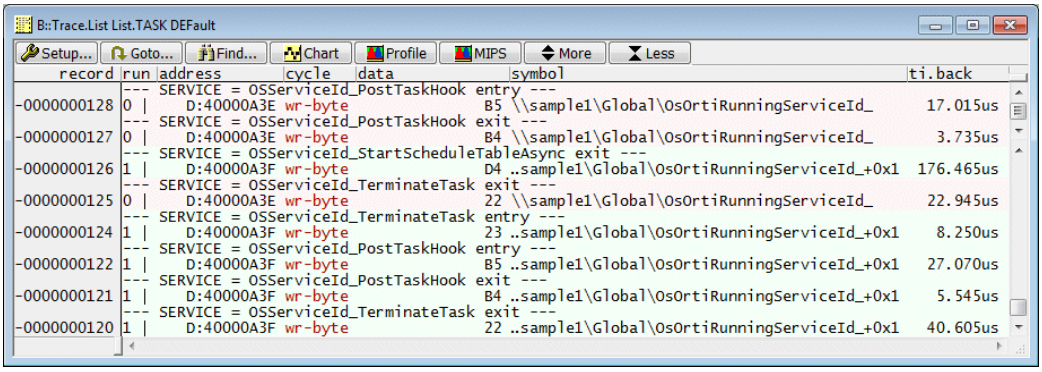



```
Break.Set TASK.CONFIG(magic_service[0]) /Write /TraceEnable

Break.Set TASK.CONFIG(magic_service[1]) /Write /TraceEnable
```

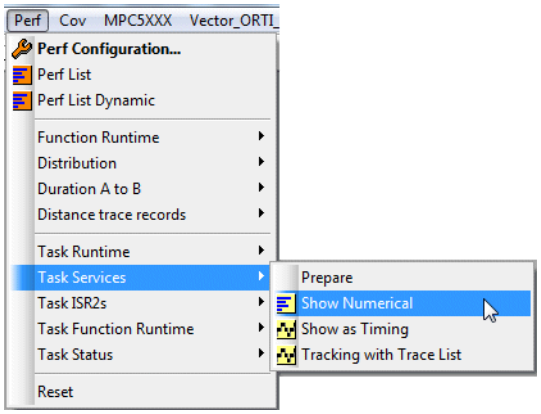
3. Start and stop the program execution to fill the trace buffer

4. Display the result.

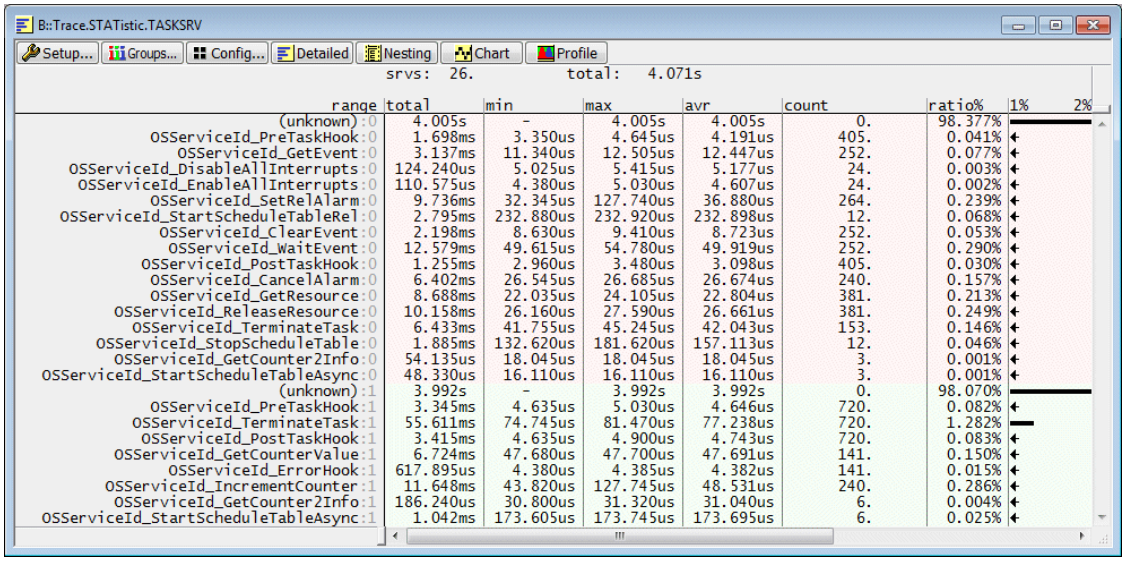




The following two commands perform a statistical analysis of the OSEK service routines:

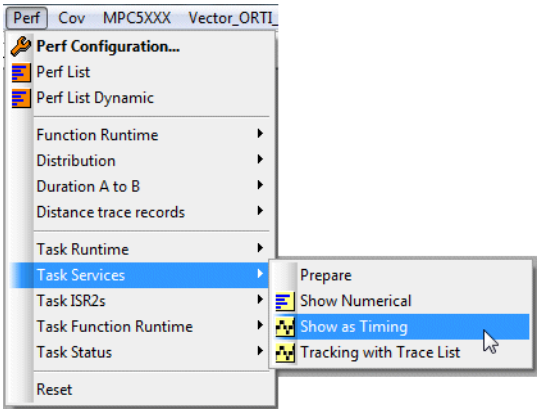


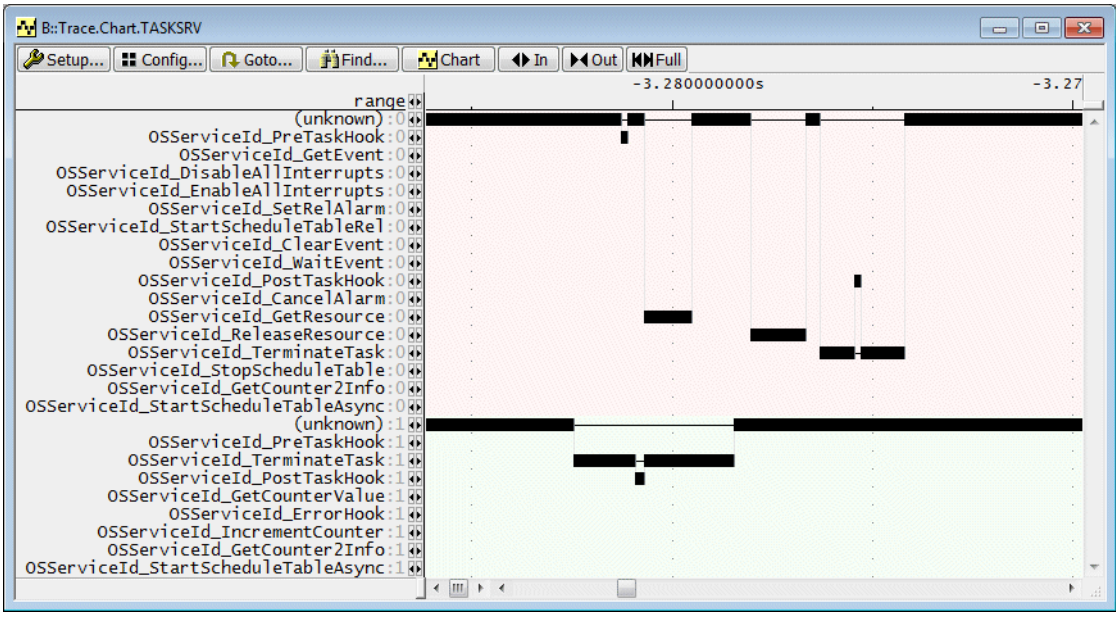
(unknown) represents the time in which the processor/core is not in an OSEK service routine



range	total	min	max	avr	count	ratio%	1% 2%
(unknown):0	4.005s	-	4.005s	4.005s	0.	98.377%	
OSServiceId_PreTaskHook:0	1.698ms	3.350us	4.645us	4.191us	405.	0.041%	
OSServiceId_GetEvent:0	3.137ms	11.340us	12.505us	12.447us	252.	0.077%	
OSServiceId_DisableAllInterrupts:0	124.240us	5.025us	5.415us	5.177us	24.	0.003%	
OSServiceId_EnableAllInterrupts:0	110.575us	4.380us	5.030us	4.607us	24.	0.002%	
OSServiceId_SetRelAlarm:0	9.736ms	32.345us	127.740us	36.880us	264.	0.239%	
OSServiceId_StartScheduleTableRel:0	2.795ms	232.880us	232.920us	232.898us	12.	0.068%	
OSServiceId_ClearEvent:0	2.198ms	8.630us	9.410us	8.723us	252.	0.053%	
OSServiceId_WaitEvent:0	12.579ms	49.615us	54.780us	49.919us	252.	0.290%	
OSServiceId_PostTaskHook:0	1.255ms	2.960us	3.480us	3.098us	405.	0.030%	
OSServiceId_CancelAlarm:0	6.402ms	26.545us	26.685us	26.674us	240.	0.157%	
OSServiceId_GetResource:0	8.688ms	22.035us	24.105us	22.804us	381.	0.213%	
OSServiceId_ReleaseResource:0	10.158ms	26.160us	27.590us	26.661us	381.	0.249%	
OSServiceId_TerminateTask:0	6.433ms	41.755us	45.245us	42.043us	153.	0.146%	
OSServiceId_StopScheduleTable:0	1.885ms	132.620us	181.620us	157.113us	12.	0.046%	
OSServiceId_GetCounter2Info:0	54.135us	18.045us	18.045us	18.045us	3.	0.001%	
OSServiceId_StartScheduleTableAsync:0	48.330us	16.110us	16.110us	16.110us	3.	0.001%	
(unknown):1	3.992s	-	3.992s	3.992s	0.	98.070%	
OSServiceId_PreTaskHook:1	3.345ms	4.635us	5.030us	4.646us	720.	0.082%	
OSServiceId_TerminateTask:1	55.611ms	74.745us	81.470us	77.238us	720.	1.282%	
OSServiceId_PostTaskHook:1	3.415ms	4.635us	4.900us	4.743us	720.	0.083%	
OSServiceId_GetCounterValue:1	6.724ms	47.680us	47.700us	47.691us	141.	0.150%	
OSServiceId_ErrorHook:1	617.895us	4.380us	4.385us	4.382us	141.	0.015%	
OSServiceId_IncrementCounter:1	11.648ms	43.820us	127.745us	48.531us	240.	0.286%	
OSServiceId_GetCounter2Info:1	186.240us	30.800us	31.320us	31.040us	6.	0.004%	
OSServiceId_StartScheduleTableAsync:1	1.042ms	173.605us	173.745us	173.695us	6.	0.025%	

Trace.STATistic.TASKSRV [/SplitCORE] Statistic on service routines, result per core





Trace.Chart.TASKSRV [/SplitCORE] Time chart on service routines, result per core

The time spent in OSEK interrupt service routines can be evaluated.

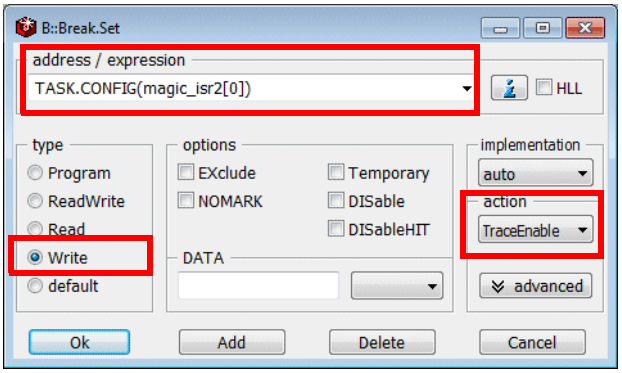
OSEK writes information on the start of an interrupt service routine to a defined variable per core as well as the information NO_ISR. One way to export information on OSEK interrupt service routines is to advise the NEXUS hardware module to generate trace information when a write access to these variables occurs.

The address of these variables is provided by the TRACE32 functions **TASK.CONFIG(magic_isr2[**<*core*>**])**.

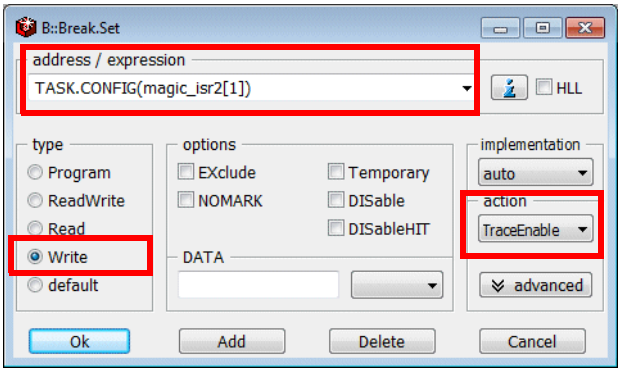
```
PRINT TASK.CONFIG(magic_isr2[0])      ; print the address of the variable
                                      ; that holds the interrupt service
                                      ; information for core 0

PRINT TASK.CONFIG(magic_isr2[1])      ; print the address of the variable
                                      ; that holds the interrupt service
                                      ; information for core 1

…

PRINT TASK.CONFIG(magic_isr2[n])      ; print the address of the variable
                                      ; that holds the interrupt service
                                      ; information for core n
```

Example: Advise the NEXUS hardware module to generate only trace information on the start of an interrupt service routine as well as on the information NO_ISR for a dual-core chip.

1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic_isr2[0]) and select the trace action TraceEnable.



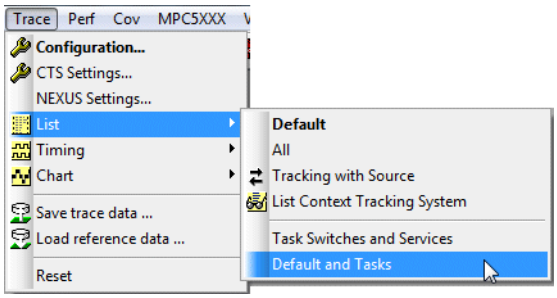
2. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic_isr2[1]) and select the trace action TraceEnable.

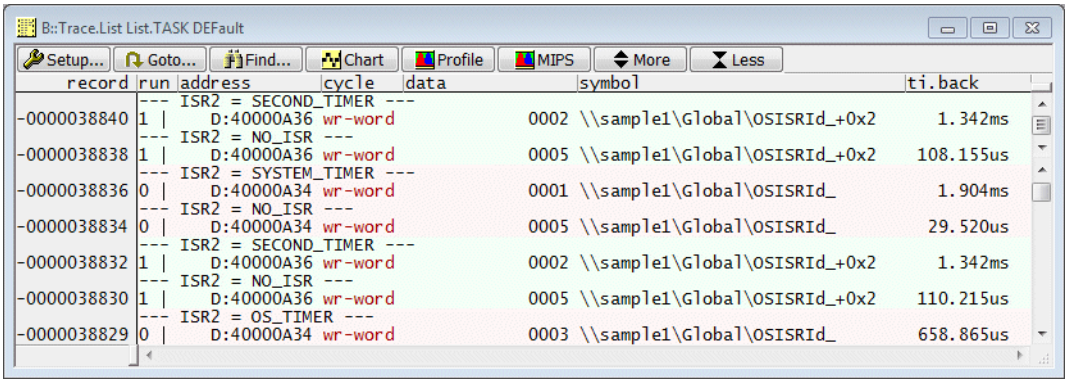



```
Break.Set TASK.CONFIG(magic_isr2[0]) /Write /TraceEnable

Break.Set TASK.CONFIG(magic_isr2[1]) /Write /TraceEnable
```

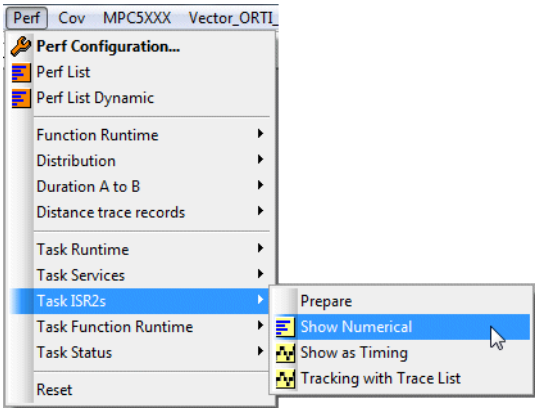
3. Start and stop the program execution to fill the trace buffer

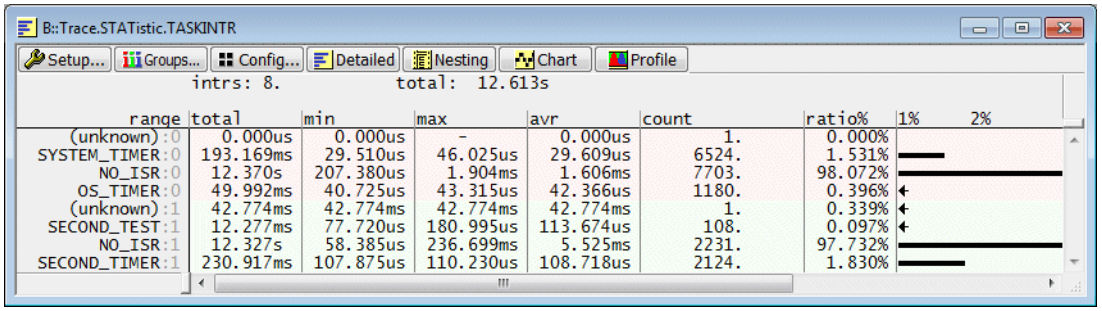
4. Display the result.



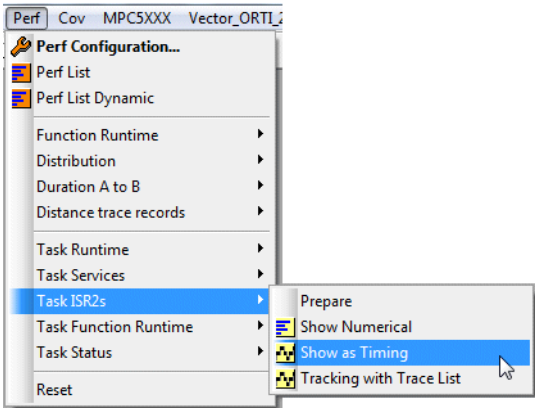


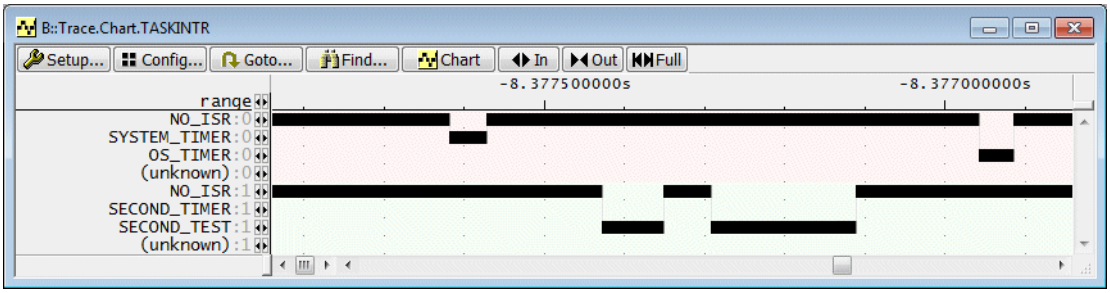
The following commands perform a statistical analysis of the OSEK interrupt service routines:





Trace.STATistic.TASKINTR [/SplitCORE] Statistic on interrupt service routines, result per core





Trace.Chart.TASKINTR [/SplitCORE] Time chart on interrupt service routines, result per core

OSEK interrupt service routines that occur in multiple tasks can be displayed per task, if the following information is available:

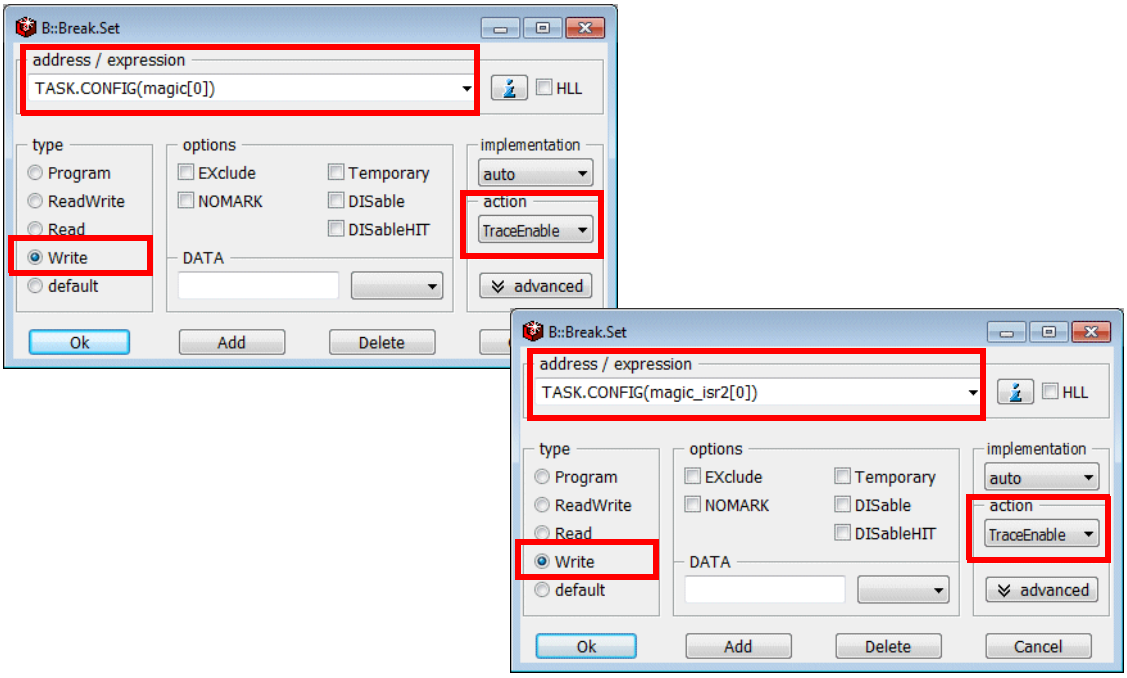
- Task switch information per core
- ISR2 start and NO_ISR information per core

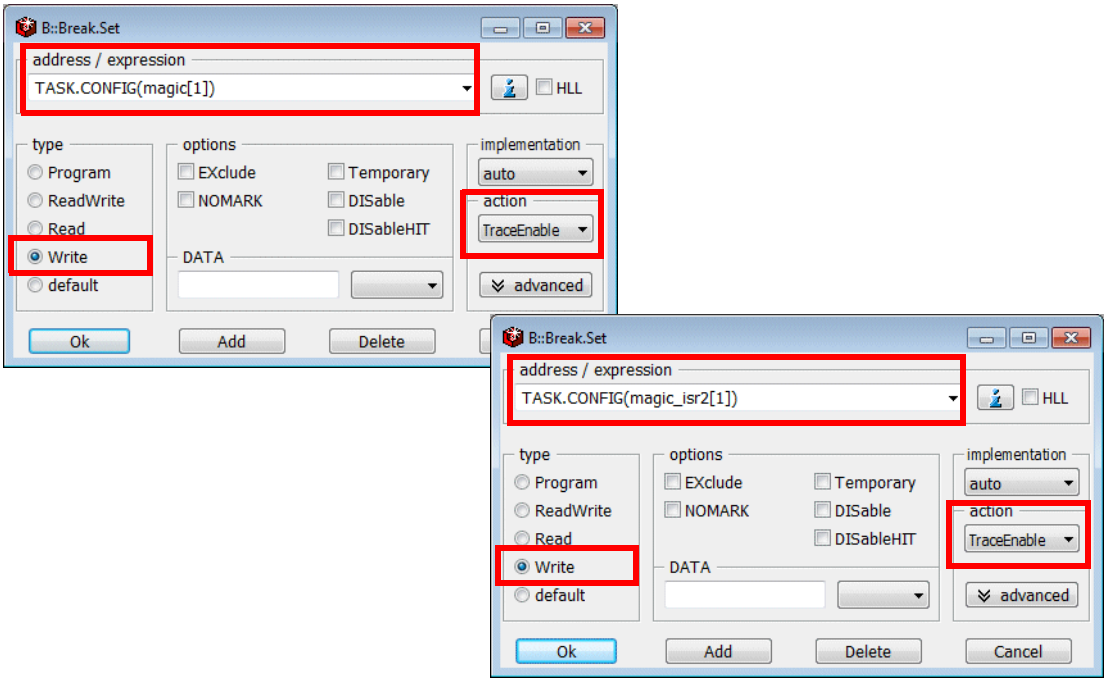
Example:

1. Advise the NEXUS hardware module to generate the following trace information for a dual-core chip:

   - task switches per core

   - start of an interrupt service routine as well as on the information NO_ISR per core.

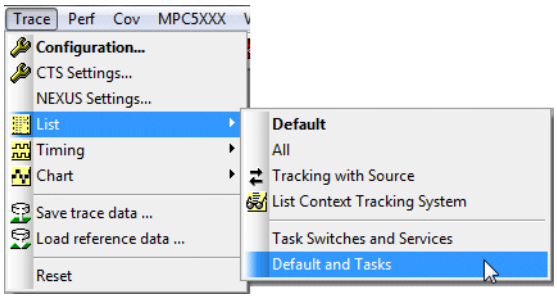


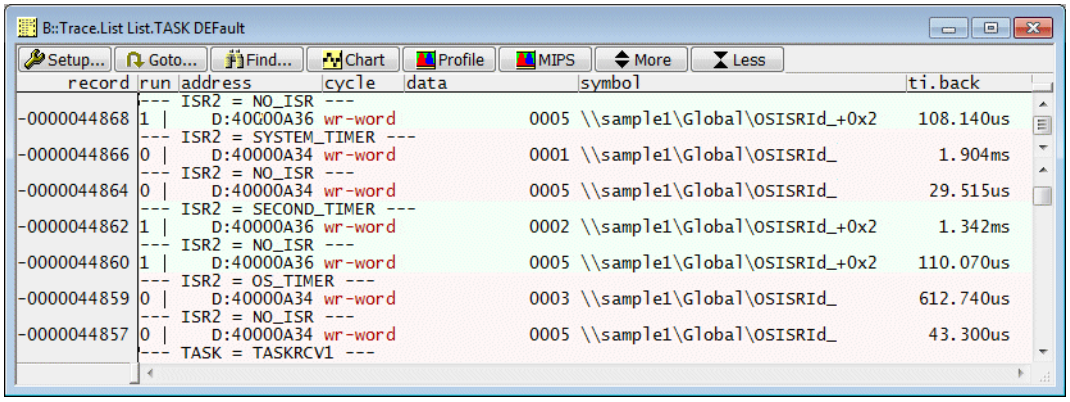



```
Break.Set TASK.CONFIG(magic[0]) /Write /TraceEnable

Break.Set TASK.CONFIG(magic_isr2[0]) /Write /TraceEnable

Break.Set TASK.CONFIG(magic[1]) /Write /TraceEnable

Break.Set TASK.CONFIG(magic_isr2[1]) /Write /TraceEnable
```

2. Start and stop the program execution to fill the trace buffer.

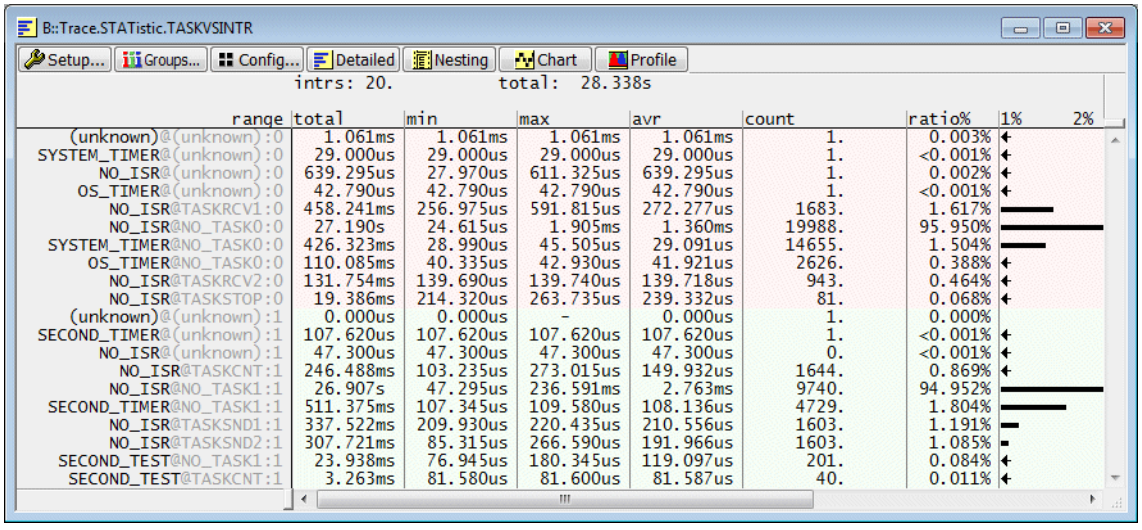
3. Display the result.



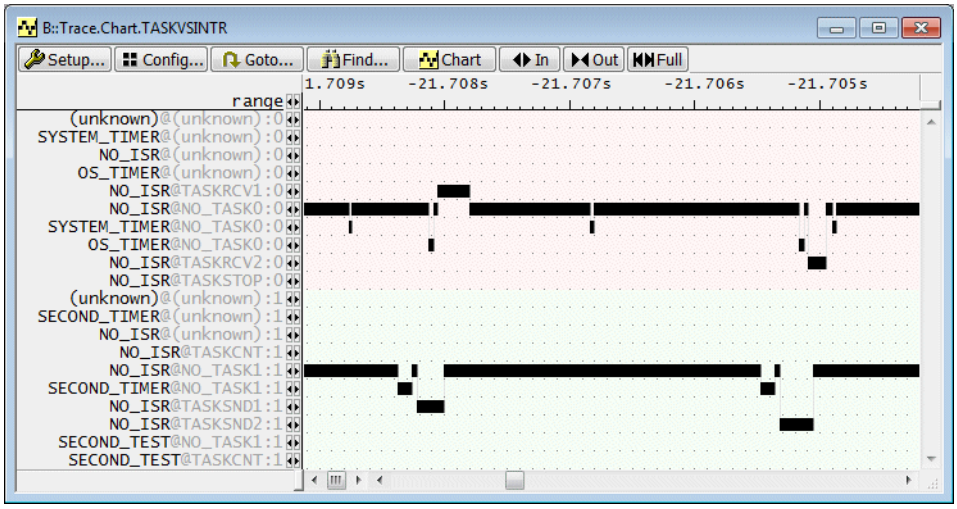


The following commands allow to perform a statistical analysis of the OSEK interrupt service routines related to the active tasks.

Trace.STATistic.TASKVSINTR [/SplitCORE]	Task-related statistic on interrupt service routines, result per core
Trace.Chart.TASKVSINTR [/SplitCORE]	Time-chart for task related interrupt service routines, result per core



range	total	min	max	avr	count	ratio%	1% 2%
(unknown)@(unknown):0	1.061ms	1.061ms	1.061ms	1.061ms	1.	0.003%	
SYSTEM_TIMER@(unknown):0	29.000us	29.000us	29.000us	29.000us	1.	<0.001%	
NO_ISR@(unknown):0	639.295us	27.970us	611.325us	639.295us	1.	0.002%	
OS_TIMER@(unknown):0	42.790us	42.790us	42.790us	42.790us	1.	<0.001%	
NO_ISR@TASKRCV1:0	458.241ms	256.975us	591.815us	272.277us	1683.	1.617%	
NO_ISR@NO_TASK0:0	27.190s	24.615us	1.905ms	1.360ms	19988.	95.950%	
SYSTEM_TIMER@NO_TASK0:0	426.323ms	28.990us	45.505us	29.091us	14655.	1.504%	
OS_TIMER@NO_TASK0:0	110.085ms	40.335us	42.930us	41.921us	2626.	0.388%	
NO_ISR@TASKRCV2:0	131.754ms	139.690us	139.740us	139.718us	943.	0.464%	
NO_ISR@TASKSTOP:0	19.386ms	214.320us	263.735us	239.332us	81.	0.068%	
(unknown)@(unknown):1	0.000us	0.000us	-	0.000us	1.	0.000%	
SECOND_TIMER@(unknown):1	107.620us	107.620us	107.620us	107.620us	1.	<0.001%	
NO_ISR@(unknown):1	47.300us	47.300us	47.300us	47.300us	0.	<0.001%	
NO_ISR@TASKCNT:1	246.488ms	103.235us	273.015us	149.932us	1644.	0.869%	
NO_ISR@NO_TASK1:1	26.907s	47.295us	236.591ms	2.763ms	9740.	94.952%	
SECOND_TIMER@NO_TASK1:1	511.375ms	107.345us	109.580us	108.136us	4729.	1.804%	
NO_ISR@TASKSND1:1	337.522ms	209.930us	220.435us	210.556us	1603.	1.191%	
NO_ISR@TASKSND2:1	307.721ms	85.315us	266.590us	191.966us	1603.	1.085%	
SECOND_TEST@NO_TASK1:1	23.938ms	76.945us	180.345us	119.097us	201.	0.084%	
SECOND_TEST@TASKCNT:1	3.263ms	81.580us	81.600us	81.587us	40.	0.011%	



intr information that was generated before the first **task** information is assigned to the **@(unknown)** task.

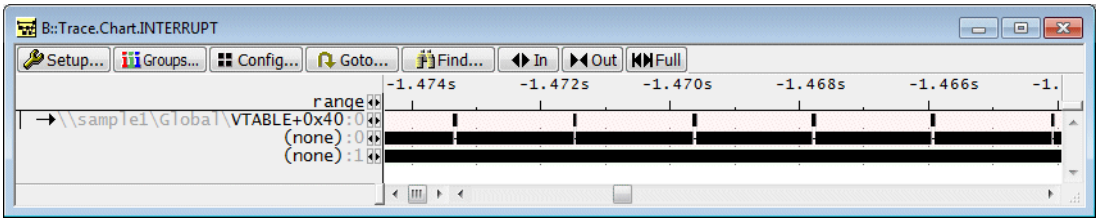
Exporting Task Switches and all Instructions (Write Access)

General setup:

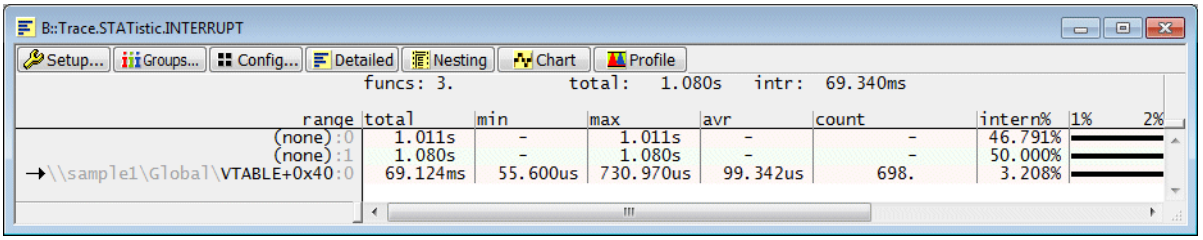
```
Break.Set TASK.CONFIG(magic[0]) /Write /TraceData

Break.Set TASK.CONFIG(magic[1]) /Write /TraceData

…

; advise TRACE32 to regard the time between interrupt entry
; and exit as function
Trace.STATistic.InterruptIsFunction ON
```

Statistic Analysis of Interrupts

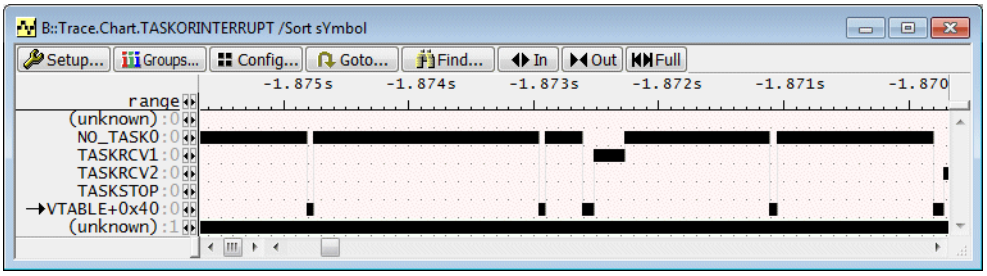
Trace.Chart.INTERRUPT [**/SplitCORE**]	Interrupt time chart (default), results split up per core
Trace.Chart.INTERRUPT /CORE *<n>*	Interrupt time chart for specified core



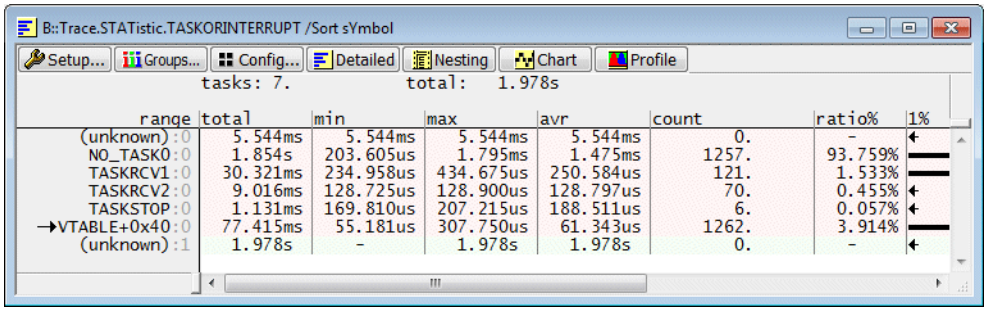
Trace.STATistic.INTERRUPT [**/SplitCORE**]	Interrupt statistic (default), results split up per core
Trace.STATistic.INTERRUPT /CORE *<n>*	Interrupt statistic for specified core



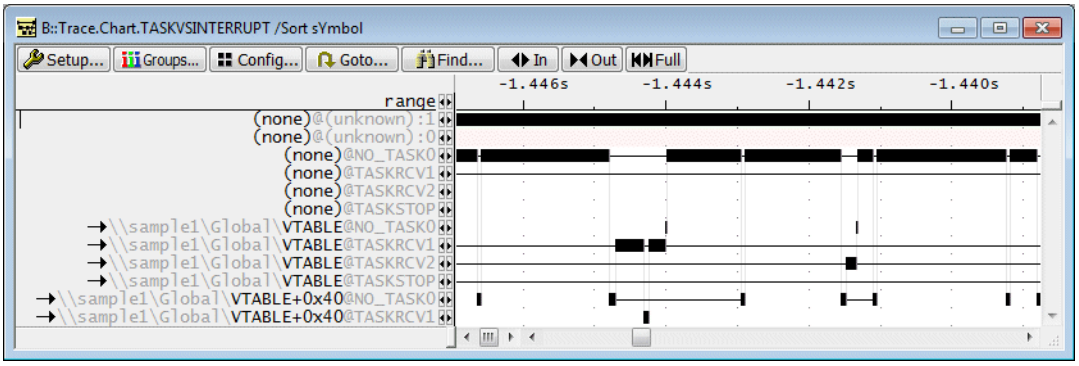
Trace.Chart.TASKORINTERRUPT [**/SplitCORE**]	Time chart for interrupts and tasks (default), results split up per core
Trace.Chart.TASKORINTERRUPT /CORE *<n>*	Time chart for interrupts and tasks for specified core



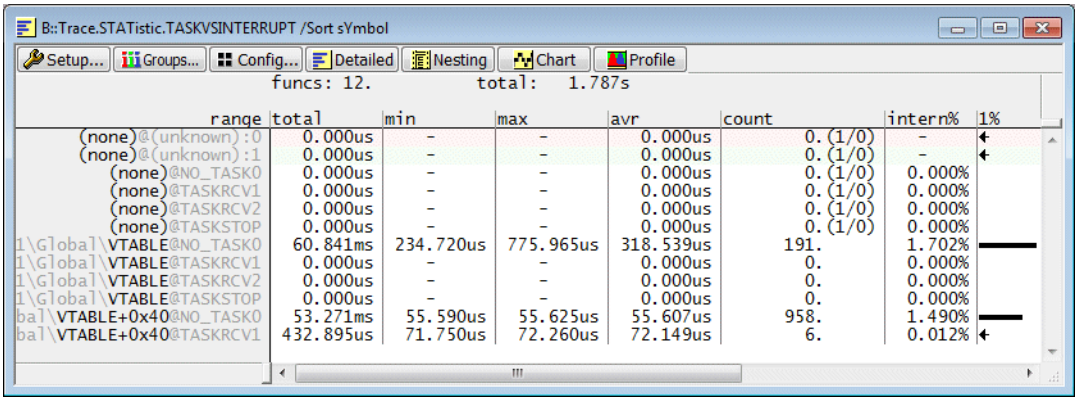
Trace.STATistic.TASKORINTERRUPT [**/SplitCORE**]	Statistic for interrupts and tasks (default), results split up per core
Trace.STATistic.TASKORINTERRUPT /CORE *<n>*	Statistic for interrupts and tasks for specified core



Trace.Chart.TASKVSINTERRUPT [**/SplitCORE**]	Interrupt time chart, task-related (default), results split up per core
Trace.Chart.TASKVSINTERRUPT /CORE *<n>*	Interrupt time chart task-related, for specified core



Trace.STATistic.TASKVSINTERRUPT [**/SplitCORE**]	Interrupt statistic, task-related (default), results split up per core
Trace.STATistic.TASKVSINTERRUPT /CORE *<n>*	Interrupt statistic, task-related, for specified core



Belated Trace Analysis (OS)

The TRACE32 Instruction Set Simulator can be used for a belated OS-aware trace evaluation. To set up the TRACE32 Instruction Set Simulator for belated OS-aware trace evaluation proceed as follows:

1. Save the trace information for the belated evaluation to a file.

```
Trace.SAVE belated__orti.ad
```

2. Set up the TRACE32 Instruction Set Simulator for a belated OS-aware trace evaluation (here OSEK on a MPC5553):

```
SYStem.CPU MPC5553                          ; select the target CPU

SYStem.Up                                   ; establish the
                                            ; communication between
                                            ; TRACE32 and the TRACE32
                                            ; Instruction Set
                                            ; Simulator

Trace.LOAD belated_orti.ad                  ; load the trace file

Data.Load.ELF my_app.out /NoCODE /GHS       ; load the symbol and
                                            ; debug information

TASK.ORTI my_orti.ort                       ; load the ORTI file

Trace.List List.TASK DEFault                ; display the trace
                                            ; listing
```

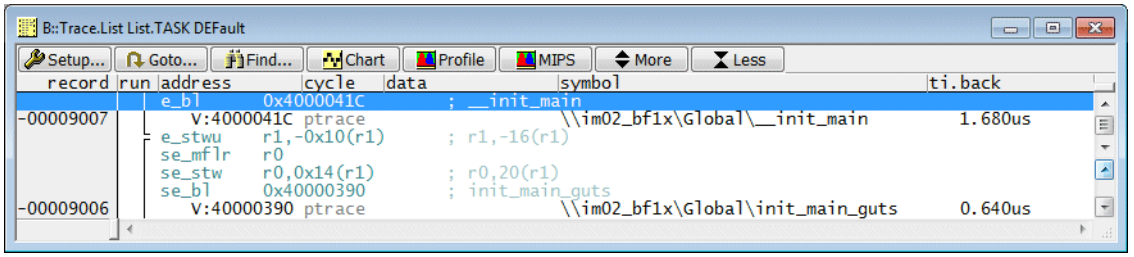
Function Run-Times Analysis (Overview)

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

If you use Branch History Tracing it is recommended to enable Program Trace Correlation Messages for bl *<func>* and e_bl *<func>* instructions (saves return address in link register, then jumps to *<func>*) (IEEE-ISTO 5001-2008 and subsequent standards only).

As a result function entries are timestamped in the trace.

```
NEXUS.HTM ON

NEXUS.PTCM BL_HTM ON                    ; generate Program Trace
                                        ; Correlation message when a
                                        ; "Branch and Link" instruction
                                        ; executes
```



Software under Analysis (no OS or OS)

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

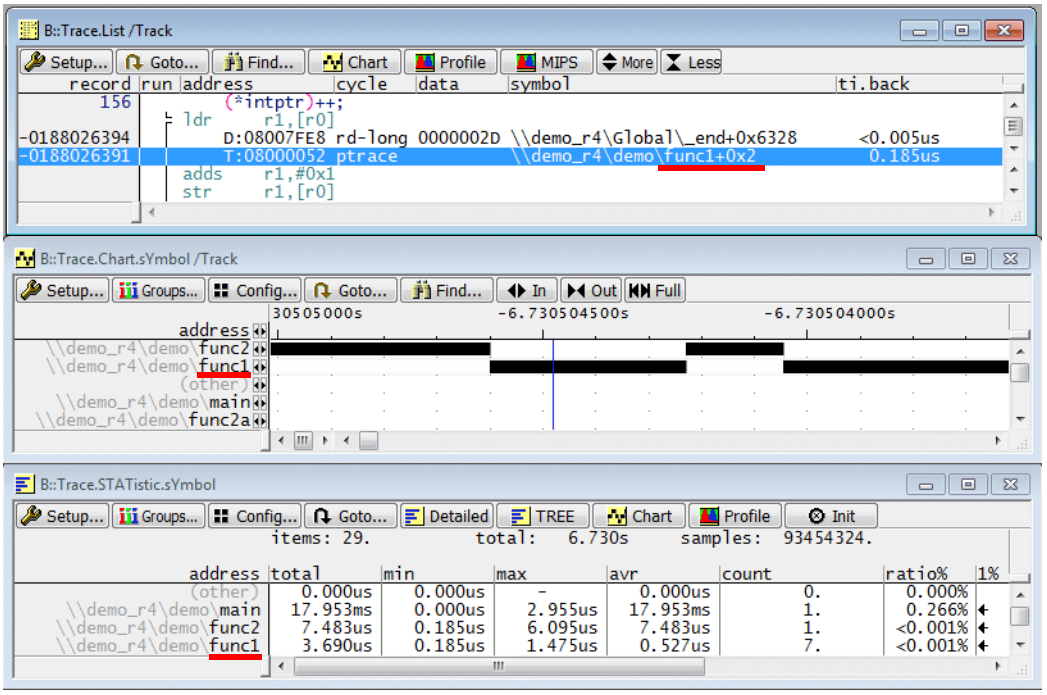
1. Software without operating system (abbreviation: **no OS**)
2. Software with an operating system (abbreviation: **OS**)

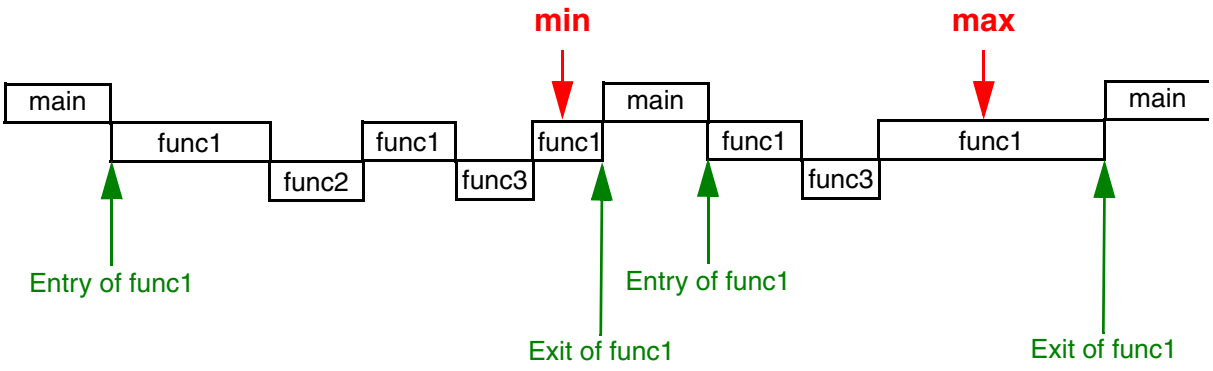
Flat vs. Nesting Analysis

TRACE32 provides two methods to analyze function run-times:

- Flat analysis
- Nesting analysis

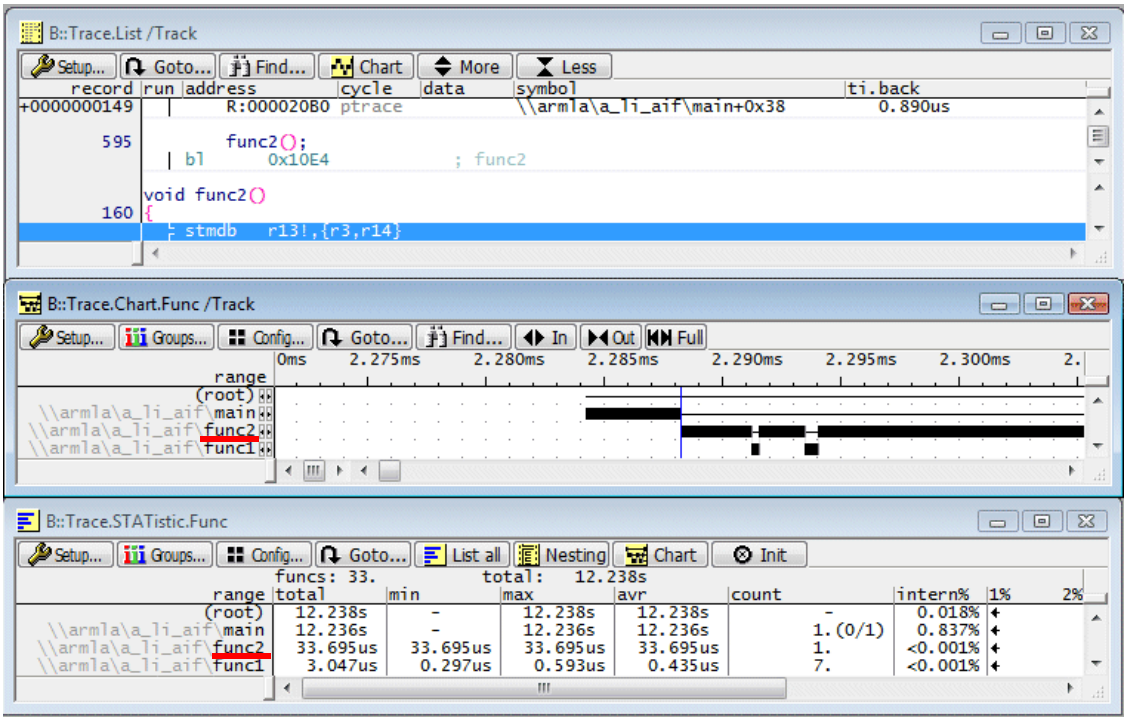
The flat function run-time analysis bases on the symbolic instruction addresses of the trace entries. The time spent by an instruction is assigned to the corresponding function/symbol region.

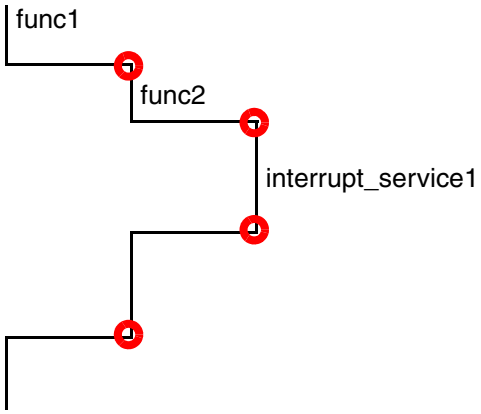




min	shortest time continuously in the address range of the function/symbol region
max	longest time continuously in the address range of the function/symbol region

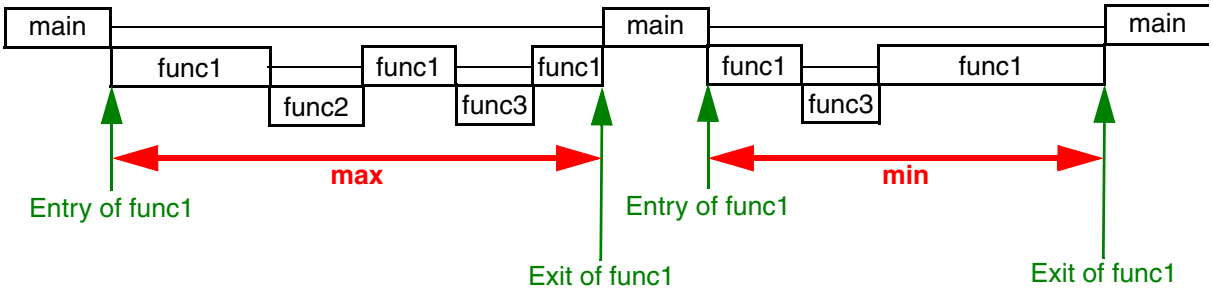
The function nesting analysis analyses only high-level language functions.





In order to display a nesting function run-time analysis TRACE32 analyzes the structure of the program execution by processing the trace information. The focus is put on the transition between functions (see picture above). The following events are of interest:

1. **Function entries**
2. **Function exits**
3. **Entries to interrupt service routines**
4. **Exits of interrupt service routines**
5. **Entries to TRAP handlers** (not implemented yet)
6. **Exits of TRAP handlers** (not implemented yet)



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.

Function Run-Times Analysis - Single

This chapter applies for single-core TRACE32 instances.

Flat Analysis

It is recommended to reduce the trace information generated by NEXUS to the required minimum.

- To avoid an overload of the NEXUS port.
- To make best use of the available trace memory.
- To get a more accurate timestamp.

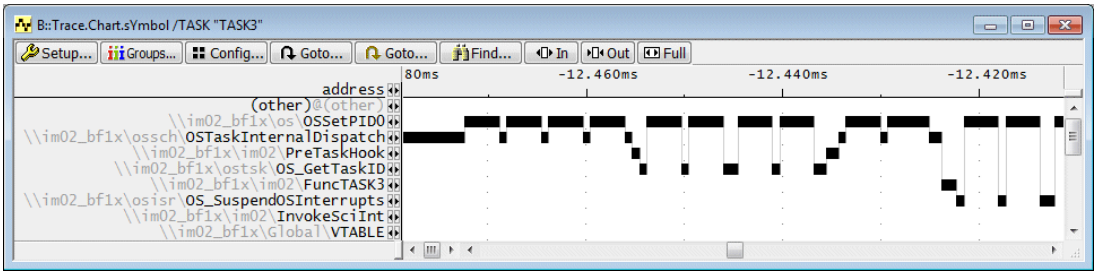
Optimum NEXUS Configuration (No OS)

Flat function run-time analysis does **not** require any **data information** if no OS is used. That's why it is recommended to switch Data Trace Messaging off.

```
NEXUS.DTM OFF
```

Your function time chart **can** include task information if you advise NEXUS to export the instruction flow and task switches. For details refer to the chapter **"OS-Aware Tracing (ORTI File)"**, page 193.

```
Trace.Chart.sYmbol /TASK "TASK3"
```



Optimum Configuration 1 (if OSEK generated OTMs)**:**

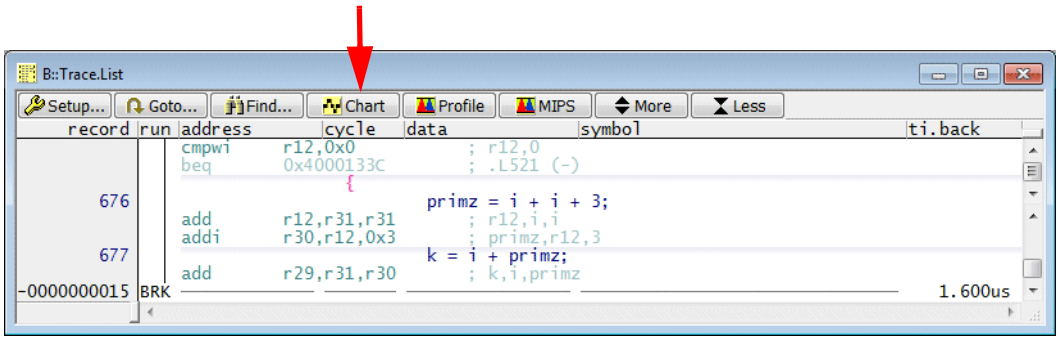
```
NEXUS.OTM ON
```

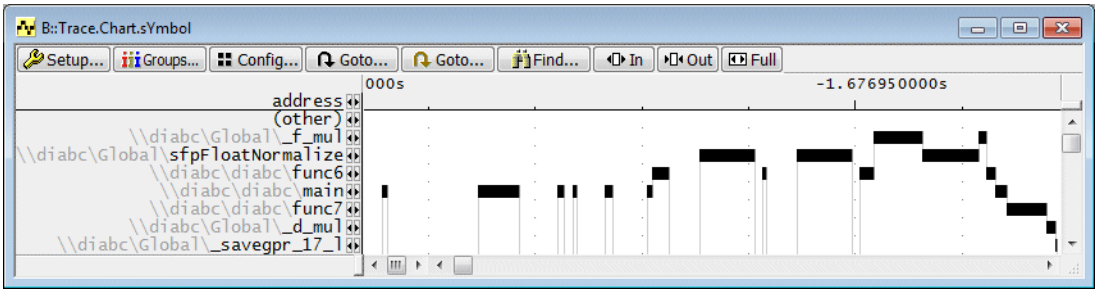
Optimum Configuration 2 (if OSEK does not support OTMs, NEXUS class 3 only)**:**

```
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

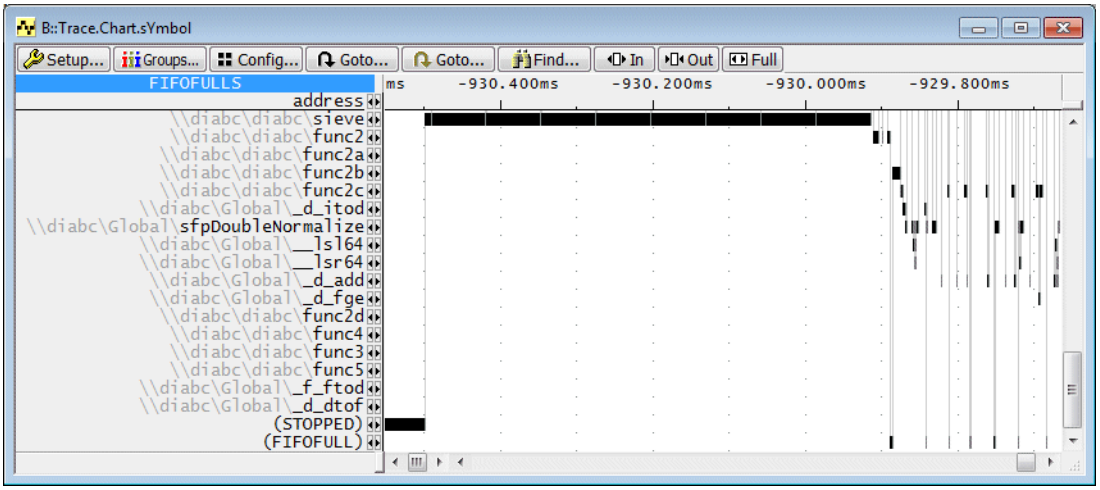
TRACE32 PowerView provides a time chart which shows when the program counter was in which function/symbol range.

Pushing the **Chart** button in the **Trace.List** window opens a **Trace.Chart.sYmbol** window





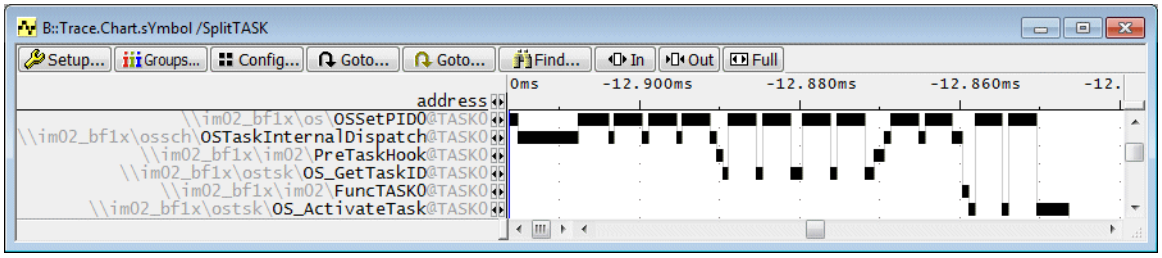
Trace.Chart.sYmbol	Display function time chart (no OS)
Trace.Chart.sYmbol [**/MergeTASK**]	Display function time chart (OS but task information is not of interest)

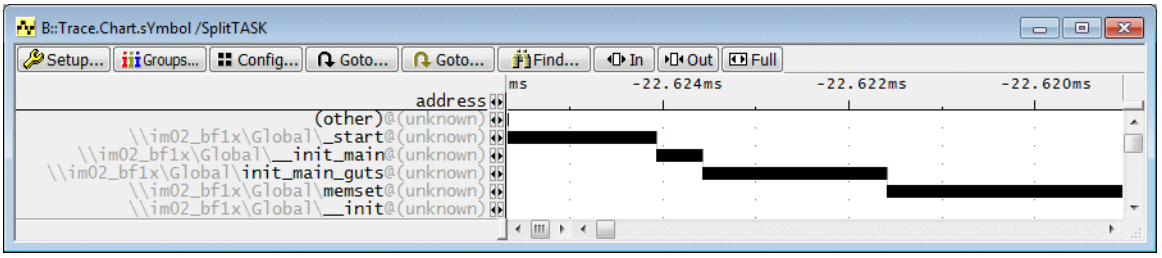


(STOPPED): If the trace recording contains time periods in which the program execution was stopped, these time periods are assigned to (STOPPED).

(FIFOFULL): If the trace recording contains time periods in which FIFO overflow was indicated, these time periods are assigned to (FIFOFULL).

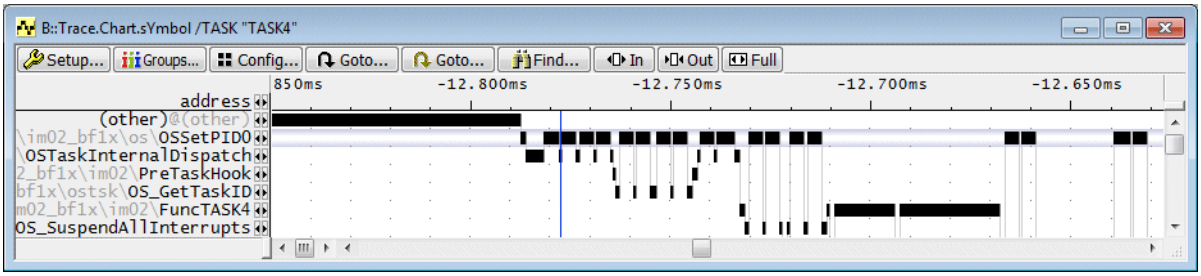
Trace.Chart.sYmbol /SplitTASK	Display function time chart including task information (OS only)



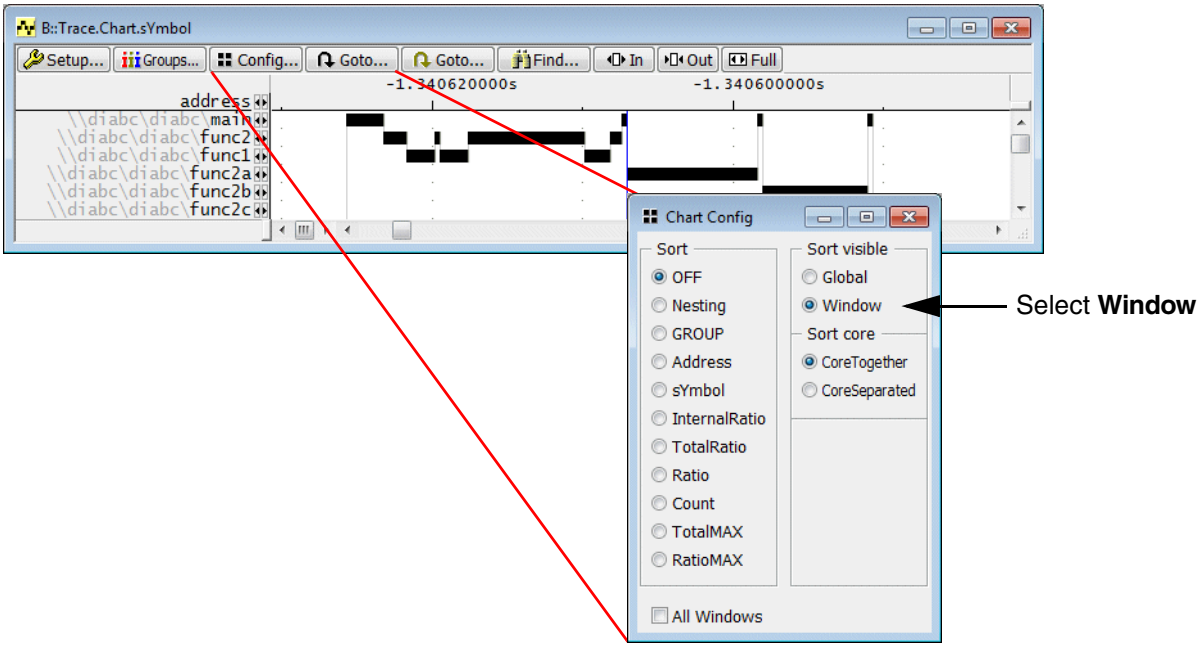


@*<task_name>*	Task name information
@(unknown)	• Function was running before the OS was started • Function was recorded before first task switch information was recorded
(UNKNOWN)@	Message decoding not possible.
(other)@(unknown)	No trace information available.

Trace.Chart.sYmbol /TASK *<task_name>*	Display function time chart for specified task (OS only)



(other)@(other)	All other trace information

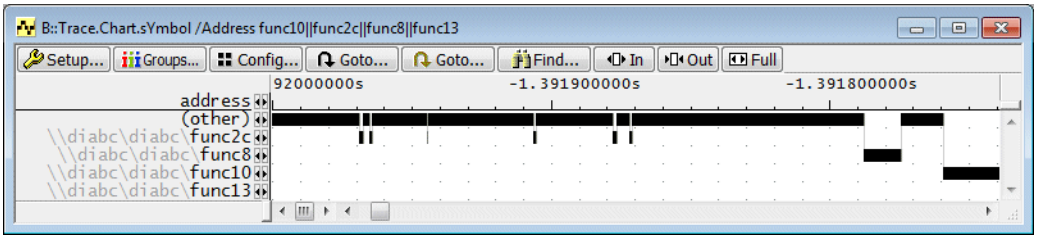


If **Window** is selected in the **Chart Config** window, the functions that are active at the selected point of time are visualized in the scope of the **Trace.Chart.sYmbol** window. This is helpful especially if you scroll horizontally.

For a detailed description of all options provided by the **Chart Config** window refer to the command description of **Trace.STATistic.Sort**.

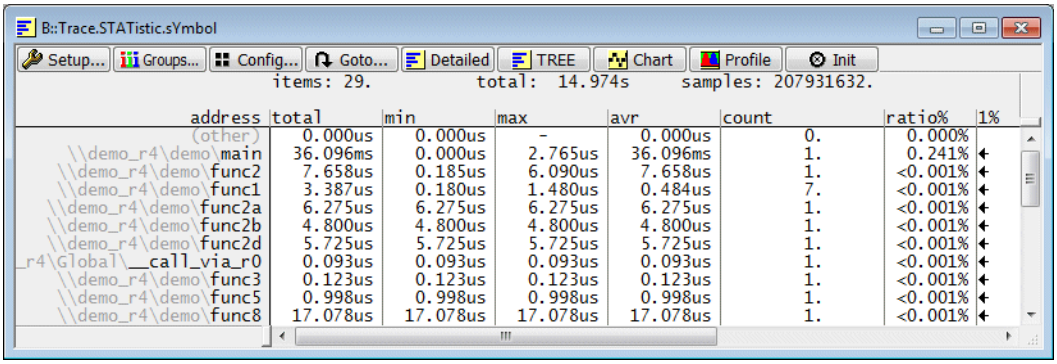
If you want to get the time chart only for a few functions, you can use the **/Address** option to list them.

Trace.Chart.sYmbol /Address *<func1>*||*<func2>*||*…*



More features to structure your trace analysis are introduced in **“Structure the Trace Evaluation”**, page 335.

Analog to the timing diagram also a numerical analysis is provided.



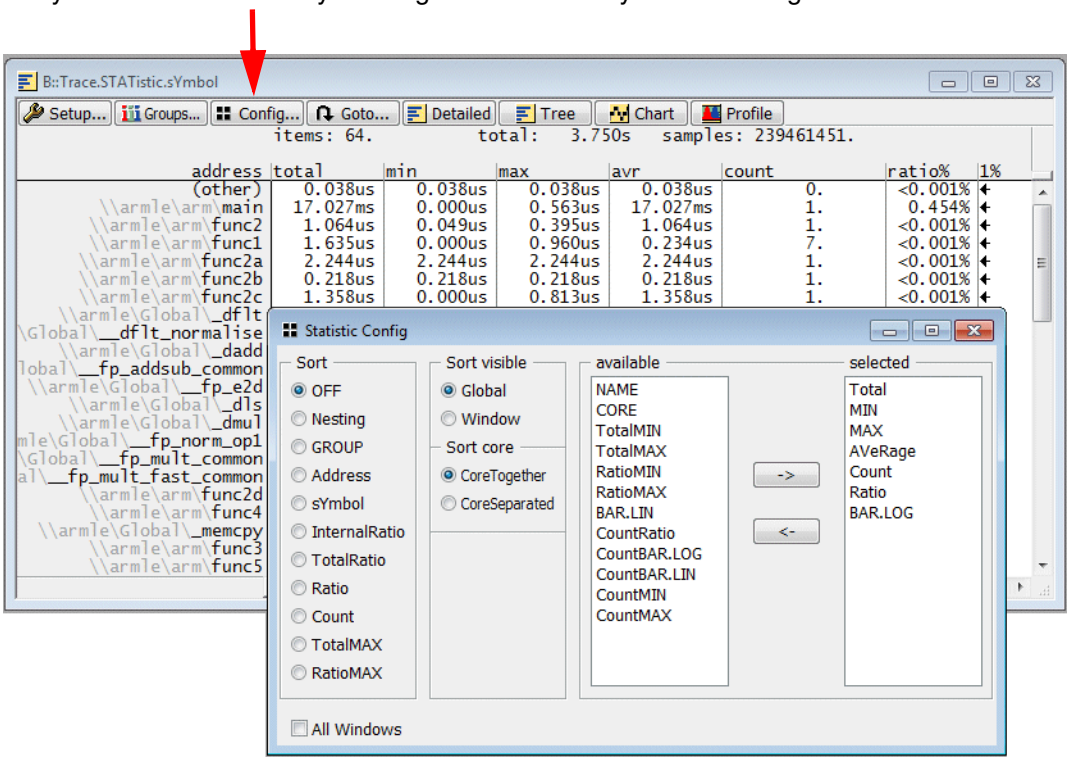
address	total	min	max	avr	count	ratio%	1%
(other)	0.000us	0.000us	-	0.000us	0.	0.000%	
\\demo_r4\demo\main	36.096ms	0.000us	2.765us	36.096ms	1.	0.241%	
\\demo_r4\demo\func2	7.658us	0.185us	6.090us	7.658us	1.	<0.001%	
\\demo_r4\demo\func1	3.387us	0.180us	1.480us	0.484us	7.	<0.001%	
\\demo_r4\demo\func2a	6.275us	6.275us	6.275us	6.275us	1.	<0.001%	
\\demo_r4\demo\func2b	4.800us	4.800us	4.800us	4.800us	1.	<0.001%	
\\demo_r4\demo\func2d	5.725us	5.725us	5.725us	5.725us	1.	<0.001%	
_r4\Global__call_via_r0	0.093us	0.093us	0.093us	0.093us	1.	<0.001%	
\\demo_r4\demo\func3	0.123us	0.123us	0.123us	0.123us	1.	<0.001%	
\\demo_r4\demo\func5	0.998us	0.998us	0.998us	0.998us	1.	<0.001%	
\\demo_r4\demo\func8	17.078us	17.078us	17.078us	17.078us	1.	<0.001%	

survey	
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)

Only the grey rows provide useful information about the run-time of a function/symbol range.

function details	
address	function/symbol region 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 as total/count)
count	number of new entries (start address executed) into the address range of the function/symbol region
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 for the address column or a different column layout.
By default the functions/symbol regions are sorted by their recording order.



Trace.STATistic.sYmbol	Flat function run-time analysis (no OS) - numerical display
Trace.STATistic.sYmbol [**/MergeTASK**]	Flat function run-time analysis (OS) - numerical display - no task information
Trace.STATistic.sYmbol /SplitTASK	Flat function run-time analysis (OS) - numerical display including task information
Trace.STATistic.sYmbol /TASK *<task_name>*	Flat function run-time analysis (OS) - numerical display for specified task

Restrictions

1. **The nesting analysis analyses only high-level language functions.**
2. **The nested function run-time analysis expects common ways to enter/exit functions.**
3. **The nesting analysis is sensitive with regards to FIFOFULLs.**

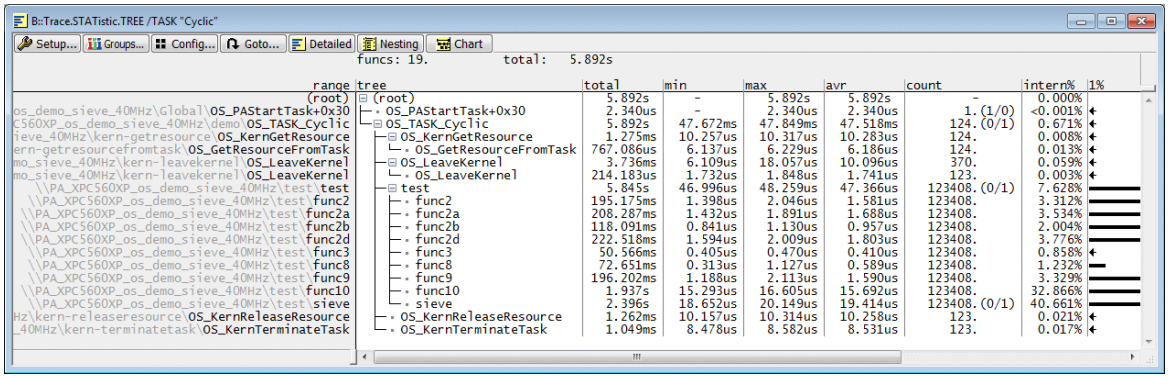
Optimum NEXUS Configuration (No OS)

The nesting function run-time analysis doesn't require any data information if no OS is used. That's why it is recommended to switch the export of data information off.

```
NEXUS.DTM OFF
```

TRACE32 PowerView builds up a separate call tree for each task.

```
Trace.STATistic.TREE /TASK "Cyclic"
```



range	tree	total	min	max	avr	count	intern%	1%
(root)	(root)	5.892s	-	5.892s	5.892s	-	0.000%	
os_demo_sieve_40MHz\Global\OS_PAStartTask+0x30	OS_PAStartTask+0x30	2.340us	-	2.340us	2.340us	1.(1/0)	<0.001%	
C560XP_os_demo_sieve_40MHz\demo\OS_TASK_Cyclic	OS_TASK_Cyclic	5.892s	47.672ms	47.849ms	47.518ms	124.(0/1)	0.671%	
ieve_40MHz\kern-getresource\OS_KernGetResource	OS_KernGetResource	1.275ms	10.257us	10.317us	10.283us	124.	0.008%	
ern-getresourcefromtask\OS_GetResourceFromTask	OS_GetResourceFromTask	767.086us	6.137us	6.229us	6.186us	124.	0.013%	
mo_sieve_40MHz\kern-leavekernel\OS_LeaveKernel	OS_LeaveKernel	3.736ms	6.109us	18.057us	10.096us	370.	0.059%	
mo_sieve_40MHz\kern-leavekernel\OS_LeaveKernel	OS_LeaveKernel	214.183us	1.732us	1.848us	1.741us	123.	0.003%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\test	test	5.845s	46.996us	48.259us	47.366us	123408.(0/1)	7.628%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func2	func2	195.175ms	1.398us	2.046us	1.581us	123408.	3.312%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func2a	func2a	208.287ms	1.432us	1.891us	1.688us	123408.	3.534%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func2b	func2b	118.091ms	0.841us	1.130us	0.957us	123408.	2.004%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func2d	func2d	222.518ms	1.594us	2.009us	1.803us	123408.	3.776%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func3	func3	50.566ms	0.405us	0.470us	0.410us	123408.	0.858%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func8	func8	72.651ms	0.313us	1.127us	0.589us	123408.	1.232%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func9	func9	196.202ms	1.188us	2.113us	1.590us	123408.	3.329%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\func10	func10	1.937s	15.293us	16.605us	15.692us	123408.	32.866%	
\\PA_XPC560XP_os_demo_sieve_40MHz\test\sieve	sieve	2.396s	18.652us	20.149us	19.414us	123408.(0/1)	40.661%	
Hz\kern-releaseresource\OS_KernReleaseResource	OS_KernReleaseResource	1.262ms	10.157us	10.314us	10.258us	123.	0.021%	
_40MHz\kern-terminatetask\OS_KernTerminateTask	OS_KernTerminateTask	1.049ms	8.478us	8.582us	8.531us	123.	0.017%	

In order to hook a function entry/exit into the correct call tree, TRACE32 PowerView needs to know which task was running when the entry/exit occurred.

The standard way to get information on the current task is to advise the NEXUS to export the instruction flow and task switches. For details refer to the chapter **"OS-Aware Tracing (ORTI File)"**, page 193.

Optimum Configuration 1 (if OSEK generated OTMs)**:**

```
NEXUS.OTM ON

; default setting since 2015-01
Trace.STATistic.InterruptIsFunction ON
```

Optimum Configuration 2 (if OSEK does not support OTMs, NEXUS class 3 only)**:**

```
Break.Set TASK.CONFIG(magic) /Write /TraceData

; default setting since 2015-01
Trace.STATistic.InterruptIsFunction ON
```

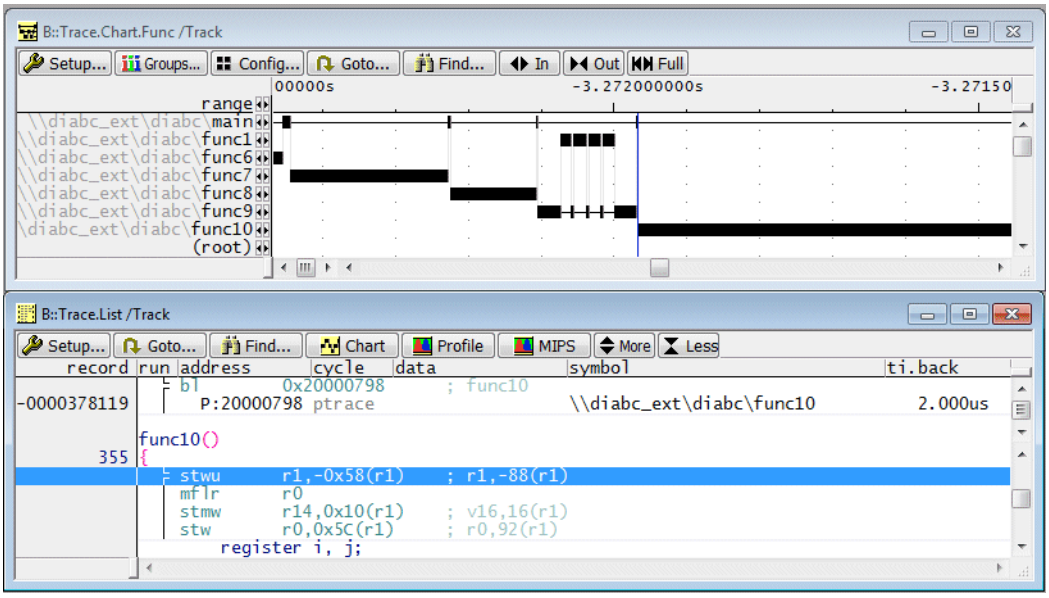
In order to prepare the results for the nesting analysis TRACE32 post-processes the instruction flow to find:

- **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.

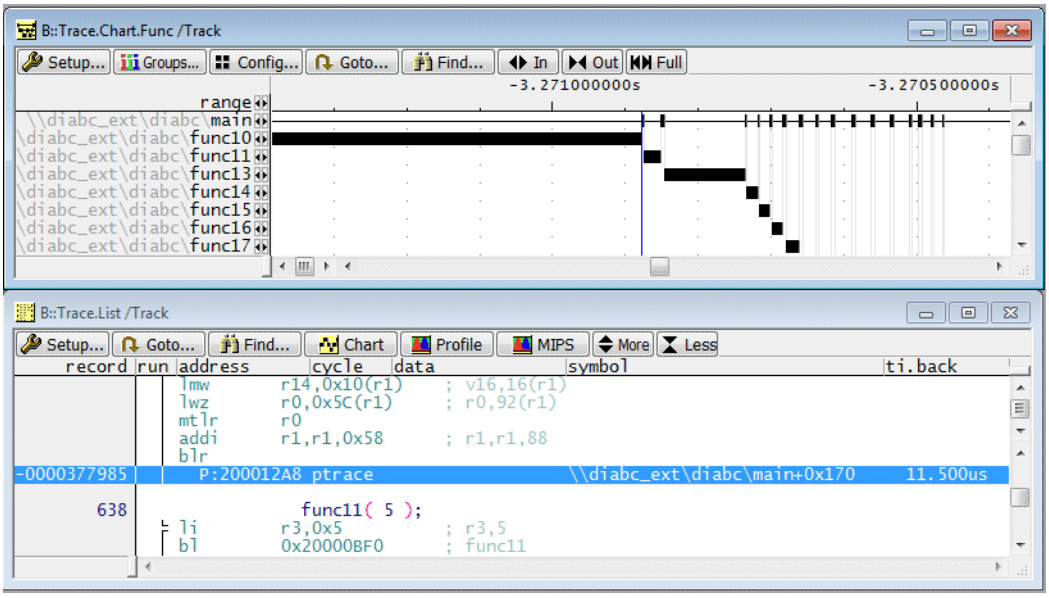
```
Trace.Chart.Func                        ; function func10 as
                                        ; example

Trace.List /Track
```



- **Function exits**

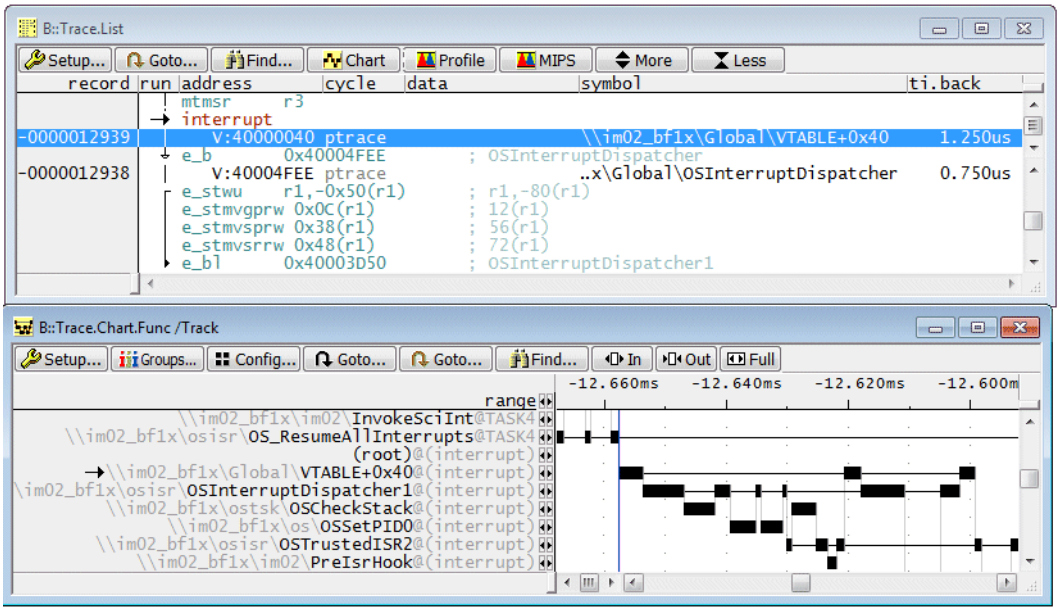
  A RETURN instruction within an HLL function is regarded as function exit.

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



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

  If an indirect branch to the Interrupt Vector Table occurs, an interrupt entry is detected. The interrupt function gets the name VTABLE+<offset> if no symbol is specified.

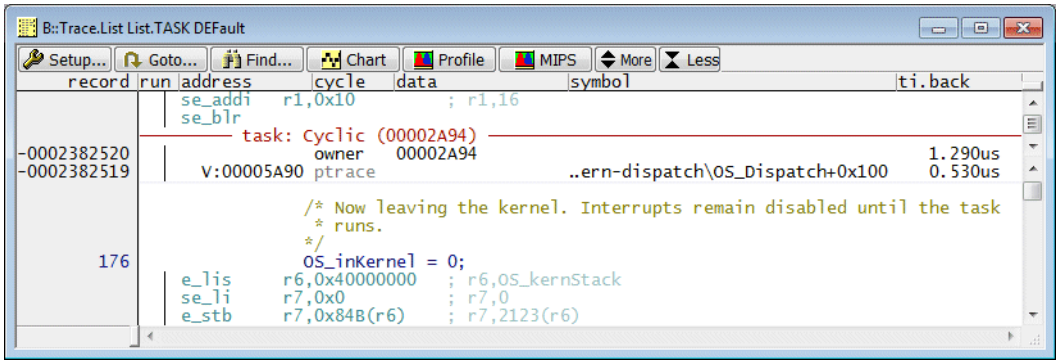


- **Exits of interrupt service routines**

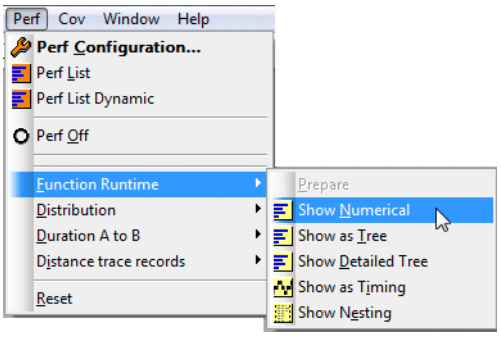
  RETURN FROM INTERRUPT is regarded as exit of the interrupt function.

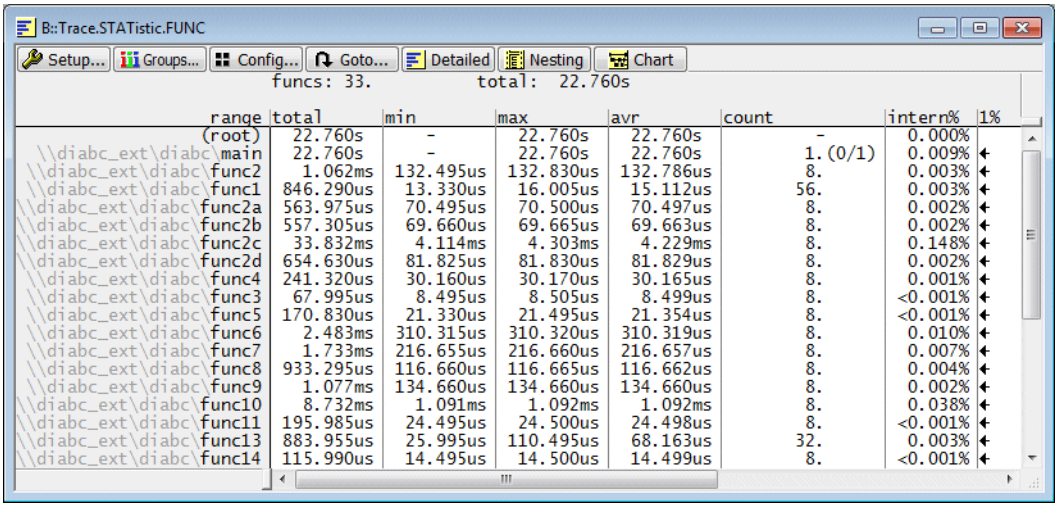
- **Entries to TRAP handlers** (not implemented yet)
- **Exits of TRAP handlers** (not implemented yet)
- **Task switches**

  Task switches are needed to build correct call trees if a target operating system is used.



Trace.STATistic.Func Nested function run-time analysis
- numeric display





range	total	min	max	avr	count	intern%	1%
(root)	22.760s	-	22.760s	22.760s	-	0.000%	
\\diabc_ext\diabc\main	22.760s	-	22.760s	22.760s	1.(0/1)	0.009%	
\\diabc_ext\diabc\func2	1.062ms	132.495us	132.830us	132.786us	8.	0.003%	
\\diabc_ext\diabc\func1	846.290us	13.330us	16.005us	15.112us	56.	0.003%	
\\diabc_ext\diabc\func2a	563.975us	70.495us	70.500us	70.497us	8.	0.002%	
\\diabc_ext\diabc\func2b	557.305us	69.660us	69.665us	69.663us	8.	0.002%	
\\diabc_ext\diabc\func2c	33.832ms	4.114ms	4.303ms	4.229ms	8.	0.148%	
\\diabc_ext\diabc\func2d	654.630us	81.825us	81.830us	81.829us	8.	0.002%	
\\diabc_ext\diabc\func4	241.320us	30.160us	30.170us	30.165us	8.	0.001%	
\\diabc_ext\diabc\func3	67.995us	8.495us	8.505us	8.499us	8.	<0.001%	
\\diabc_ext\diabc\func5	170.830us	21.330us	21.495us	21.354us	8.	<0.001%	
\\diabc_ext\diabc\func6	2.483ms	310.315us	310.320us	310.319us	8.	0.010%	
\\diabc_ext\diabc\func7	1.733ms	216.655us	216.660us	216.657us	8.	0.007%	
\\diabc_ext\diabc\func8	933.295us	116.660us	116.665us	116.662us	8.	0.004%	
\\diabc_ext\diabc\func9	1.077ms	134.660us	134.660us	134.660us	8.	0.002%	
\\diabc_ext\diabc\func10	8.732ms	1.091ms	1.092ms	1.092ms	8.	0.038%	
\\diabc_ext\diabc\func11	195.985us	24.495us	24.500us	24.498us	8.	<0.001%	
\\diabc_ext\diabc\func13	883.955us	25.995us	110.495us	68.163us	32.	0.003%	
\\diabc_ext\diabc\func14	115.990us	14.495us	14.500us	14.499us	8.	<0.001%	

```
funcs: 103.          total:  22.618ms  intr:   2.574ms  10 workarounds
```

survey	
funcs: *<number>*	number of functions in the trace
total: *<time>*	total measurement time
intr: *<time>*	total time in interrupt service routines

survey (issue indication)	
stopped: *<time>*	The analyzed trace recording contains program stops. *<time>* indicates the total time the program execution was stopped.
<number> **problems**	The nested analysis contains problems. Please contact **support@lauterbach.com**.
<number> **workarounds**	The nested analysis contains issues, but TRACE32 found solutions for them. It is recommended to perform a sanity check on the proposed solutions.
stack overflow at *<record>*	The nested analysis exceeds the nesting level 200. It is highly likely that the function exit for an often called function is missing. The command **Trace.STATistic.TREE** can help you to identify the function. If you need further help please contact **support@lauterbach.com**.
stack underflow at *<record>*	The nested analysis exceeds the nesting level 200. It is highly likely that the function entry for an often executed function is missing. The command **Trace.STATistic.TREE** can help you to identify the function. If you need further help please contact **support@lauterbach.com**.

The main reasons for all the issues are code optimizations.

columns	
range (NAME)	function name, sorted by their recording order as default

- **HLL function**

  ```
  \\diabc_ext\diabc\func6
  ```

- **(root)**

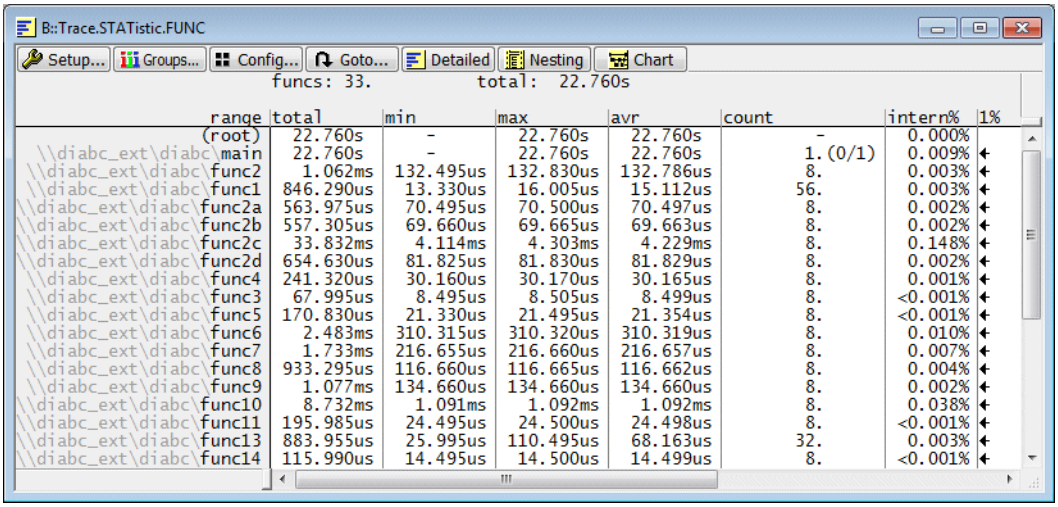
  ```
  (root)
  ```

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

- **Interrupt service routine**

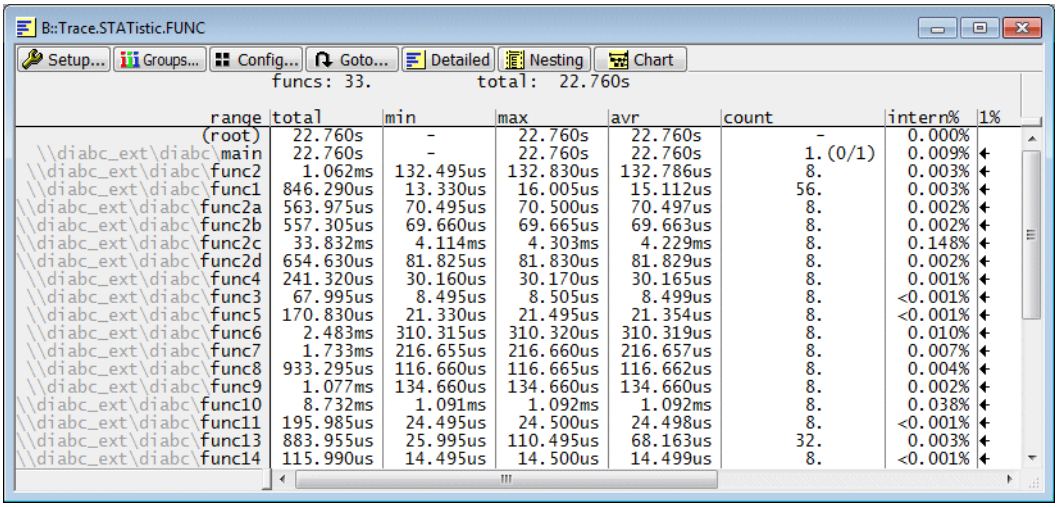
  ```
  →\\PA_XPC560XP_os_demo_sieve_40MHz\Global\OS_InterruptTable+0x78
  ```

- **HLL trap handler** (not implemented yet)



range	total	min	max	avr	count	intern%	1%
(root)	22.760s	-	22.760s	22.760s	-	0.000%	
\\diabc_ext\diabc\main	22.760s	-	22.760s	22.760s	1.(0/1)	0.009%	
\\diabc_ext\diabc\func2	1.062ms	132.495us	132.830us	132.786us	8.	0.003%	
\\diabc_ext\diabc\func1	846.290us	13.330us	16.005us	15.112us	56.	0.003%	
\\diabc_ext\diabc\func2a	563.975us	70.495us	70.500us	70.497us	8.	0.002%	
\\diabc_ext\diabc\func2b	557.305us	69.660us	69.665us	69.663us	8.	0.002%	
\\diabc_ext\diabc\func2c	33.832ms	4.114ms	4.303ms	4.229ms	8.	0.148%	
\\diabc_ext\diabc\func2d	654.630us	81.825us	81.830us	81.829us	8.	0.002%	
\\diabc_ext\diabc\func4	241.320us	30.160us	30.170us	30.165us	8.	0.001%	
\\diabc_ext\diabc\func3	67.995us	8.495us	8.505us	8.499us	8.	<0.001%	
\\diabc_ext\diabc\func5	170.830us	21.330us	21.495us	21.354us	8.	<0.001%	
\\diabc_ext\diabc\func6	2.483ms	310.315us	310.320us	310.319us	8.	0.010%	
\\diabc_ext\diabc\func7	1.733ms	216.655us	216.660us	216.657us	8.	0.007%	
\\diabc_ext\diabc\func8	933.295us	116.660us	116.665us	116.662us	8.	0.004%	
\\diabc_ext\diabc\func9	1.077ms	134.660us	134.660us	134.660us	8.	0.002%	
\\diabc_ext\diabc\func10	8.732ms	1.091ms	1.092ms	1.092ms	8.	0.038%	
\\diabc_ext\diabc\func11	195.985us	24.495us	24.500us	24.498us	8.	<0.001%	
\\diabc_ext\diabc\func13	883.955us	25.995us	110.495us	68.163us	32.	0.003%	
\\diabc_ext\diabc\func14	115.990us	14.495us	14.500us	14.499us	8.	<0.001%	

columns (cont.)	
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

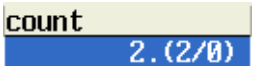


range	total	min	max	avr	count	intern%	1%
(root)	22.760s	-	22.760s	22.760s	-	0.000%	
\\diabc_ext\diabc\main	22.760s	-	22.760s	22.760s	1.(0/1)	0.009%	
\\diabc_ext\diabc\func2	1.062ms	132.495us	132.830us	132.786us	8.	0.003%	
\\diabc_ext\diabc\func1	846.290us	13.330us	16.005us	15.112us	56.	0.003%	
\\diabc_ext\diabc\func2a	563.975us	70.495us	70.500us	70.497us	8.	0.002%	
\\diabc_ext\diabc\func2b	557.305us	69.660us	69.665us	69.663us	8.	0.002%	
\\diabc_ext\diabc\func2c	33.832ms	4.114ms	4.303ms	4.229ms	8.	0.148%	
\\diabc_ext\diabc\func2d	654.630us	81.825us	81.830us	81.829us	8.	0.002%	
\\diabc_ext\diabc\func4	241.320us	30.160us	30.170us	30.165us	8.	0.001%	
\\diabc_ext\diabc\func3	67.995us	8.495us	8.505us	8.499us	8.	<0.001%	
\\diabc_ext\diabc\func5	170.830us	21.330us	21.495us	21.354us	8.	<0.001%	
\\diabc_ext\diabc\func6	2.483ms	310.315us	310.320us	310.319us	8.	0.010%	
\\diabc_ext\diabc\func7	1.733ms	216.655us	216.660us	216.657us	8.	0.007%	
\\diabc_ext\diabc\func8	933.295us	116.660us	116.665us	116.662us	8.	0.004%	
\\diabc_ext\diabc\func9	1.077ms	134.660us	134.660us	134.660us	8.	0.002%	
\\diabc_ext\diabc\func10	8.732ms	1.091ms	1.092ms	1.092ms	8.	0.038%	
\\diabc_ext\diabc\func11	195.985us	24.495us	24.500us	24.498us	8.	<0.001%	
\\diabc_ext\diabc\func13	883.955us	25.995us	110.495us	68.163us	32.	0.003%	
\\diabc_ext\diabc\func14	115.990us	14.495us	14.500us	14.499us	8.	<0.001%	

columns (cont.)	
count	number of 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>*).



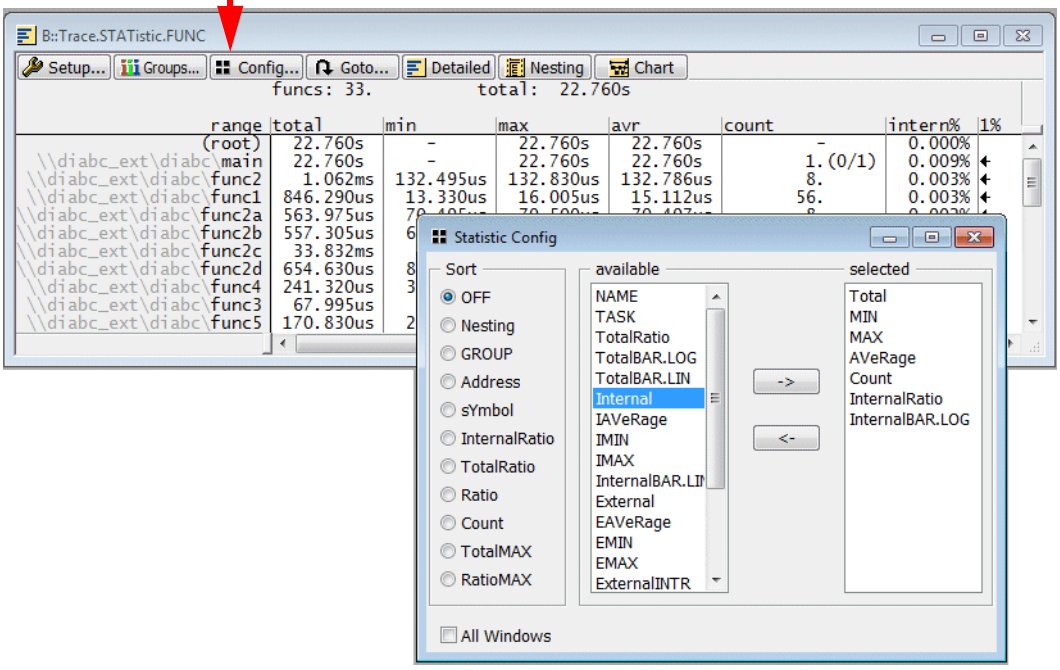
Interpretation examples:

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.)	
intern% (InternalRatio, InternalBAR.LOG)	ratio of time within the function without subfunctions, TRAP handlers, interrupts

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

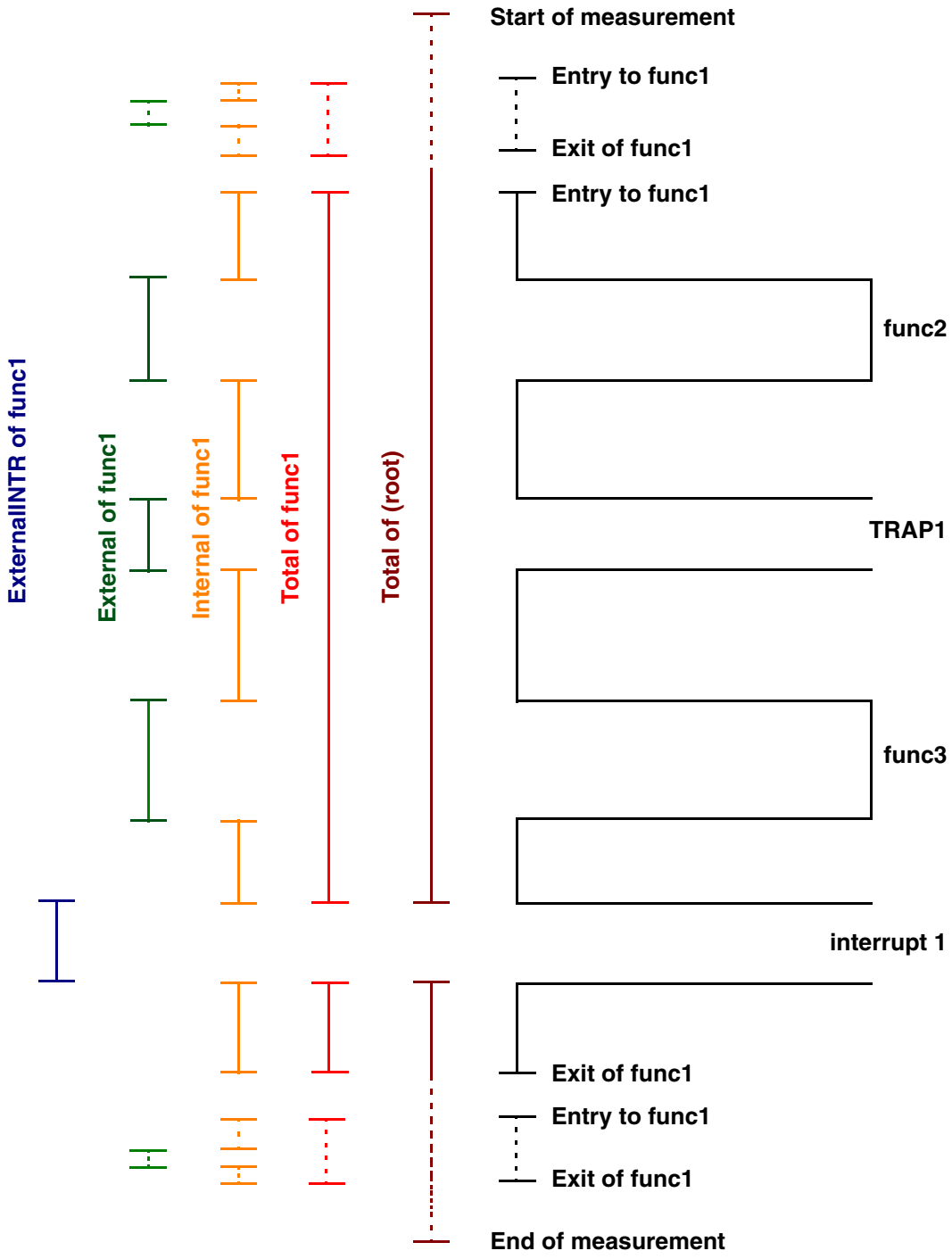


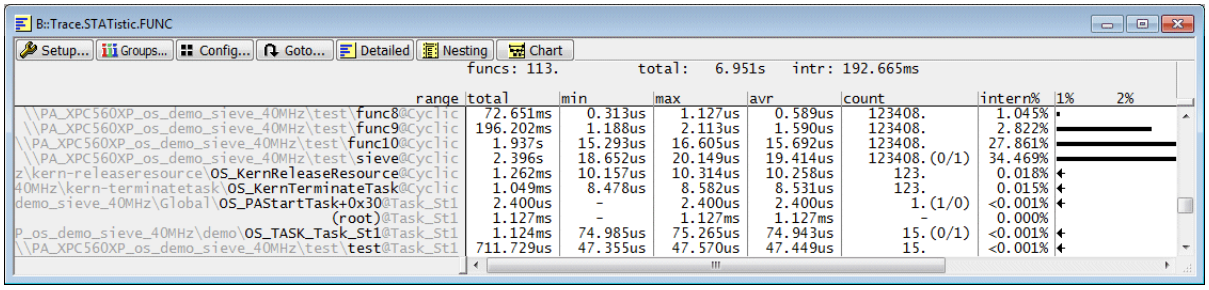
columns (cont.) - times only in function	
Internal	total time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IAVeRage	average time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IMIN	shortest 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	
ExternalINTR	total time the function was interrupted
ExternalINTRMAX	max. time one function pass was interrupted
INTRCount	number of interrupts that occurred during the function run-time

The following graphic give an overview how times are calculated:





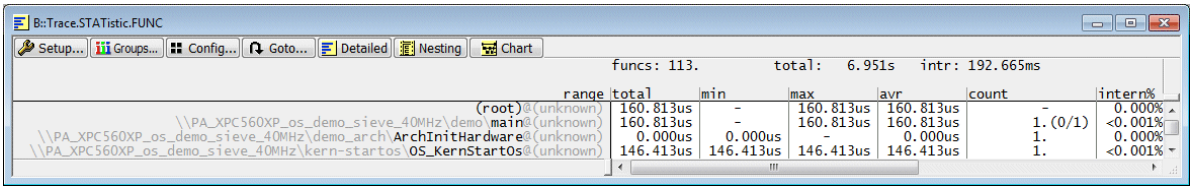
- **HLL function**

  \\PA_XPC560XP_os_demo_sieve_40MHz\test\func2@Cyclic

  HLL function “func2” running in task “Cyclic”

- **Root of call tree for task “Cyclic”**

  (root)@Cyclic



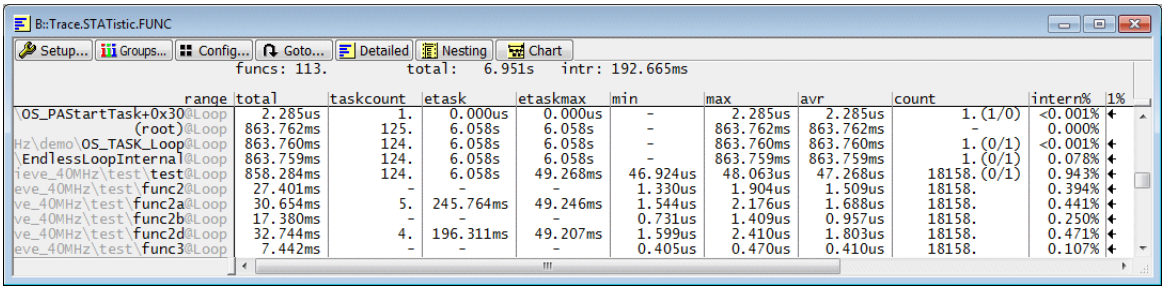
- **Unknown task**

  \\PA_XPC560XP_os_demo_sieve_40MHz\demo\main@(unknown)

  Before the first task switch is found in the trace, the task is unknown

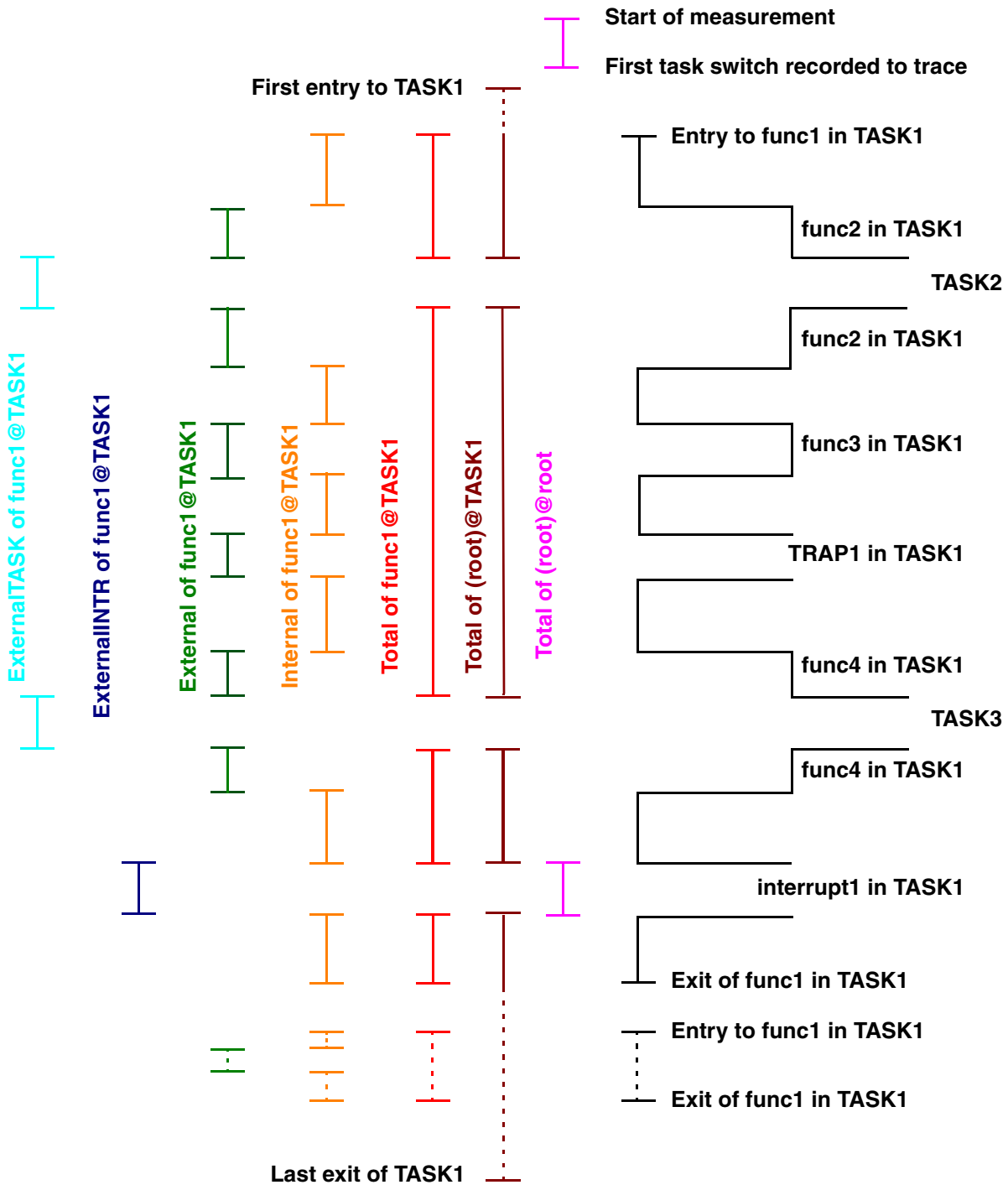
- **Root of unknown task**

  (root)@(unknown)



range	total	taskcount	etask	etaskmax	min	max	avr	count	intern%	1%
\OS_PAStartTask+0x30@Loop	2.285us	1.	0.000us	0.000us	-	2.285us	2.285us	1.(1/0)	<0.001%	
(root)@Loop	863.762ms	125.	6.058s	6.058s	-	863.762ms	863.762ms	-	0.000%	
Hz\demo\OS_TASK_Loop@Loop	863.760ms	124.	6.058s	6.058s	-	863.760ms	863.760ms	1.(0/1)	<0.001%	
\EndlessLoopInternal@Loop	863.759ms	124.	6.058s	6.058s	-	863.759ms	863.759ms	1.(0/1)	0.078%	
ieve_40MHz\test\test@Loop	858.284ms	124.	6.058s	49.268ms	46.924us	48.063us	47.268us	18158.(0/1)	0.943%	
eve_40MHz\test\func2@Loop	27.401ms	-	-	-	1.330us	1.904us	1.509us	18158.	0.394%	
ve_40MHz\test\func2a@Loop	30.654ms	5.	245.764ms	49.246ms	1.544us	2.176us	1.688us	18158.	0.441%	
ve_40MHz\test\func2b@Loop	17.380ms	-	-	-	0.731us	1.409us	0.957us	18158.	0.250%	
ve_40MHz\test\func2d@Loop	32.744ms	4.	196.311ms	49.207ms	1.599us	2.410us	1.803us	18158.	0.471%	
eve_40MHz\test\func3@Loop	7.442ms	-	-	-	0.405us	0.470us	0.410us	18158.	0.107%	

columns - task/thread related information	
TASKCount	number of tasks that interrupt the function
ExternalTASK	total time in other tasks
ExternalTASKMAX	max. time 1 function pass was interrupted by a task


Start of measurement
First task switch recorded to trace
First entry to TASK1
Entry to func1 in TASK1
func2 in TASK1
TASK2
func2 in TASK1
func3 in TASK1
TRAP1 in TASK1
func4 in TASK1
TASK3
func4 in TASK1
interrupt1 in TASK1
Exit of func1 in TASK1
Entry to func1 in TASK1
Exit of func1 in TASK1
Last exit of TASK1
ExternalTASK of func1@TASK1
ExternalINTR of func1@TASK1
External of func1@TASK1
Internal of func1@TASK1
Total of func1@TASK1
Total of (root)@TASK1
Total of (root)@root

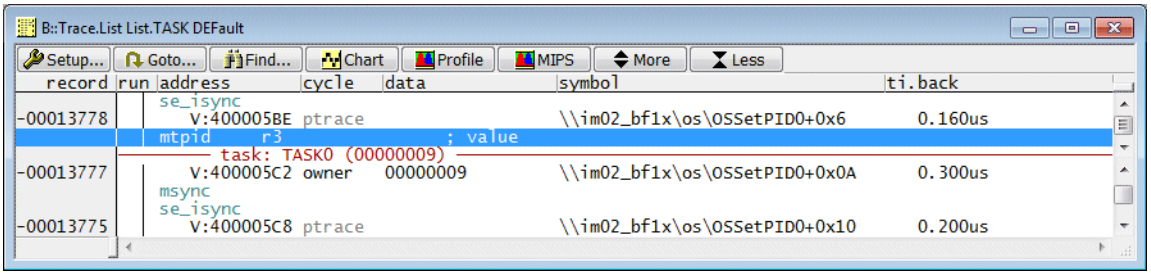
The standard NEXUS settings do often not allow to locate exactly the instructions that are already executed by a newly activated task. This is especially true is Branch History Messaging is used. This might disturb the task-aware function run-time measurement.

An instruction-accurate assignment of the task switches may improve the results.

IEEE-ISTO 5001-2008 and Subsequent Standards

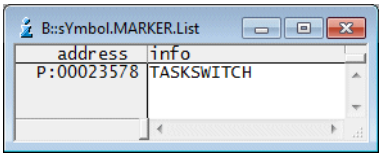
The Ownership Trace Messages (task switches) can be exactly assigned to an instruction, if the following setting is done.

```
NEXUS.PTCM PID_MSR ON                   ; enable Program Trace Correlation
                                        ; Messages for PID0/NPIDR accesses

NEXUS.POTD ON                           ; disable Periodic Ownership Trace
                                        ; Messages
```



Alternative

```
; mark instruction that performs the task switch for the task-aware
; function run-time analysis
sYmbol.MARKER.Create TASKSWITCH osDispatcher+0x100
```



TRACE32 analyzes the structure of the program execution by processing the trace information in order to provide the nesting statistic. The objective is to construct a complete call tree. When a OS is used, it is more likely the TRACE32 has issues while construction the call tree. There are two types of issues:

- **PROBLEMS**

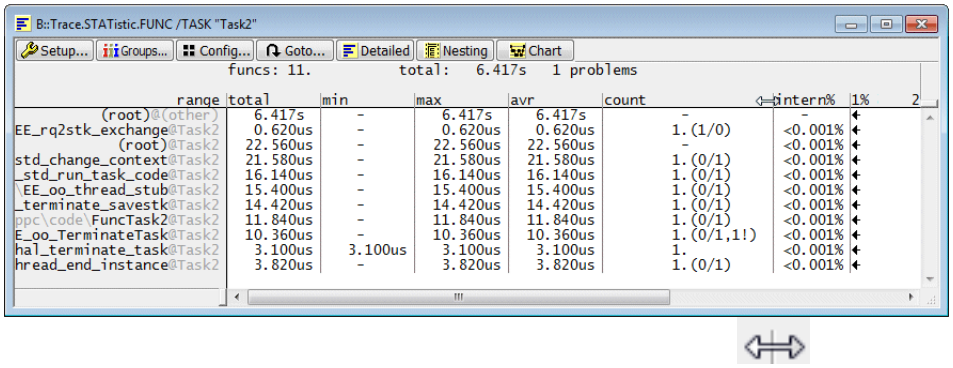
  A PROBLEM is a point it the trace recording that TRACE32 can not integrate into the current nesting. TRACE32 does not discard this point for the call tree, it integrates this point by assigning a meaningful interpretation.

  TRACE32 marks functions that include a PROBLEM with **!** in the count column.

- **WORKAROUNDS**

  A WORKAROUND is a point it the trace recording that TRACE32 can not integrate into the current nesting. But TRACE32 integrates this point into the function nesting, by supplementing information based on previous scenarios in the nesting. TRACE32 marks functions that include a WORKAROUND with **?** in the count column.

It is recommended to drag the count column wider to see all details.

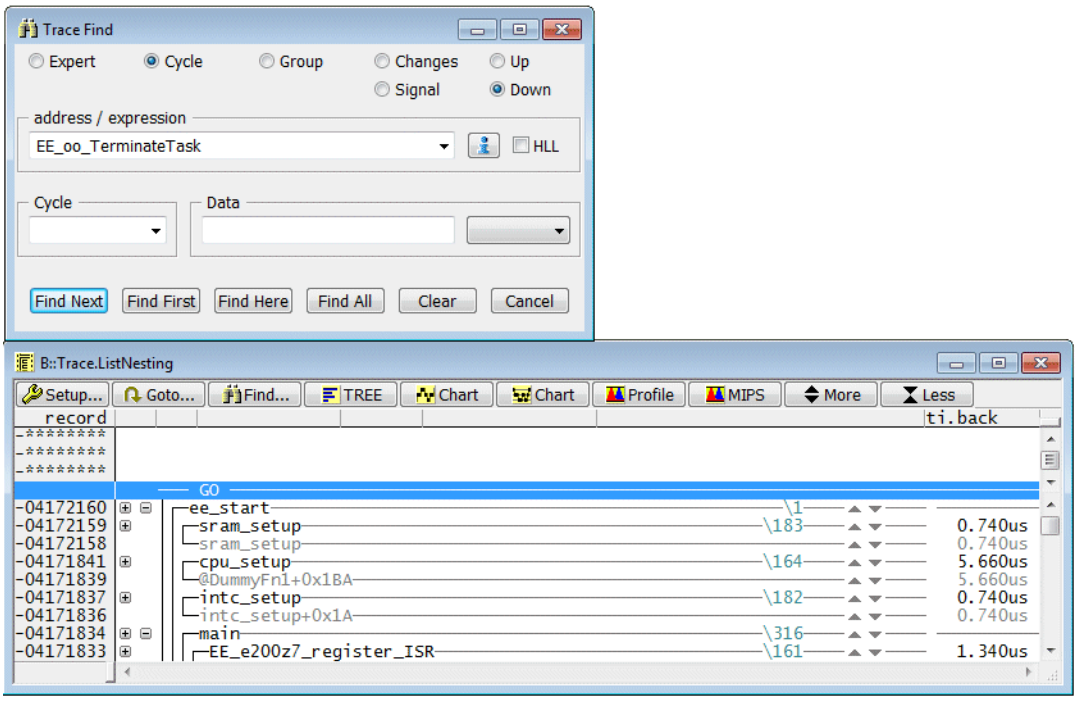


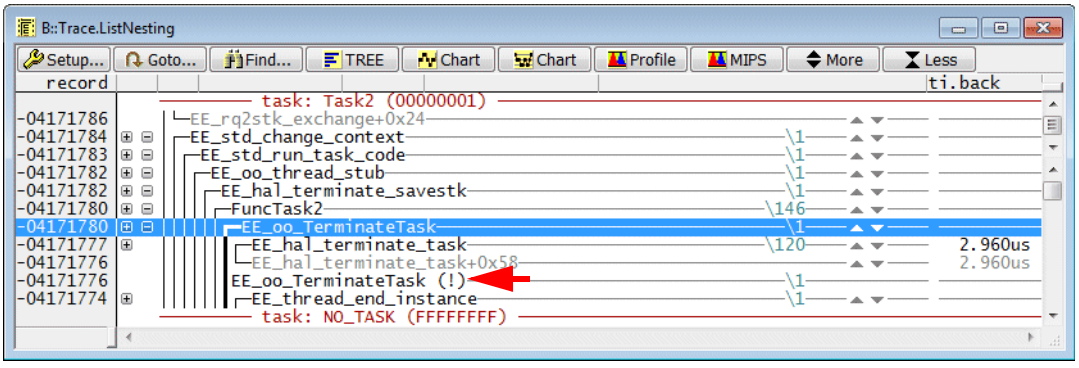
The following two TRACE32 windows are recommended if you want to inspect the issues:

```
Trace.ListNesting

Trace.List List.TASK List.ADDRESS List.sYmbol DEFault /Track
```

Example 1: We inspect the problem with the function EE_oo_TerminateTask.

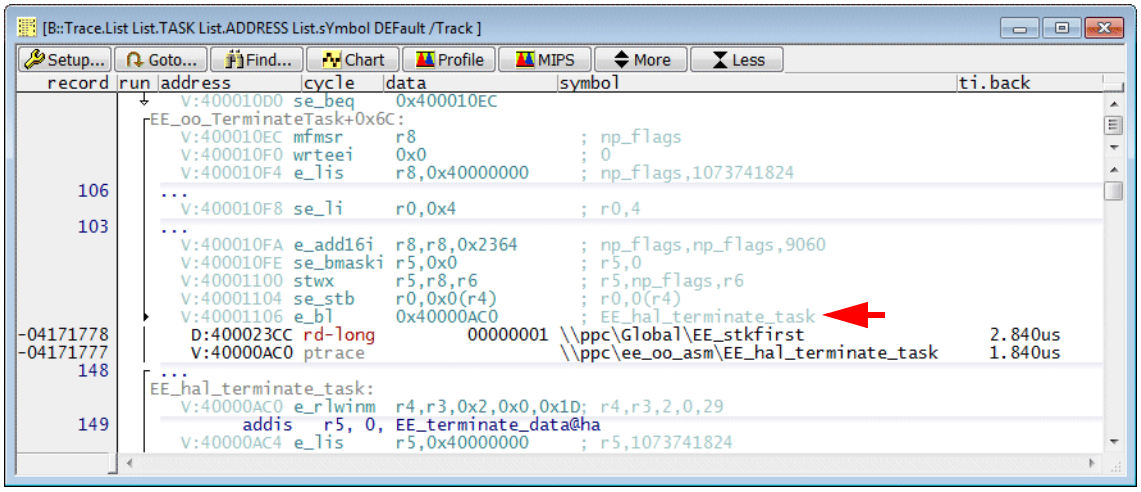
We start to search for the entry to the function EE_oo_TerminateTask in the task Task2 in the **Trace.ListNesting** window.



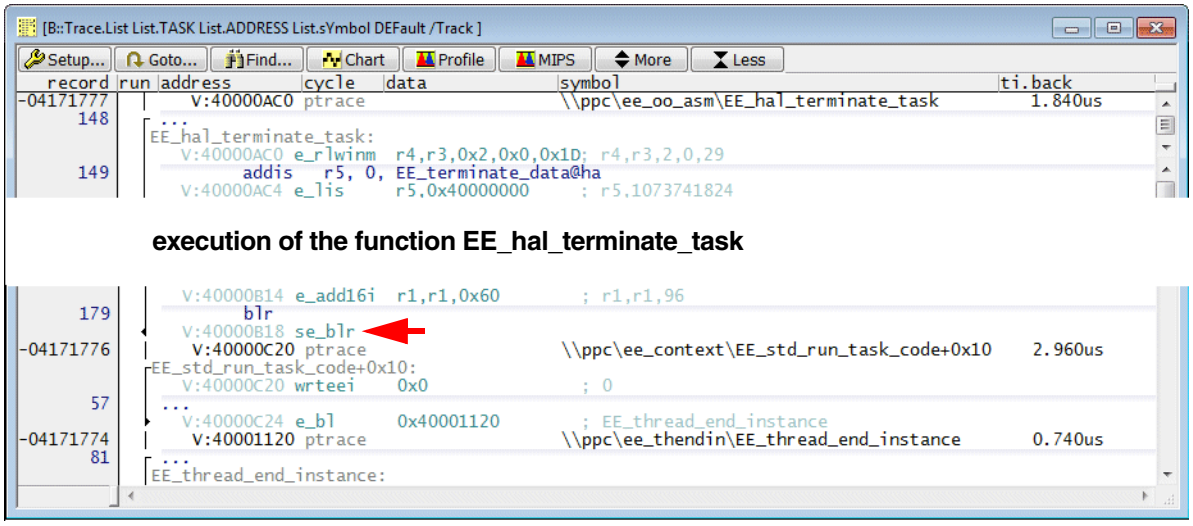


In the screenshot above we can see that the exit from the function EE_oo_TerminateTask is marked as a problem. Why is that?

Let's examine the function EE_oo_TerminateTask by looking to the trace listing.

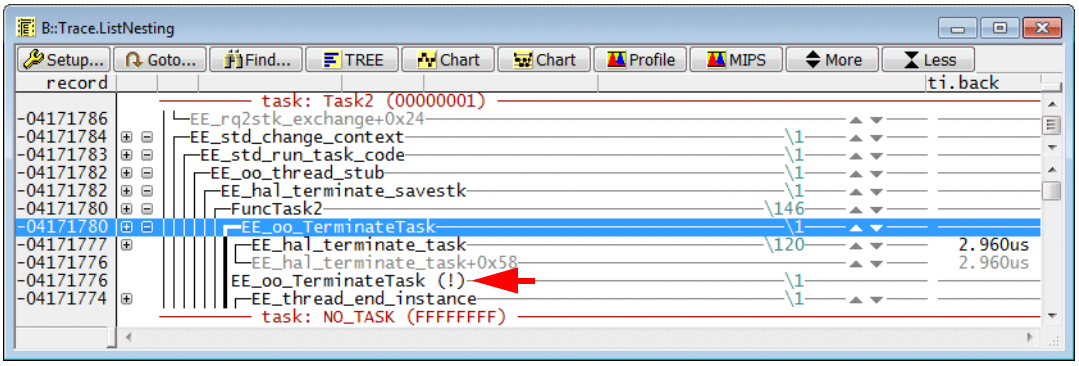


In its execution the function EE_oo_TerminateTask calls the function EE_hal_terminate_task.



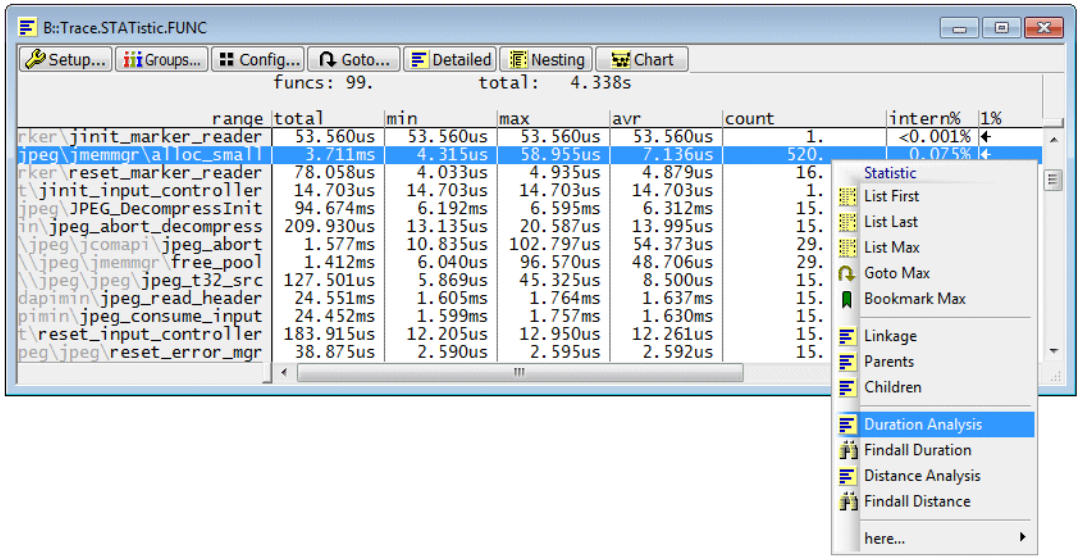
Now one would expect the function EE_hal_terminate_task returns with the se_blr instruction to the calling function (which was EE_oo_TerminateTask). But if we look at the trace listing, we see that the program execution continued in the middle of the function EE_std_run_task_code.

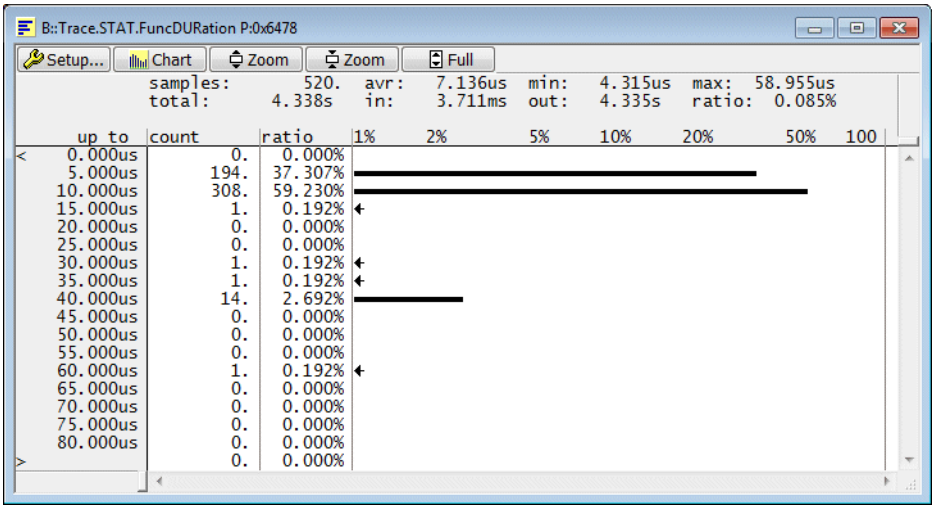
But since function EE_std_run_task_code+0x10 does not fit into the call tree and the function EE_oo_TerminateTask does not continue later in the trace recording, TRACE32 adds the function exit of EE_oo_TerminateTask to the call tree and marks it with !.



Look and Feel (No OS)

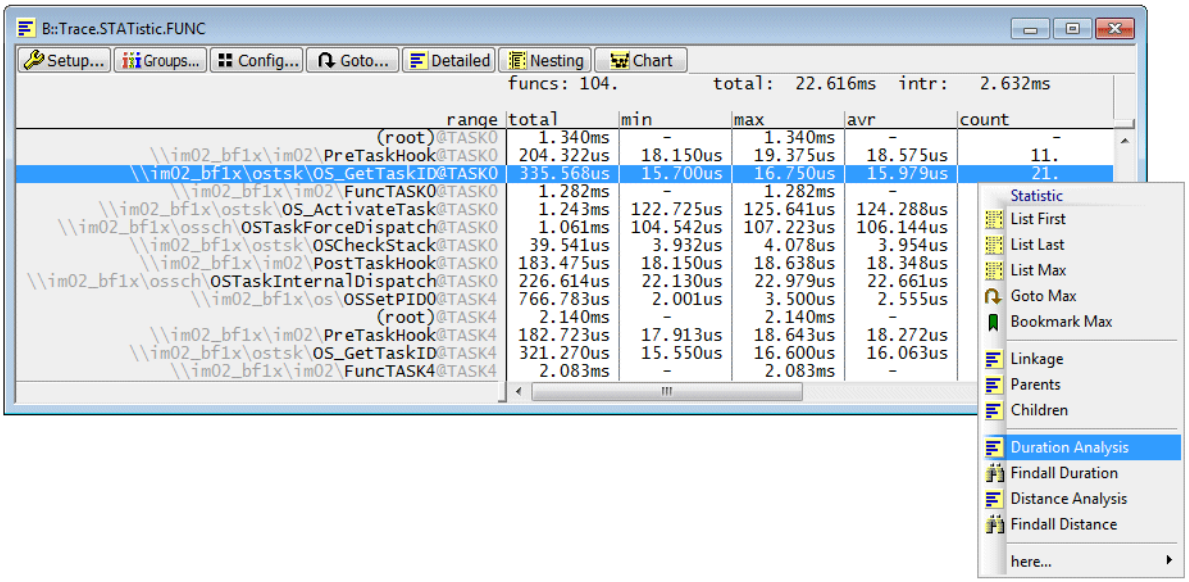
Trace.STATistic.FuncDURation *<function>* Detailed analysis of a single function, time between function entry and exit, time spent in interrupt service routines is excluded.

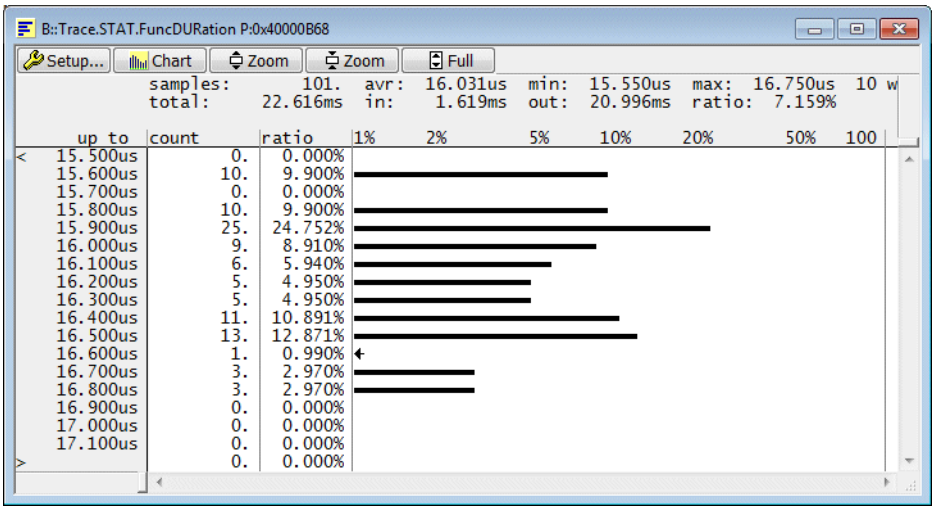




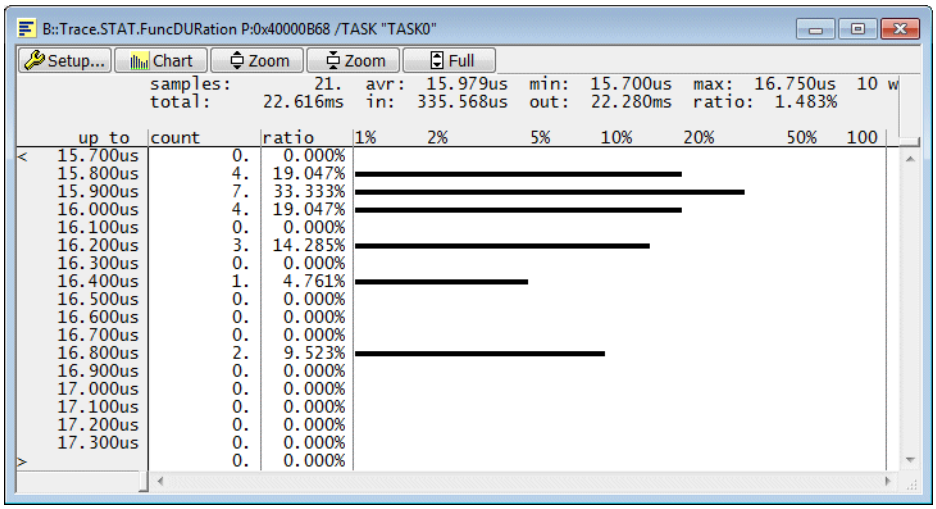
Detailed analysis of a single function, time between function entry and exit, time spent in interrupt service routines and other tasks is excluded.

Trace.STATistic.FuncDURation *<function>* [**/TASK** *"<task_name>"*]



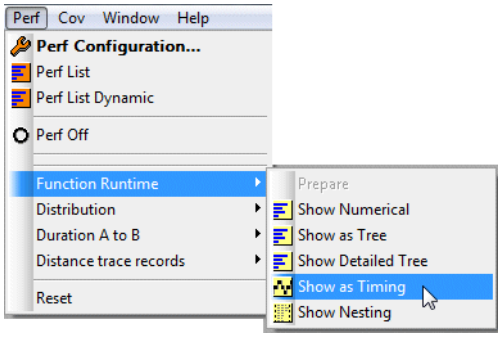


Please be aware, that details are shown for all function runs. If you are interested in a task-specific analysis, you have to use the **/TASK** *"<task_name>"* option.

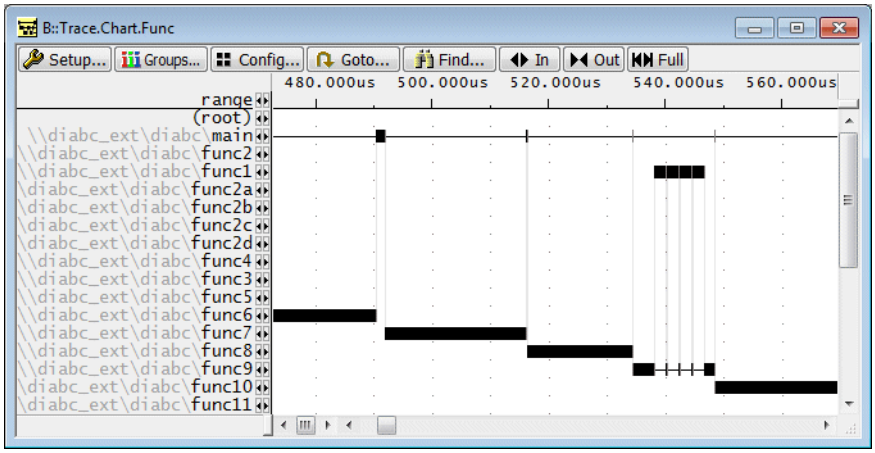


Trace.Chart.Func

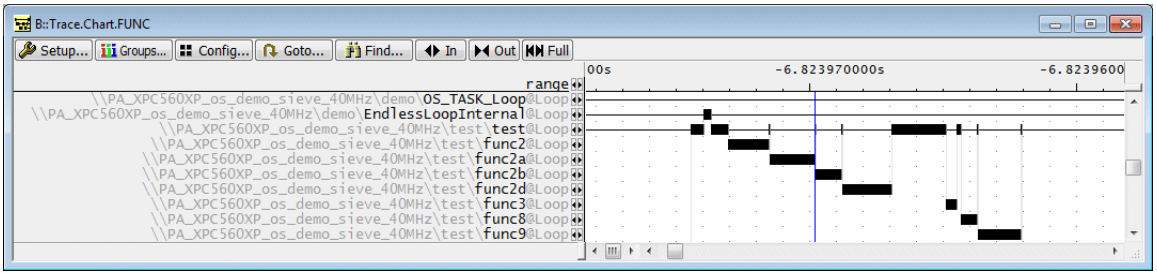
Nested function run-time analysis
- graphical display



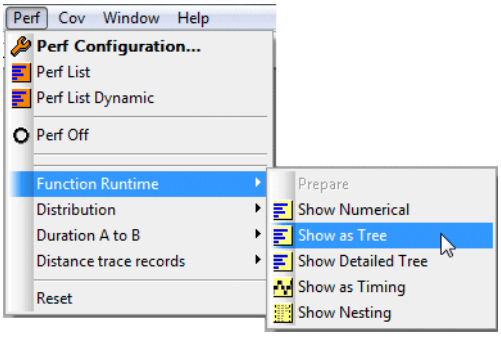
Look and Feel (No OS)



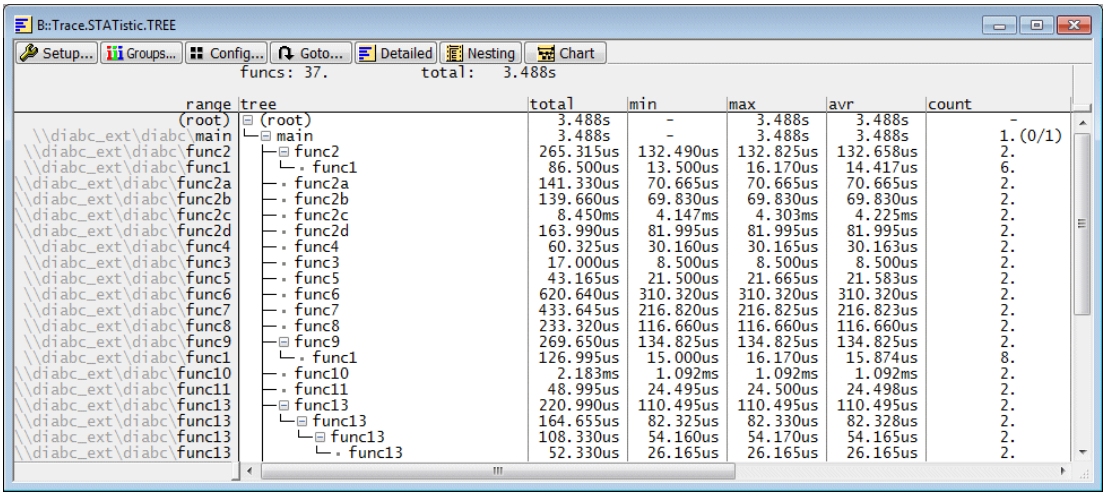
Look and Feel (OS)



Trace.STATistic.TREE Nested function run-time analysis
- tree display

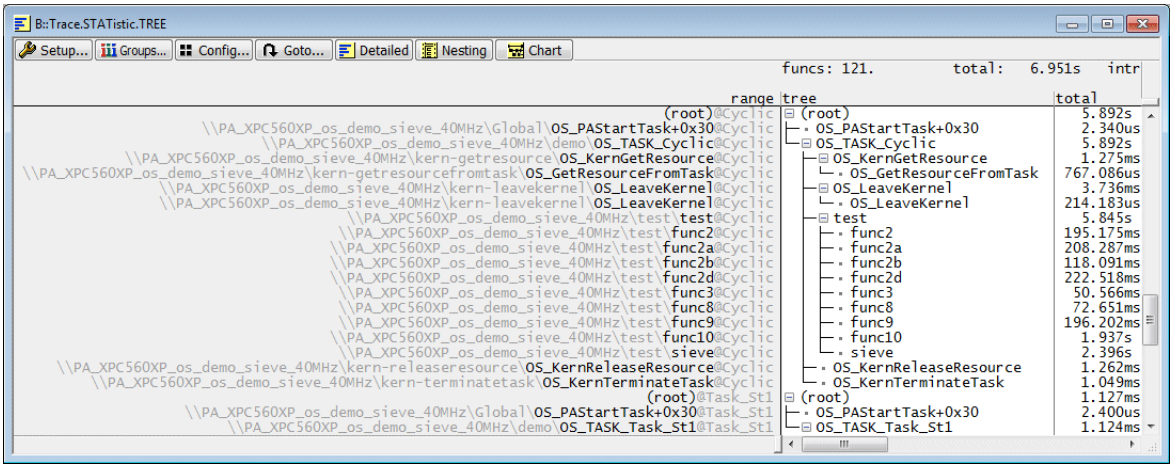


Look and Feel (No OS)



range	tree	total	min	max	avr	count
(root)	(root)	3.488s	-	3.488s	3.488s	-
\\diabc_ext\diabc\main	main	3.488s	-	3.488s	3.488s	1.(0/1)
\\diabc_ext\diabc\func2	func2	265.315us	132.490us	132.825us	132.658us	2.
\\diabc_ext\diabc\func1	func1	86.500us	13.500us	16.170us	14.417us	6.
\\diabc_ext\diabc\func2a	func2a	141.330us	70.665us	70.665us	70.665us	2.
\\diabc_ext\diabc\func2b	func2b	139.660us	69.830us	69.830us	69.830us	2.
\\diabc_ext\diabc\func2c	func2c	8.450ms	4.147ms	4.303ms	4.225ms	2.
\\diabc_ext\diabc\func2d	func2d	163.990us	81.995us	81.995us	81.995us	2.
\\diabc_ext\diabc\func4	func4	60.325us	30.160us	30.165us	30.163us	2.
\\diabc_ext\diabc\func3	func3	17.000us	8.500us	8.500us	8.500us	2.
\\diabc_ext\diabc\func5	func5	43.165us	21.500us	21.665us	21.583us	2.
\\diabc_ext\diabc\func6	func6	620.640us	310.320us	310.320us	310.320us	2.
\\diabc_ext\diabc\func7	func7	433.645us	216.820us	216.825us	216.823us	2.
\\diabc_ext\diabc\func8	func8	233.320us	116.660us	116.660us	116.660us	2.
\\diabc_ext\diabc\func9	func9	269.650us	134.825us	134.825us	134.825us	2.
\\diabc_ext\diabc\func1	func1	126.995us	15.000us	16.170us	15.874us	8.
\\diabc_ext\diabc\func10	func10	2.183ms	1.092ms	1.092ms	1.092ms	2.
\\diabc_ext\diabc\func11	func11	48.995us	24.495us	24.500us	24.498us	2.
\\diabc_ext\diabc\func13	func13	220.990us	110.495us	110.495us	110.495us	2.
\\diabc_ext\diabc\func13	func13	164.655us	82.325us	82.330us	82.328us	2.
\\diabc_ext\diabc\func13	func13	108.330us	54.160us	54.170us	54.165us	2.
\\diabc_ext\diabc\func13	func13	52.330us	26.165us	26.165us	26.165us	2.

Look and Feel (OS)

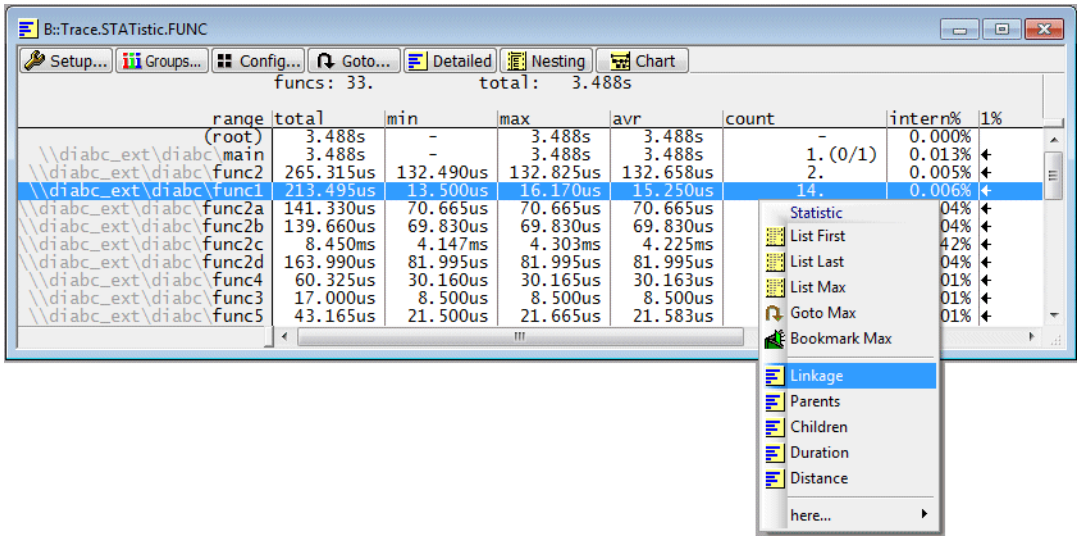


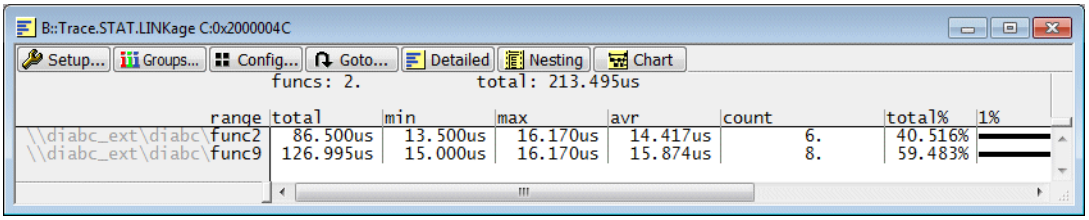
It is also possible to get a task-specific tree.

```
Trace.STATistic.TREE /TASK "Cyclic"
```

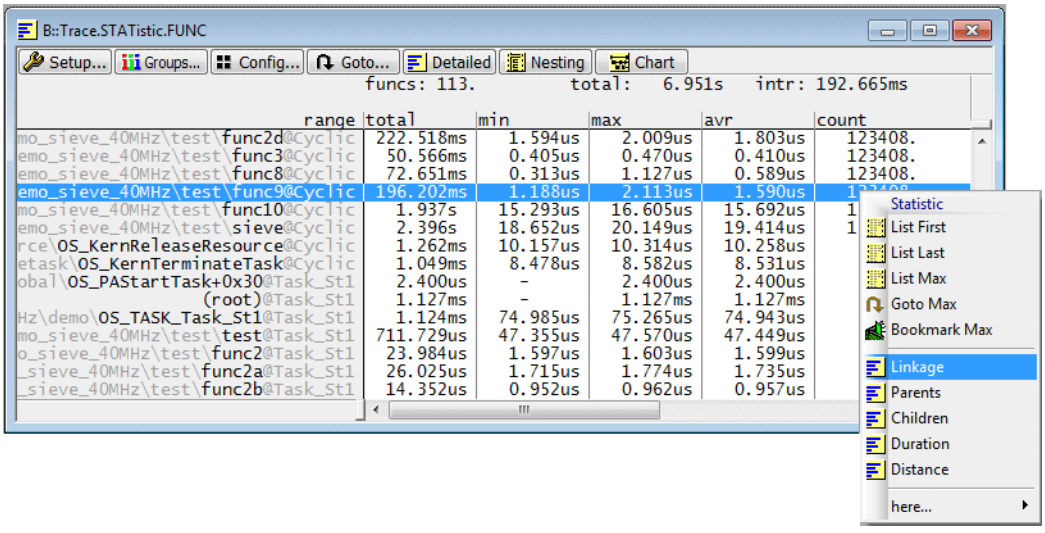
Trace.STATistic.LINKage *<address>* Nested function run-time analysis
- linkage analysis

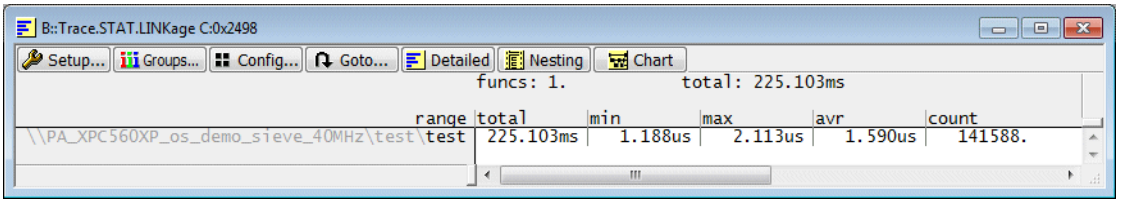
Look and Feel (No OS)





Look and Feel (OS)





Third-party Timing Tools

TRACE32 also provides an interface to third-party timing tools. For details refer to **"Trace Export for Third-Party Timing Tools"** (app_timing_tools.pdf).

Function Run-Times Analysis - SMP Instance

This chapter applies for SMP TRACE32 instances.

Flat Analysis

It is recommended to reduce the trace information generated by NEXUS to the required minimum.

- To avoid an overload of the NEXUS port.
- To make best use of the available trace memory.
- To get a more accurate timestamp.

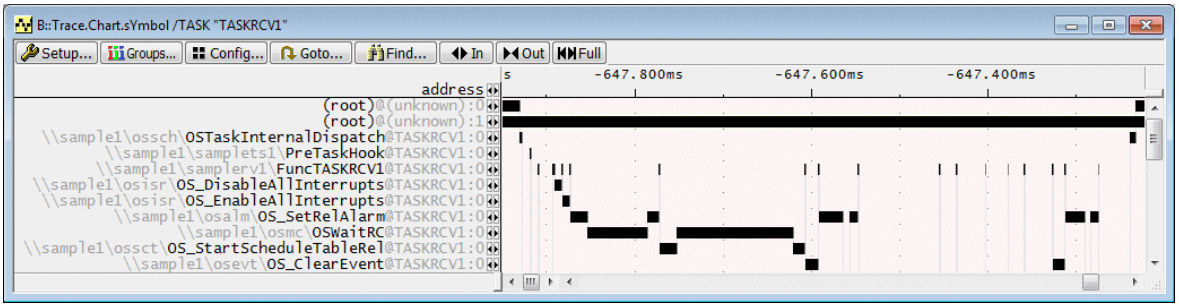
Optimum NEXUS Configuration (No OS)

Flat function run-time analysis does **not** require any **data information** if no OS is used. That's why it is recommended to switch the broadcasting of data information off.

```
NEXUS.DTM OFF
```

Your function time chart **can** include task information if you advise NEXUS to export the instruction flow and task switches. For details refer to the chapter **OS-Aware Tracing** of this training.

```
Trace.Chart.sYmbol /TASK "TASKRCV1"
```



Optimum Configuration 1 (if OSEK generated OTMs)**:**

```
NEXUS.OTM ON
```

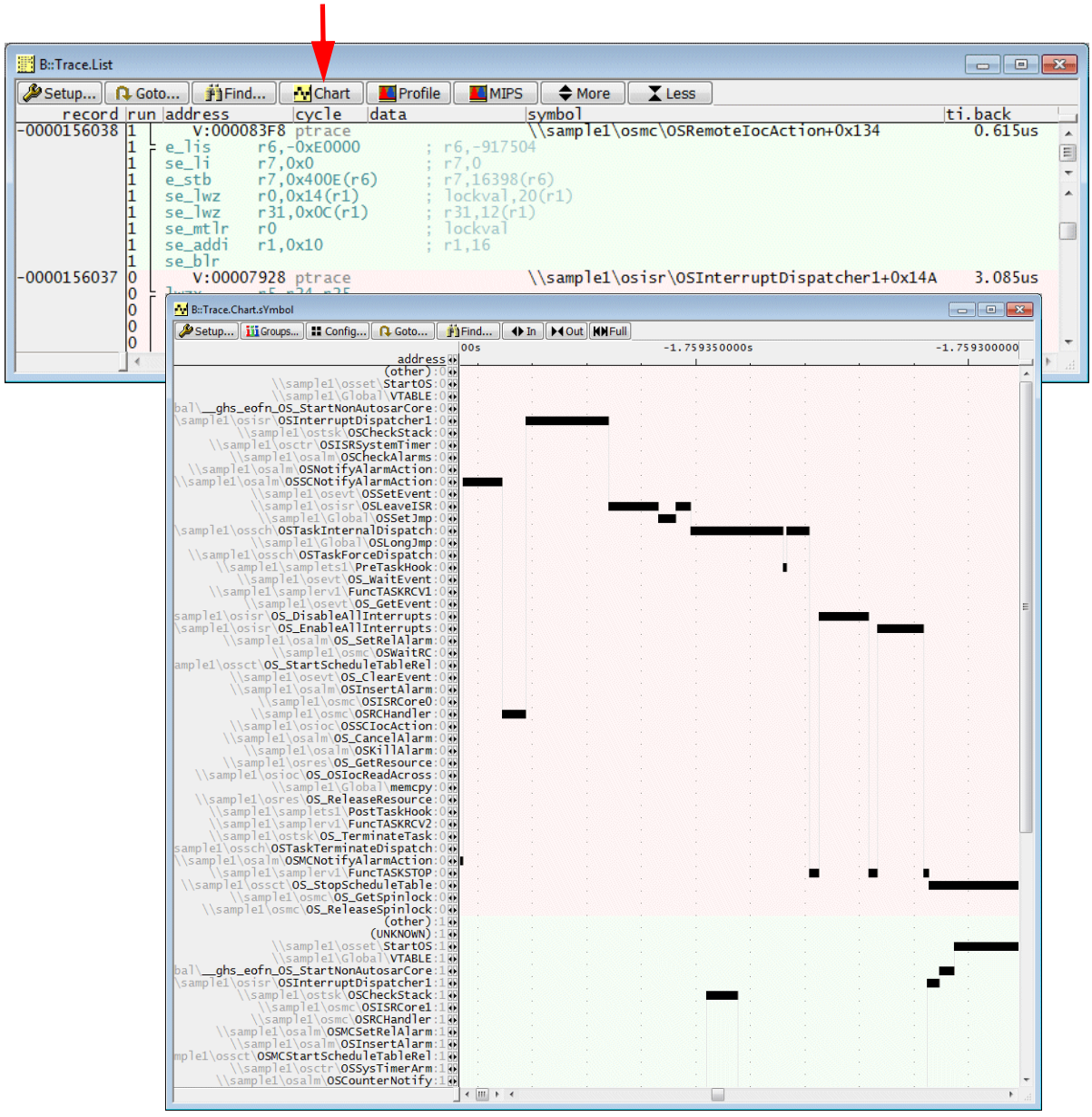
Optimum Configuration 2 (if OSEK does not support OTMs, NEXUS class 3 only)**:**

```
Break.Set TASK.CONFIG(magic[0]) /Write /TraceData
Break.Set TASK.CONFIG(magic[1]) /Write /TraceData
…
```

Function Timing Diagram

TRACE32 PowerView provides a timing diagram which shows when the program counter was in which function/symbol range.

Pushing the **Chart** button in the **Trace.List** window opens a **Trace.Chart.sYmbol** window

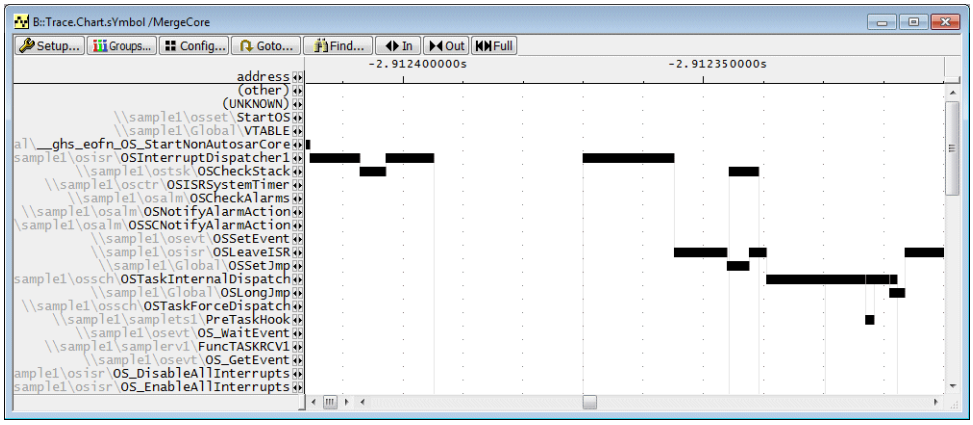


Trace.Chart.sYmbol [/SplitCore /Sort CoreTogether]

Flat function run-time analysis
- graphical display
- split the result per core
- sort results per core and then per recording order

Trace.Chart.sYmbol [/SplitCore] /Sort CoreSeparated

Flat function run-time analysis
- graphical display
- split the result per core
- sort the results per recording order

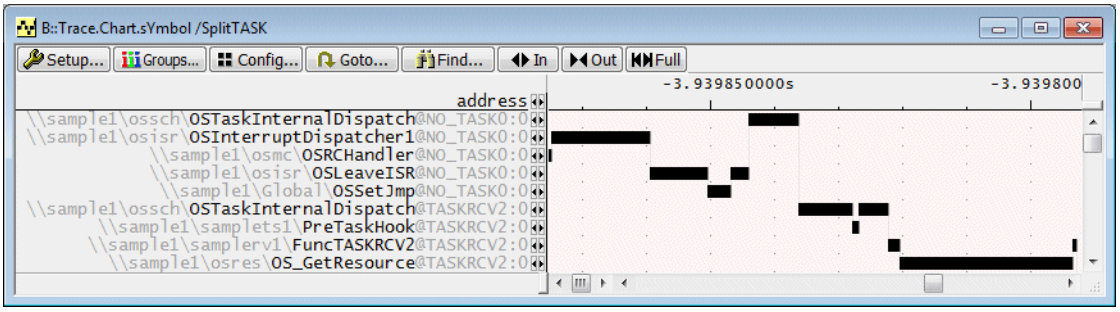


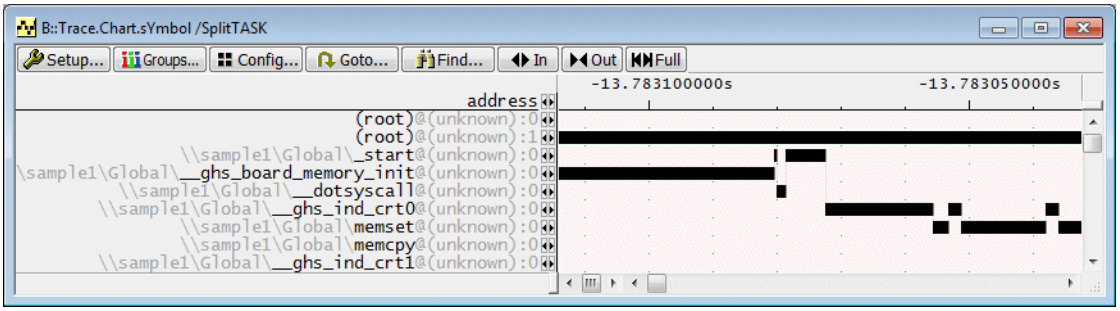
Trace.Chart.sYmbol /MergeCore

Flat function run-time analysis
- graphical display
- merge the results of all cores

Default setting
Trace.Chart.sYmbol [/MergeTASK] [/SplitCore /Sort CoreTogether]

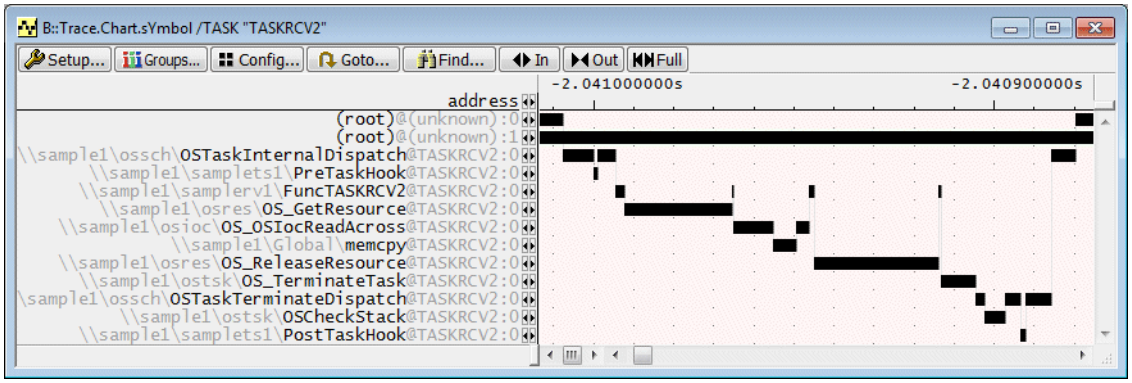
Display function time chart with task information
Trace.Chart.sYmbol /SplitTASK [/SplitCore /Sort CoreTogether]



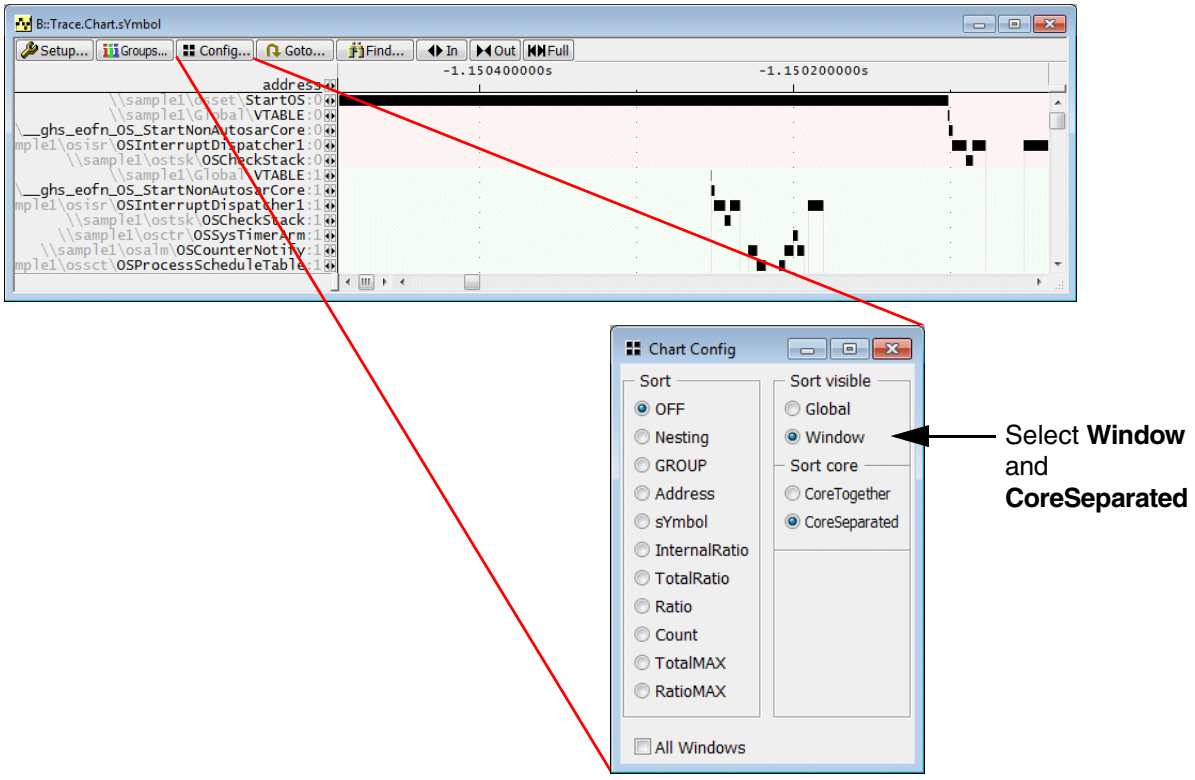


@*<task_name>*	Task name information
@(unknown)	• Function was running before the OS was started • Function was recorded before first task switch information was recorded
(root)@(unknown)	No trace information available

Display function time chart for the specified task
Trace.Chart.sYmbol /TASK *<task_name>* **[/SplitCore /Sort CoreTogether]**



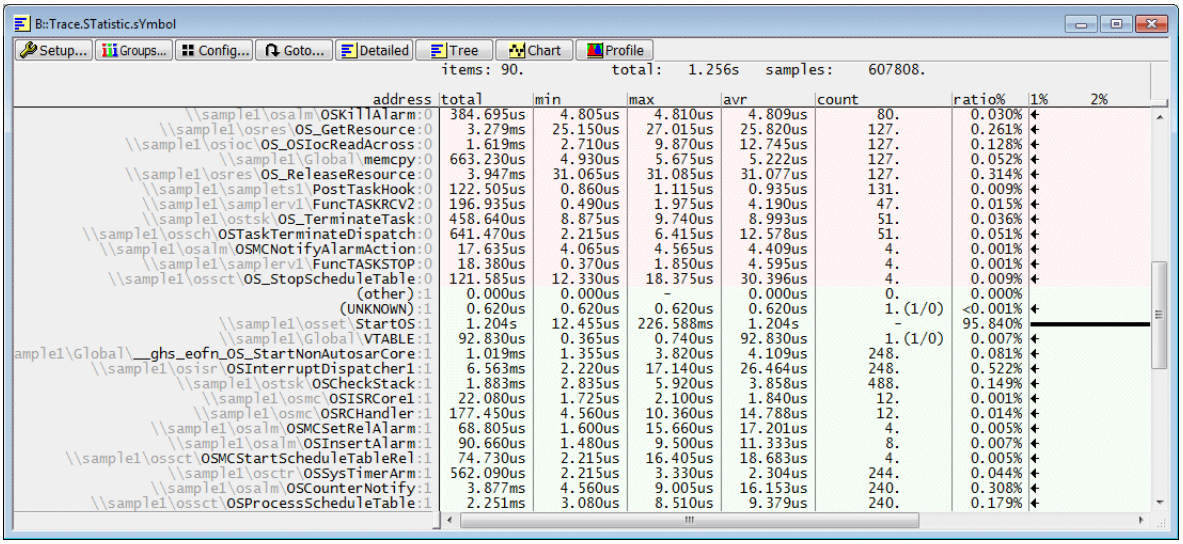
@*<task_name>*	Functions running while the specified task was running
(root)@(unknown)	All other trace information



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

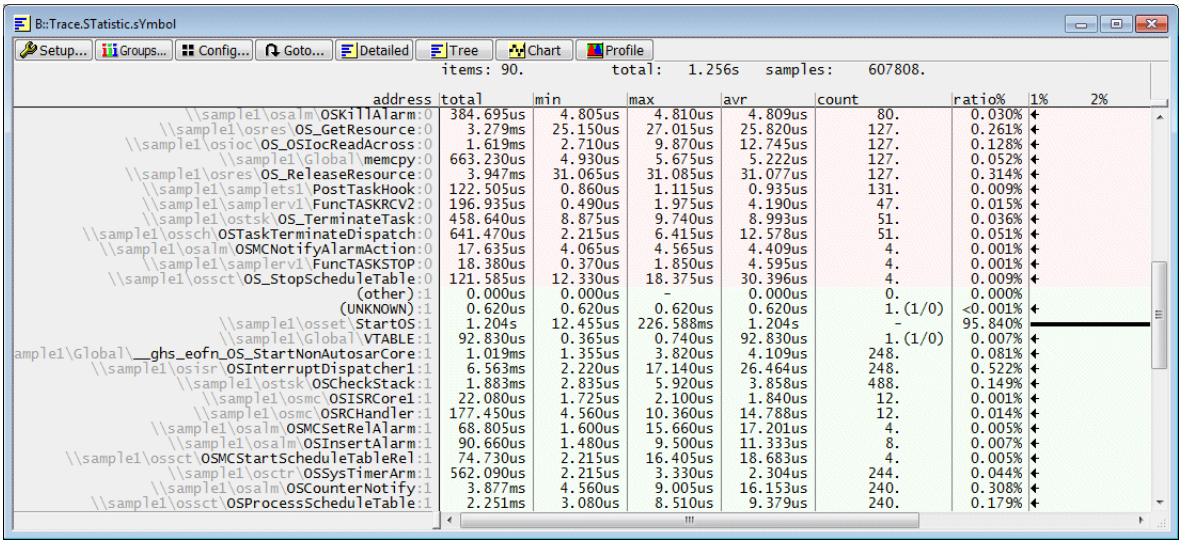
For a detailed description of all **Sort** options provided by the **Chart Config** window refer to the command description of **Trace.STATistic.Sort**.

Analog to the timing diagram also a numerical analysis is provided.



address	total	min	max	avr	count	ratio%
\\sample1\osalm\OSKillAlarm:0	384.695us	4.805us	4.810us	4.809us	80.	0.030%
\\sample1\osres\OS_GetResource:0	3.279ms	25.150us	27.015us	25.820us	127.	0.261%
\\sample1\osioc\OS_OSIocReadAcross:0	1.619ms	2.710us	9.870us	12.745us	127.	0.128%
\\sample1\Global\memcpy:0	663.230us	4.930us	5.675us	5.222us	127.	0.052%
\\sample1\osres\OS_ReleaseResource:0	3.947ms	31.065us	31.085us	31.077us	127.	0.314%
\\sample1\samplets1\PostTaskHook:0	122.505us	0.860us	1.115us	0.935us	131.	0.009%
\\sample1\samplerv1\FuncTASKRCV2:0	196.935us	0.490us	1.975us	4.190us	47.	0.015%
\\sample1\ostsk\OS_TerminateTask:0	458.640us	8.875us	9.740us	8.993us	51.	0.036%
\\sample1\ossch\OSTaskTerminateDispatch:0	641.470us	2.215us	6.415us	12.578us	51.	0.051%
\\sample1\osalm\OSMCNotifyAlarmAction:0	17.635us	4.065us	4.565us	4.409us	4.	0.001%
\\sample1\samplerv1\FuncTASKSTOP:0	18.380us	0.370us	1.850us	4.595us	4.	0.001%
\\sample1\ossct\OS_StopScheduleTable:0	121.585us	12.330us	18.375us	30.396us	4.	0.009%
(other):1	0.000us	0.000us	-	0.000us	0.	0.000%
(UNKNOWN):1	0.620us	0.620us	0.620us	0.620us	1.(1/0)	<0.001%
\\sample1\osset\StartOS:1	1.204s	12.455us	226.588ms	1.204s	-	95.840%
\\sample1\Global\VTABLE:1	92.830us	0.365us	0.740us	92.830us	1.(1/0)	0.007%
ample1\Global__ghs_eofn_OS_StartNonAutosarCore:1	1.019ms	1.355us	3.820us	4.109us	248.	0.081%
\\sample1\osisr\OSInterruptDispatcher1:1	6.563ms	2.220us	17.140us	26.464us	248.	0.522%
\\sample1\ostsk\OSCheckStack:1	1.883ms	2.835us	5.920us	3.858us	488.	0.149%
\\sample1\osmc\OSISRCore1:1	22.080us	1.725us	2.100us	1.840us	12.	0.001%
\\sample1\osmc\OSRCHandler:1	177.450us	4.560us	10.360us	14.788us	12.	0.014%
\\sample1\osalm\OSMCSetRelAlarm:1	68.805us	1.600us	15.660us	17.201us	4.	0.005%
\\sample1\osalm\OSInsertAlarm:1	90.660us	1.480us	9.500us	11.333us	8.	0.007%
\\sample1\ossct\OSMCStartScheduleTableRel:1	74.730us	2.215us	16.405us	18.683us	4.	0.005%
\\sample1\osctr\OSSysTimerArm:1	562.090us	2.215us	3.330us	2.304us	244.	0.044%
\\sample1\osalm\OSCounterNotify:1	3.877ms	4.560us	9.005us	16.153us	240.	0.308%
\\sample1\ossct\OSProcessScheduleTable:1	2.251ms	3.080us	8.510us	9.379us	240.	0.179%

survey	
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)



function details	
address	function/symbol region name **(other)** program sections that can not be assigned to a function/symbol region **(UNKNOWN)** program sections that can not be decoded
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 as total/count)
count	number of new entries (start address executed) into the address range of the function/symbol region
ratio	ratio of time in the function/symbol region with regards to the total time period recorded

Trace.STATistic.sYmbol /MergeCORE	Flat function run-time analysis - numerical display - merge the results of all cores
Trace.STATistic.sYmbol /Sort CoreSeparated	Flat function run-time analysis - numerical display - split the result per core - sort the results per recording order

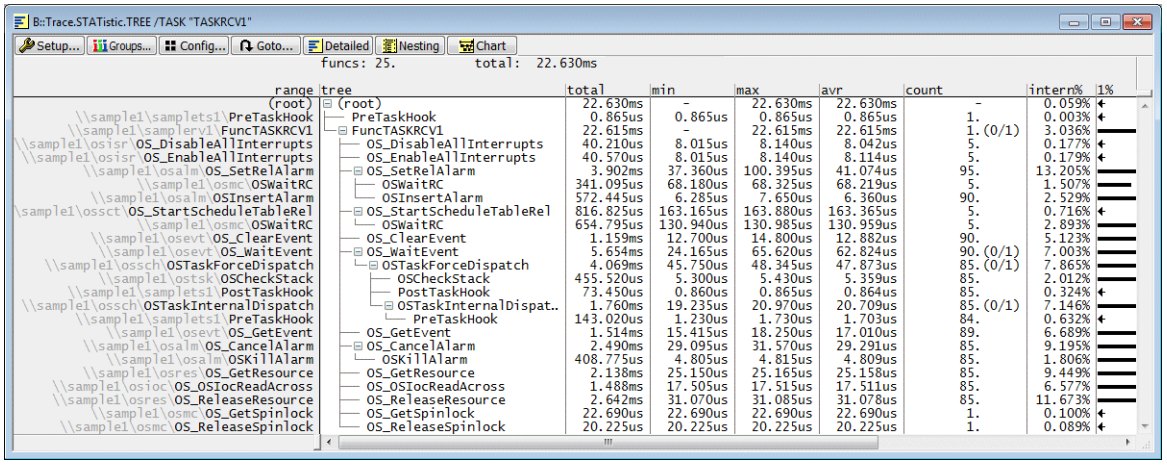
Trace.STATistic.sYmbol [**/MergeTASK**]	Flat function run-time analysis (OS) - numerical display - no task information
Trace.STATistic.sYmbol /SplitTASK	Flat function run-time analysis (OS) - numerical display including task information
Trace.STATistic.sYmbol /TASK *<task_name>*	Flat function run-time analysis (OS) - numerical display for specified task

Restrictions

1. **The nesting analysis analyses only high-level language functions.**
2. **The nested function run-time analysis expects common ways to enter/exit functions.**
3. **The nesting analysis is sensitive with regards to FIFOFULLs.**

TRACE32 PowerView builds up a separate call tree for each task.

```
Trace.STATistic.TREE /TASK "TASKRCV1"
```



range	tree	total	min	max	avr	count	interr%	1%
(root)	(root)	22.630ms	-	22.630ms	22.630ms	-	0.059%	
\\sample1\samplets1\PreTaskHook	PreTaskHook	0.865us	0.865us	0.865us	0.865us	1.	0.003%	
\\sample1\samplerv1\FuncTASKRCV1	FuncTASKRCV1	22.615ms	-	22.615ms	22.615ms	1.(0/1)	3.036%	
\\sample1\osisr\OS_DisableAllInterrupts	OS_DisableAllInterrupts	40.210us	8.015us	8.140us	8.042us	5.	0.177%	
\\sample1\osisr\OS_EnableAllInterrupts	OS_EnableAllInterrupts	40.570us	8.015us	8.140us	8.114us	5.	0.179%	
\\sample1\osalm\OS_SetRelAlarm	OS_SetRelAlarm	3.902ms	37.360us	100.395us	41.074us	95.	13.205%	
\\sample1\osmc\OSWaitRC	OSWaitRC	341.095us	68.180us	68.325us	68.219us	5.	1.507%	
\\sample1\osalm\OSInsertAlarm	OSInsertAlarm	572.445us	6.285us	7.650us	6.360us	90.	2.529%	
\sample1\ossct\OS_StartScheduleTableRel	OS_StartScheduleTableRel	816.825us	163.165us	163.880us	163.365us	5.	0.716%	
\\sample1\osmc\OSWaitRC	OSWaitRC	654.795us	130.940us	130.985us	130.959us	5.	2.893%	
\\sample1\osevt\OS_ClearEvent	OS_ClearEvent	1.159ms	12.700us	14.800us	12.882us	90.	5.123%	
\\sample1\osevt\OS_WaitEvent	OS_WaitEvent	5.654ms	24.165us	65.620us	62.824us	90.(0/1)	7.003%	
\\sample1\ossch\OSTaskForceDispatch	OSTaskForceDispatch	4.069ms	45.750us	48.345us	47.873us	85.(0/1)	7.865%	
\\sample1\ostsk\OSCheckStack	OSCheckStack	455.520us	5.300us	5.430us	5.359us	85.	2.012%	
\\sample1\samplets1\PostTaskHook	PostTaskHook	73.450us	0.860us	0.865us	0.864us	85.	0.324%	
\\sample1\ossch\OSTaskInternalDispatch	OSTaskInternalDispat..	1.760ms	19.235us	20.970us	20.709us	85.(0/1)	7.146%	
\\sample1\samplets1\PreTaskHook	PreTaskHook	143.020us	1.230us	1.730us	1.703us	84.	0.632%	
\\sample1\osevt\OS_GetEvent	OS_GetEvent	1.514ms	15.415us	18.250us	17.010us	89.	6.689%	
\\sample1\osalm\OS_CancelAlarm	OS_CancelAlarm	2.490ms	29.095us	31.570us	29.291us	85.	9.195%	
\\sample1\osalm\OSKillAlarm	OSKillAlarm	408.775us	4.805us	4.815us	4.809us	85.	1.806%	
\\sample1\osres\OS_GetResource	OS_GetResource	2.138ms	25.150us	25.165us	25.158us	85.	9.449%	
\\sample1\osioc\OS_OSIocReadAcross	OS_OSIocReadAcross	1.488ms	17.505us	17.515us	17.511us	85.	6.577%	
\\sample1\osres\OS_ReleaseResource	OS_ReleaseResource	2.642ms	31.070us	31.085us	31.078us	85.	11.673%	
\\sample1\osmc\OS_GetSpinlock	OS_GetSpinlock	22.690us	22.690us	22.690us	22.690us	1.	0.100%	
\\sample1\osmc\OS_ReleaseSpinlock	OS_ReleaseSpinlock	20.225us	20.225us	20.225us	20.225us	1.	0.089%	

In order to hook a function entry/exit into the correct call tree, TRACE32 PowerView needs to know which task was running when the entry/exit occurred.

The standard way to get information on the current task is to advise the NEXUS to export the instruction flow and task switches. For details refer to the chapter **OS-Aware Tracing** of this training.

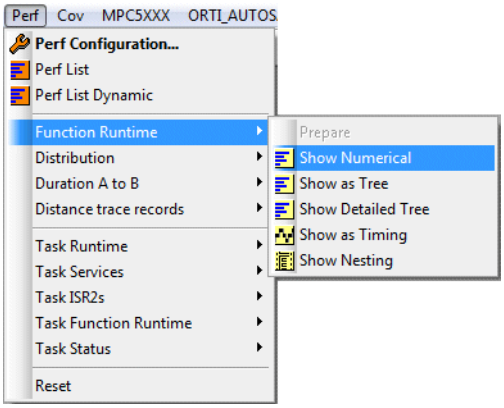
Optimum Configuration 1 (if OSEK generated OTMs)**:**

```
NEXUS.OTM ON

Trace.STATistic.InterruptIsFunction ON
```

Optimum Configuration 2 (if OSEK does not support OTMs, NEXUS class 3 only)**:**

```
Break.Set TASK.CONFIG(magic[0]) /Write /TraceData
Break.Set TASK.CONFIG(magic[1]) /Write /TraceData
…

Trace.STATistic.InterruptIsFunction ON
```

Trace.STATistic.Func Nested function run-time analysis
- numeric display



B::Trace.STATistic.FUNC

Setup... Groups... Config... Goto... Detailed Nesting Chart

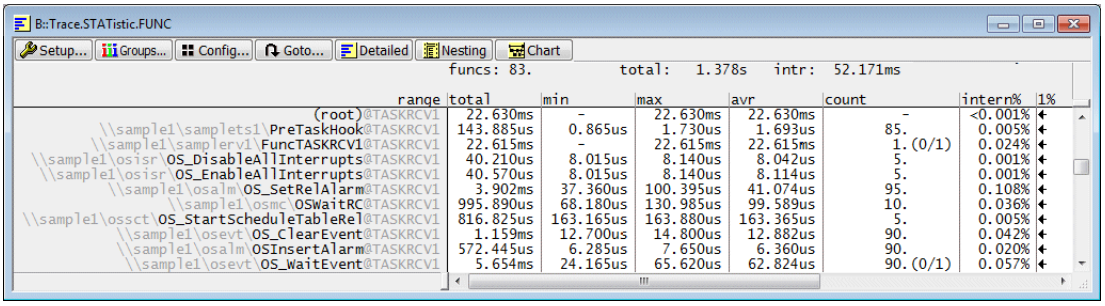
funcs: 83. total: 1.378s intr: 52.171ms

range	total	min	max	avr	count	intern%	1%
(root)@(unknown):0	5.542ms	-	5.542ms	5.542ms	-	0.112%	
(root)@(unknown):1	1.378s	-	1.378s	1.378s	-	50.000%	
\\sample1\samplerv1\main@(unknown):0	2.436ms	-	2.436ms	2.436ms	1.(0/1)	<0.001%	
\\sample1\osmc\StartCore@(unknown):0	9.865us	9.865us	9.865us	9.865us	1.	<0.001%	
\\sample1\osset\StartOS@(unknown):0	2.414ms	-	2.414ms	2.414ms	1.(0/1)	0.007%	
e1\osset\OSInitApplications@(unknown):0	28.980us	28.980us	28.980us	28.980us	1.	0.001%	
mple1\osisr\OSInitializeISR@(unknown):0	2.014ms	2.014ms	2.014ms	2.014ms	1.	0.073%	
le1\osctr\OSInitSystemTimer@(unknown):0	3.575us	3.575us	3.575us	3.575us	1.	<0.001%	
ample1\osmc\OSWaitCoreSynch@(unknown):0	10.235us	3.945us	6.290us	5.118us	2.	<0.001%	
\\sample1\ostsk\OSInitTasks@(unknown):0	68.200us	68.200us	68.200us	68.200us	1.	0.002%	
mple1\osres\OSInitResources@(unknown):0	20.840us	20.840us	20.840us	20.840us	1.	<0.001%	
ample1\osctr\OSInitCounters@(unknown):0	10.605us	10.605us	10.605us	10.605us	1.	<0.001%	
\sample1\osalm\OSInitAlarms@(unknown):0	11.100us	11.100us	11.100us	11.100us	1.	<0.001%	
\ossct\OSInitScheduleTables@(unknown):0	3.580us	3.580us	3.580us	3.580us	1.	<0.001%	
sample1\osmc\OSInitSpinlock@(unknown):0	2.590us	2.590us	2.590us	2.590us	1.	<0.001%	
\\sample1\osioc\OSInitIOC@(unknown):0	11.350us	11.350us	11.350us	11.350us	1.	<0.001%	
ple1\sample1_cfg\OSIOC_Init@(unknown):0	0.865us	0.865us	0.865us	0.865us	1.	<0.001%	
e1\osctr\OSStartSystemTimer@(unknown):0	3.085us	3.085us	3.085us	3.085us	1.	<0.001%	
1\ossch\OSTaskForceDispatch@(unknown):0	21.830us	-	21.830us	21.830us	1.(0/1)	<0.001%	
\sample1\ostsk\OSCheckStack@(unknown):0	3.575us	3.575us	3.575us	3.575us	1.	<0.001%	
ssch\OSTaskInternalDispatch@(unknown):0	6.170us	-	6.170us	6.170us	1.(0/1)	<0.001%	
(root)@TASKRCV1	22.630ms	-	22.630ms	22.630ms	-	<0.001%	
\sample1\samplets1\PreTaskHook@TASKRCV1	143.885us	0.865us	1.730us	1.693us	85.	0.005%	
sample1\samplerv1\FuncTASKRCV1@TASKRCV1	22.615ms	-	22.615ms	22.615ms	1.(0/1)	0.024%	
\osisr\OS_DisableAllInterrupts@TASKRCV1	40.210us	8.015us	8.140us	8.042us	5.	0.001%	
1\osisr\OS_EnableAllInterrupts@TASKRCV1	40.570us	8.015us	8.140us	8.114us	5.	0.001%	
\\sample1\osalm\OS_SetRelAlarm@TASKRCV1	3.902ms	37.360us	100.395us	41.074us	95.	0.108%	
\\sample1\osmc\OSWaitRC@TASKRCV1	995.890us	68.180us	130.985us	99.589us	10.	0.036%	
ossct\OS_StartScheduleTableRel@TASKRCV1	816.825us	163.165us	163.880us	163.365us	5.	0.005%	
\\sample1\osevt\OS_ClearEvent@TASKRCV1	1.159ms	12.700us	14.800us	12.882us	90.	0.042%	
\\sample1\osalm\OSInsertAlarm@TASKRCV1	572.445us	6.285us	7.650us	6.360us	90.	0.020%	
\\sample1\osevt\OS_WaitEvent@TASKRCV1	5.654ms	24.165us	65.620us	62.824us	90.(0/1)	0.057%	
(root)@(interrupt):0	0.000us	-	-	0.000us	-	0.000%	
sr\OSInterruptDispatcher1@(interrupt):0	52.171ms	51.040us	105.080us	59.285us	880.(0/1)	0.964%	
ample1\ostsk\OSCheckStack@(interrupt):0	3.422ms	3.695us	6.170us	3.888us	880.	0.124%	
\\sample1\osmc\OSISRCore0@(interrupt):0	3.690ms	24.535us	28.490us	26.544us	139.	0.010%	
\sample1\osmc\OSRCHandler@(interrupt):0	3.407ms	22.440us	26.635us	24.508us	139.	0.075%	
mple1\osioc\OSSCIocAction@(interrupt):0	1.294ms	7.890us	10.730us	9.586us	135.	0.029%	
\sample1\osevt\OSSetEvent@(interrupt):0	496.865us	3.210us	5.800us	5.583us	89.	0.018%	
\\sample1\osevt\OS_GetEvent@TASKRCV1	1.514ms	15.415us	18.250us	17.010us	89.	0.054%	

- Task-specific function run-time analysis, core information is discarded.
- Functions that can not be assigned to a task are assigned to the (@unknown) task, per core display.
- Interrupt service routines are assigned to (@interrupt) task, per core display.

```
funcs: 83.          total:   1.378s   intr:  52.171ms
```

survey	
func	number of functions in the trace
total	total measurement time
intr	total time in interrupt service routines



range	total	min	max	avr	count	intern%	1%
(root)@TASKRCV1	22.630ms	-	22.630ms	22.630ms	-	<0.001%	
\\sample1\samplets1\PreTaskHook@TASKRCV1	143.885us	0.865us	1.730us	1.693us	85.	0.005%	
\\sample1\samplerv1\FuncTASKRCV1@TASKRCV1	22.615ms	-	22.615ms	22.615ms	1.(0/1)	0.024%	
\\sample1\osisr\OS_DisableAllInterrupts@TASKRCV1	40.210us	8.015us	8.140us	8.042us	5.	0.001%	
\\sample1\osisr\OS_EnableAllInterrupts@TASKRCV1	40.570us	8.015us	8.140us	8.114us	5.	0.001%	
\\sample1\osalm\OS_SetRelAlarm@TASKRCV1	3.902ms	37.360us	100.395us	41.074us	95.	0.108%	
\\sample1\osmc\OSWaitRC@TASKRCV1	995.890us	68.180us	130.985us	99.589us	10.	0.036%	
\\sample1\ossct\OS_StartScheduleTableRel@TASKRCV1	816.825us	163.165us	163.880us	163.365us	5.	0.005%	
\\sample1\osevt\OS_ClearEvent@TASKRCV1	1.159ms	12.700us	14.800us	12.882us	90.	0.042%	
\\sample1\osalm\OSInsertAlarm@TASKRCV1	572.445us	6.285us	7.650us	6.360us	90.	0.020%	
\\sample1\osevt\OS_WaitEvent@TASKRCV1	5.654ms	24.165us	65.620us	62.824us	90.(0/1)	0.057%	

columns	
range (NAME)	function name, sorted by their recording order as default

- **HLL function, task specific**

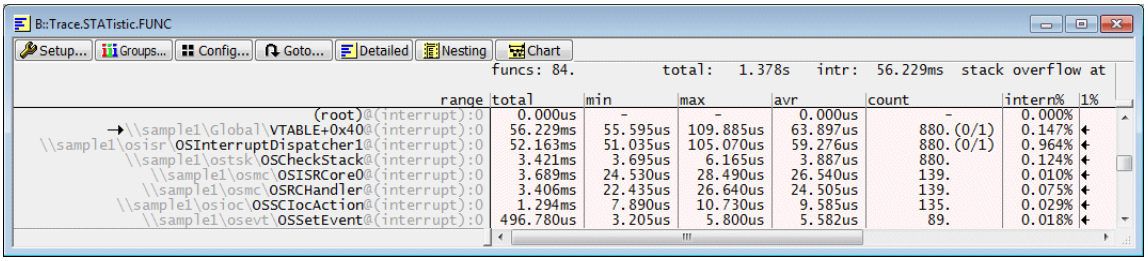
  ```
  \\sample1\osmc\OSWaitRC@TASKRCV1
  ```

  HLL function “OSWaitRC” running in task “TASKRCV1”

- **Root of task-specific call tree**

  ```
  (root)@TASKRCV1
  ```

  (root) of call tree for task TASKRCV1



- **Indirect branch into interrupt vector table**

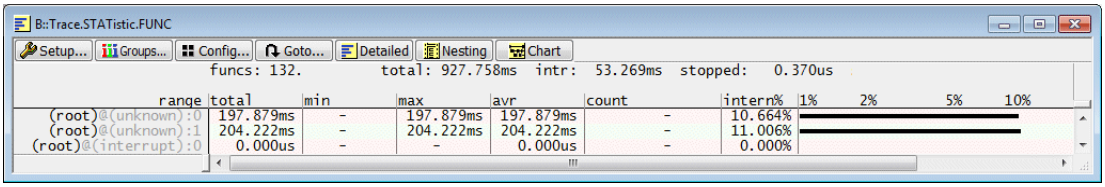
  →\\sample1\Global\VTABLE+0x40@(interrupt):0

- **Interrupt service function**

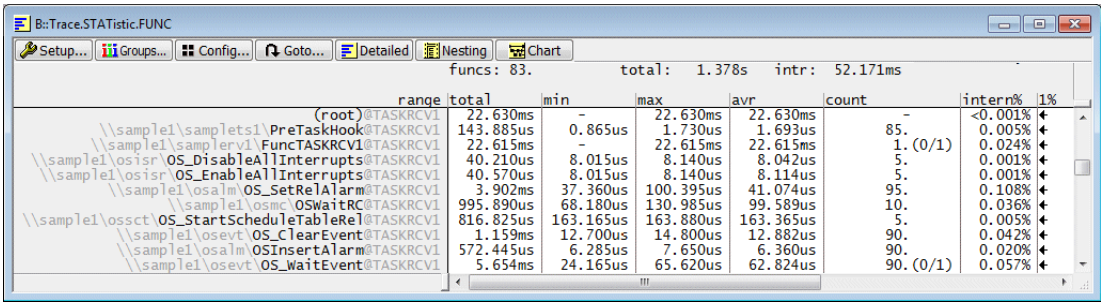
  \\sample1\osevt\OSSetEvent@(interrupt):0

- **Root of @(interrupt)**

  (root)@(interrupt):0

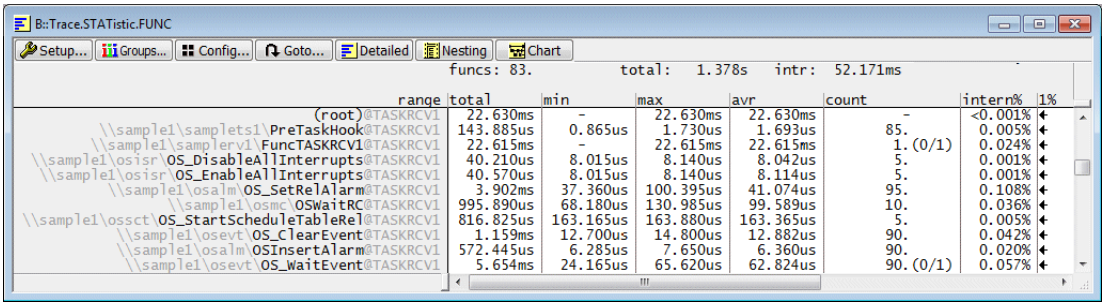


TRACE32 assigns all trace information generated before the first task switch to the @(unknown) task.



range	total	min	max	avr	count	intern%	1%
(root)@TASKRCV1	22.630ms	-	22.630ms	22.630ms	-	<0.001%	
\\sample1\samplets1\PreTaskHook@TASKRCV1	143.885us	0.865us	1.730us	1.693us	85.	0.005%	
\\sample1\samplerv1\FuncTASKRCV1@TASKRCV1	22.615ms	-	22.615ms	22.615ms	1.(0/1)	0.024%	
\\sample1\osisr\OS_DisableAllInterrupts@TASKRCV1	40.210us	8.015us	8.140us	8.042us	5.	0.001%	
\\sample1\osisr\OS_EnableAllInterrupts@TASKRCV1	40.570us	8.015us	8.140us	8.114us	5.	0.001%	
\\sample1\osalm\OS_SetRelAlarm@TASKRCV1	3.902ms	37.360us	100.395us	41.074us	95.	0.108%	
\\sample1\osmc\OSWaitRC@TASKRCV1	995.890us	68.180us	130.985us	99.589us	10.	0.036%	
\\sample1\ossct\OS_StartScheduleTableRel@TASKRCV1	816.825us	163.165us	163.880us	163.365us	5.	0.005%	
\\sample1\osevt\OS_ClearEvent@TASKRCV1	1.159ms	12.700us	14.800us	12.882us	90.	0.042%	
\\sample1\osalm\OSInsertAlarm@TASKRCV1	572.445us	6.285us	7.650us	6.360us	90.	0.020%	
\\sample1\osevt\OS_WaitEvent@TASKRCV1	5.654ms	24.165us	65.620us	62.824us	90.(0/1)	0.057%	

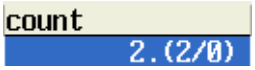
columns (cont.)	
total	total time within the function
min	shortest time between function entry and exit, time spent in interrupt service routines is excluded No **min** time is displayed if a function exit was never executed.
max	longest time between function entry and exit, time spent in interrupt service routines is excluded
avr	average time between function entry and exit, time spent in interrupt service routines is excluded



columns (cont.)	
count	number of 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>*).



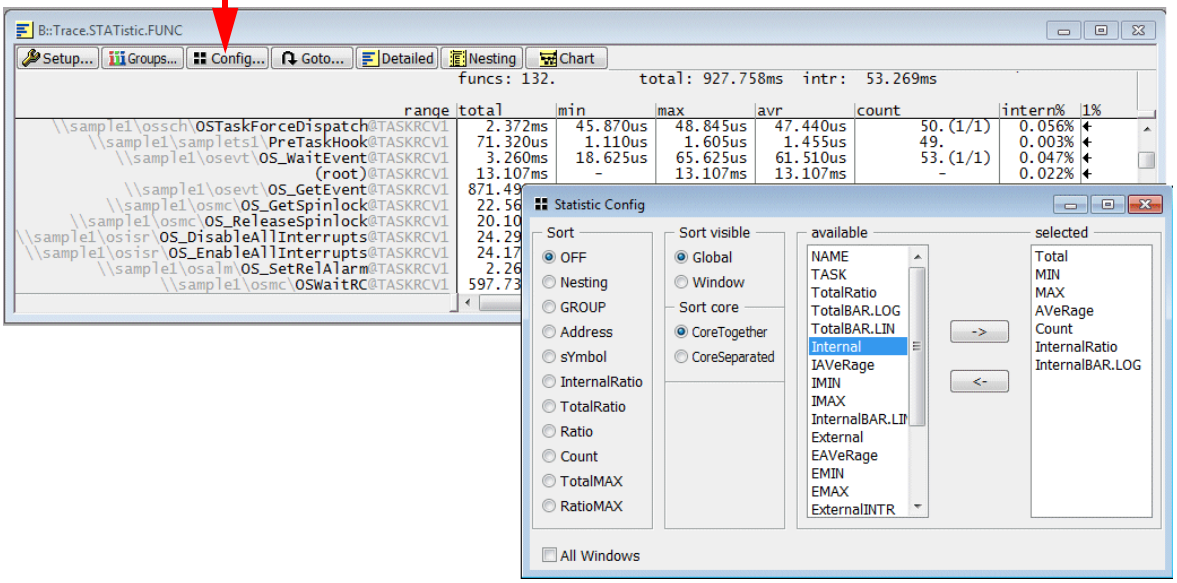
Interpretation examples:

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.)	
intern% **(InternalRatio, InternalBAR.LOG)**	ratio of time within the function without subfunctions and interrupts

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

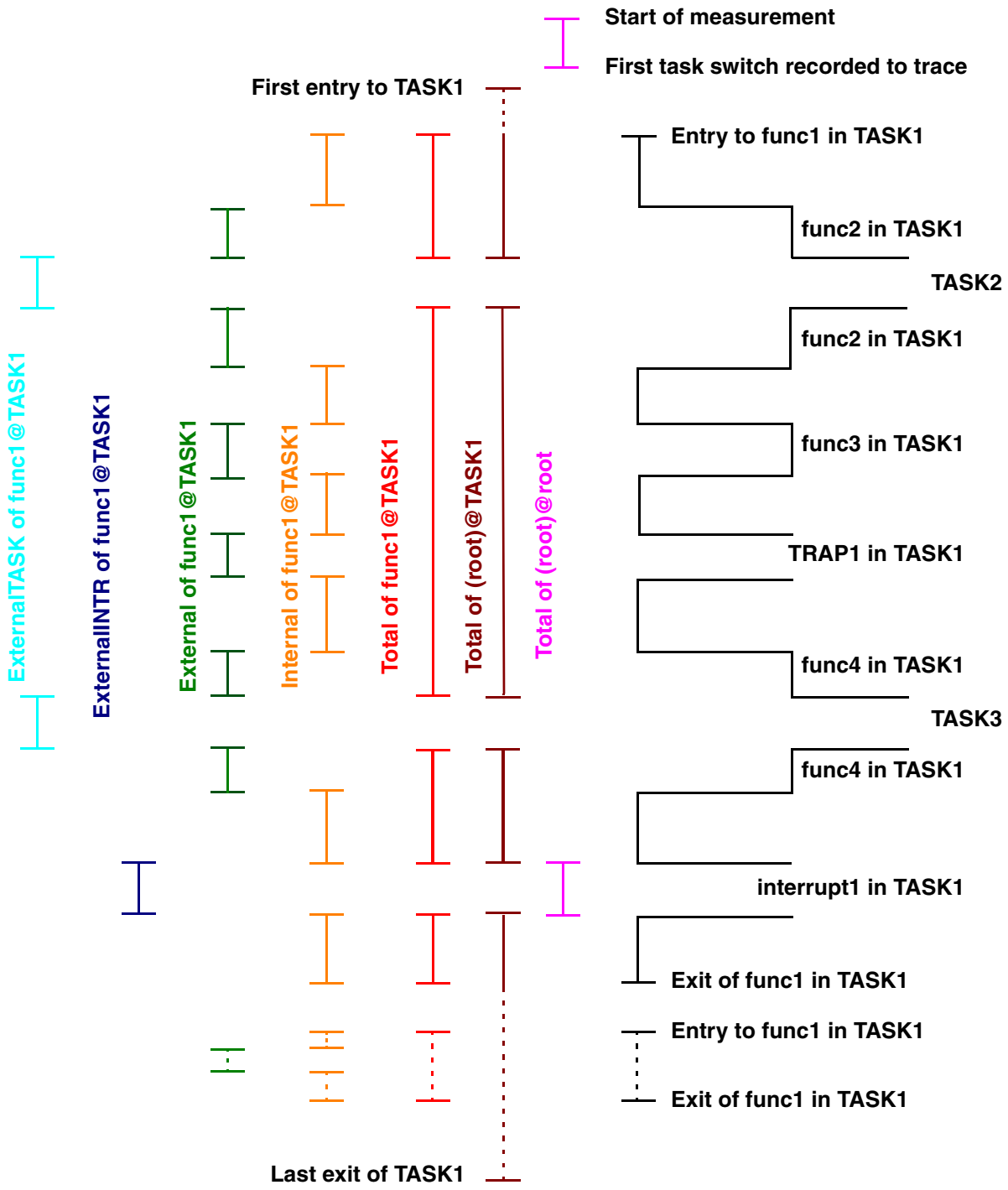


columns (cont.) - times only in function	
Internal	total time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IAVeRage	average time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IMIN	shortest 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	
ExternalINTR	total time the function was interrupted
ExternalINTRMAX	max. time one function pass was interrupted
INTRCount	number of interrupts that occurred during the function run-time

columns - task/thread related information	
TASKCount	number of tasks that interrupt the function
ExternalTASK	total time in other tasks
ExternalTASKMAX	max. time 1 function pass was interrupted by a task


Start of measurement
First task switch recorded to trace
First entry to TASK1
Entry to func1 in TASK1
func2 in TASK1
TASK2
func2 in TASK1
func3 in TASK1
TRAP1 in TASK1
func4 in TASK1
TASK3
func4 in TASK1
interrupt1 in TASK1
Exit of func1 in TASK1
Entry to func1 in TASK1
Exit of func1 in TASK1
Last exit of TASK1
ExternalTASK of func1@TASK1
ExternalINTR of func1@TASK1
External of func1@TASK1
Internal of func1@TASK1
Total of func1@TASK1
Total of (root)@TASK1
Total of (root)@root

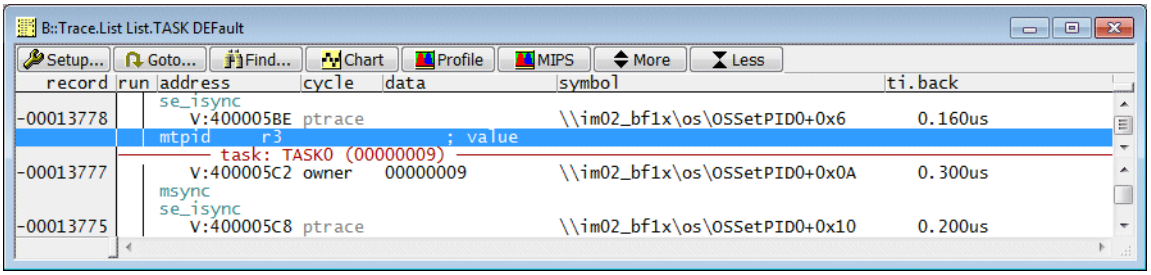
The standard NEXUS settings do often not allow to locate exactly the instructions that are already executed by a newly activated task. This is especially true is Branch History Messaging is used. This might disturb the task-aware function run-time measurement.

An instruction-accurate assignment of the task switches may improve the results.

IEEE-ISTO 5001-2008 and Subsequent Standards

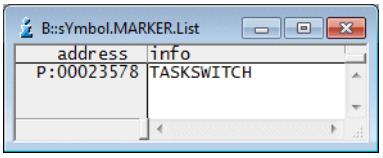
The Ownership Trace Messages (task switches) can be exactly assigned to an instruction, if the following setting is done.

```
NEXUS.PTCM PID_MSR ON                   ; enable Program Trace Correlation
                                        ; Messages for PID0/NPIDR accesses

NEXUS.POTD ON                           ; disable Periodic Ownership Trace
                                        ; Messages
```



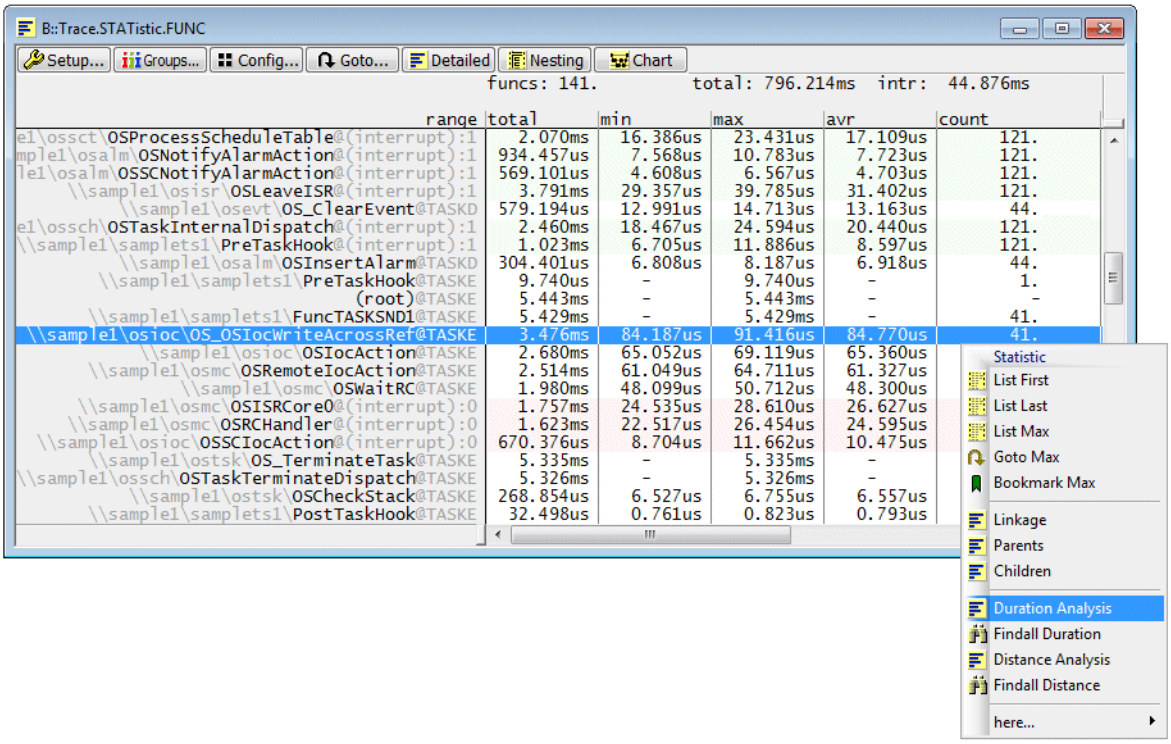
Alternative

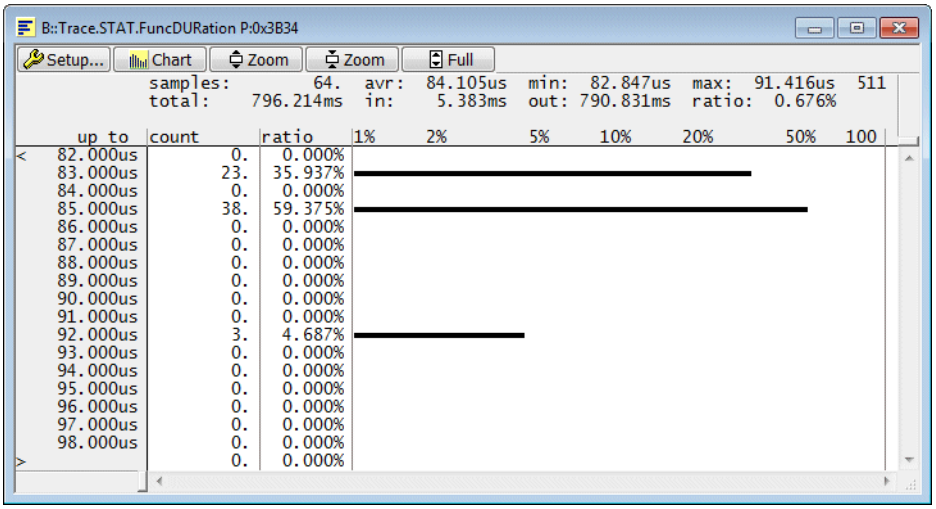
```
; mark instruction that performs the task switch for the task-aware
; function run-time analysis
sYmbol.MARKER.Create TASKSWITCH osDispatcher+0x100
```



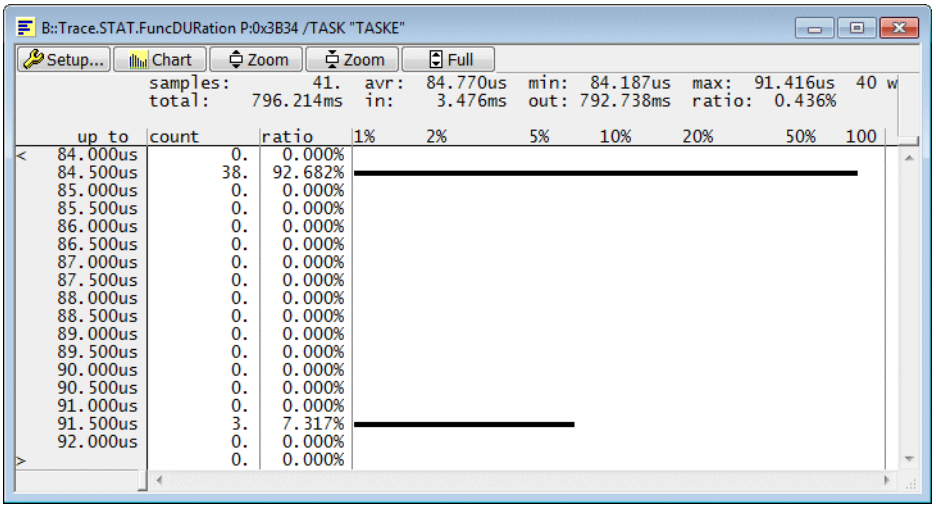
Detailed analysis of a single function, time between function entry and exit, time spent in interrupt service routines and other tasks is excluded.

Trace.STATistic.FuncDURation *<function>* [**/TASK** *"<task_name>"*] [**/FilterCORE** *"<core_number>"*]

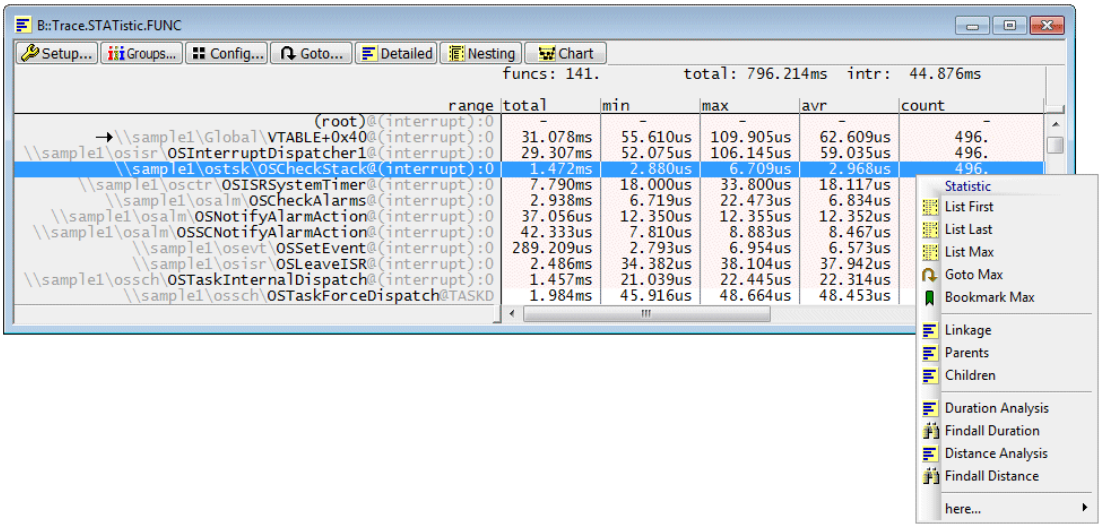


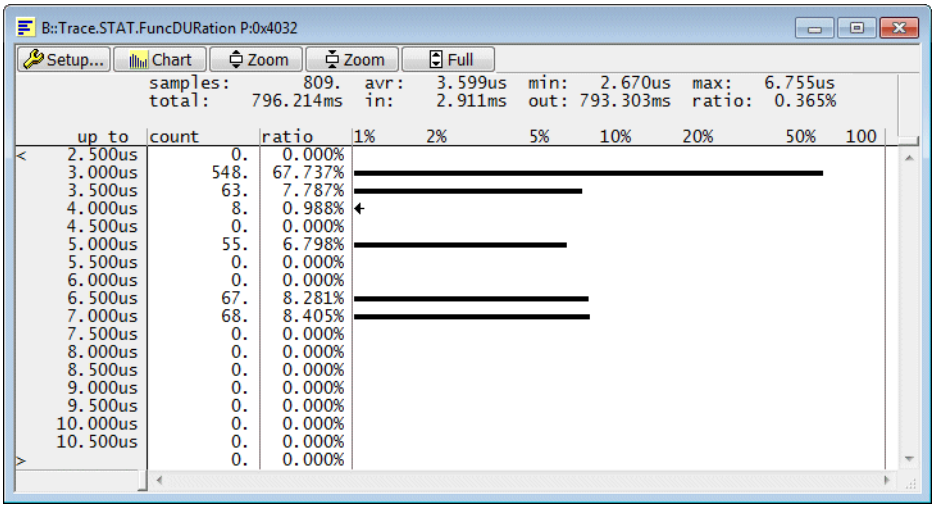


Please be aware, that details are shown for all function runs. If you are interested in a task-specific analysis you have to use the **/TASK** "*<task_name>*" option.

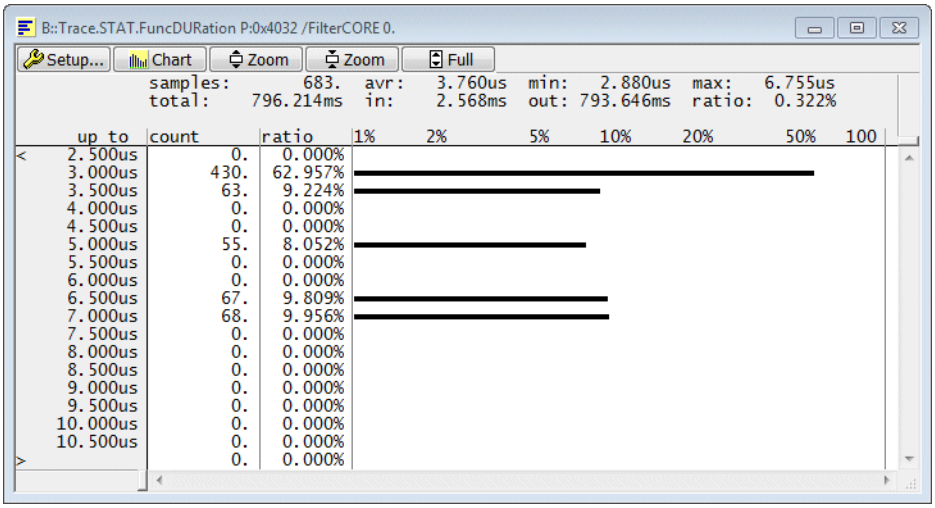


The @(interrrupt) and the @(unknown) task are split up per core.



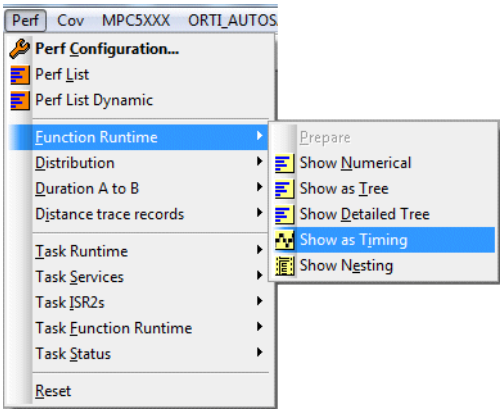


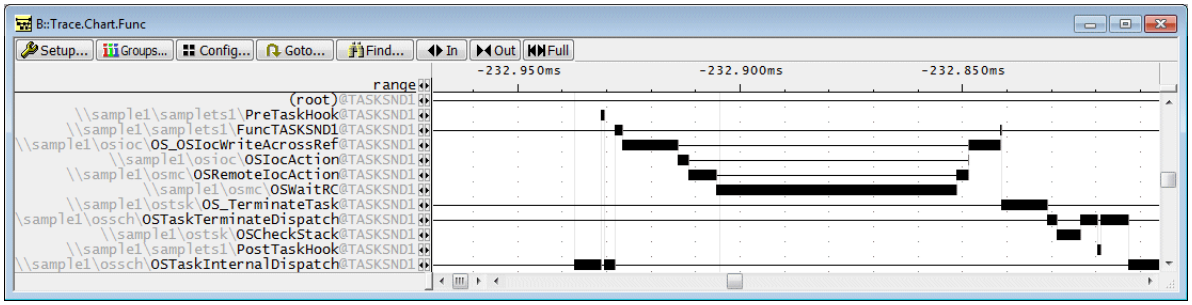
Please be aware, that details are shown for all function runs. If you are interested in a core-specific analysis you have to use the **/FilterCORE** *<core_number>* option.



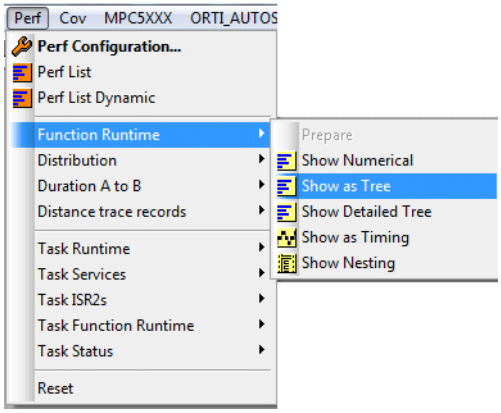
Trace.Chart.Func

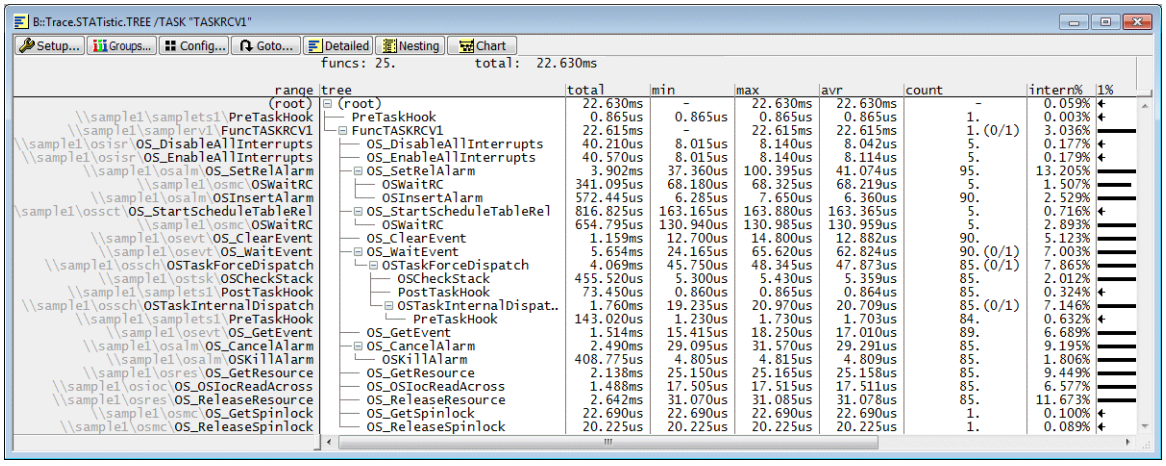
Nested function run-time analysis
- graphical display





Trace.STATistic.TREE Nested function run-time analysis
- tree display



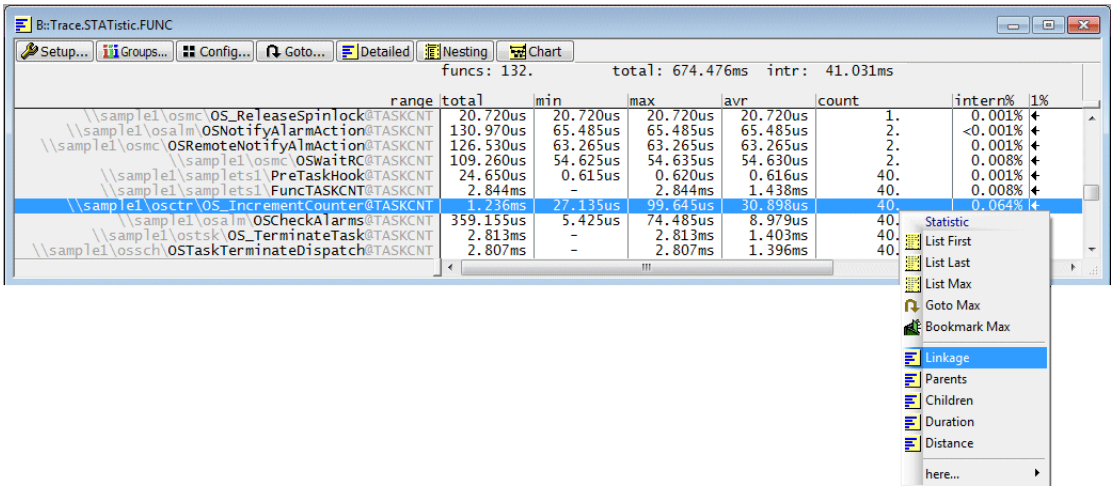


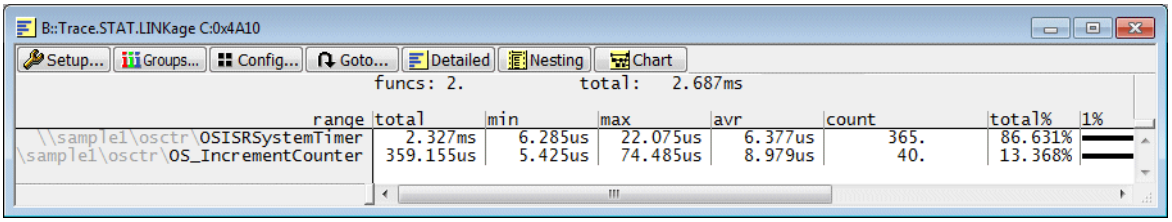
It is also possible to get a task-specific tree.

```
Trace.STATistic.TREE /TASK "TASKRCV1"
```

Trace.STATistic.LINKage *<address>*

Nested function run-time analysis
- linkage analysis





Third-party Timing Tools

TRACE32 also provides an interface to third-party timing tools. For details refer to **"Trace Export for Third-Party Timing Tools"** (app_timing_tools.pdf).

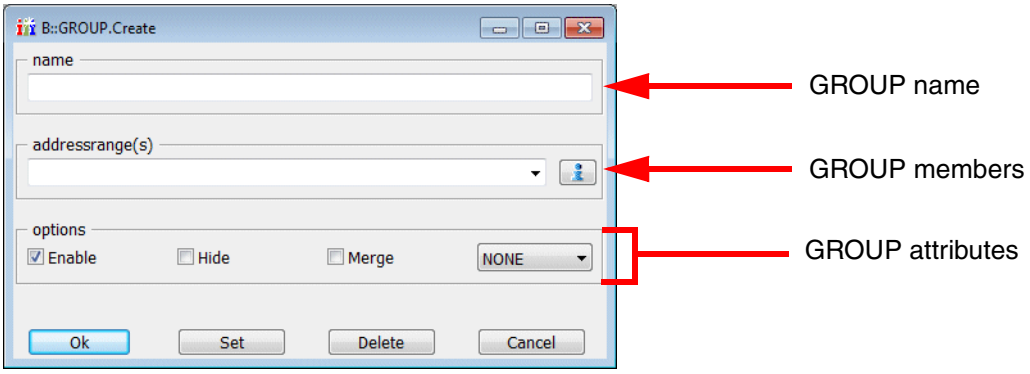
Structure the Trace Evaluation

The command group GROUP allows to structure the software for the trace evaluation. This is especially useful if the software consists of a huge number of functions/modules.

GROUP Creation

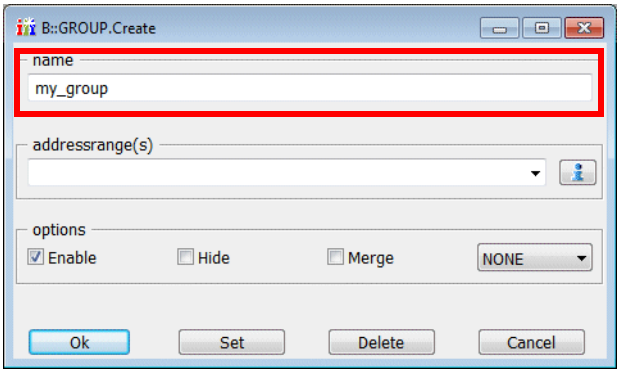
```
GROUP.Create
```

If the command **GROUP.Create** is entered without parameters, the **Group.Create** dialog is opened.



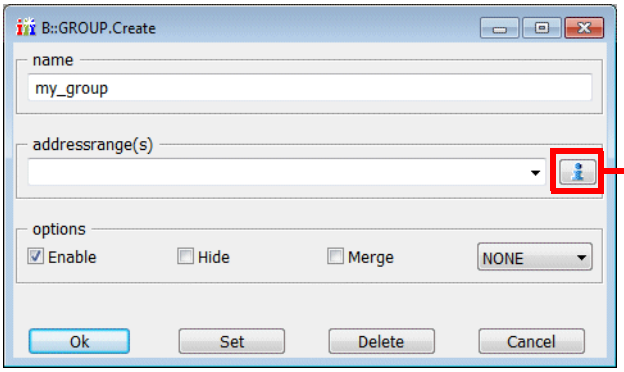
The basic setup for a GROUP includes the following steps:

1. **Specify the GROUP name.**



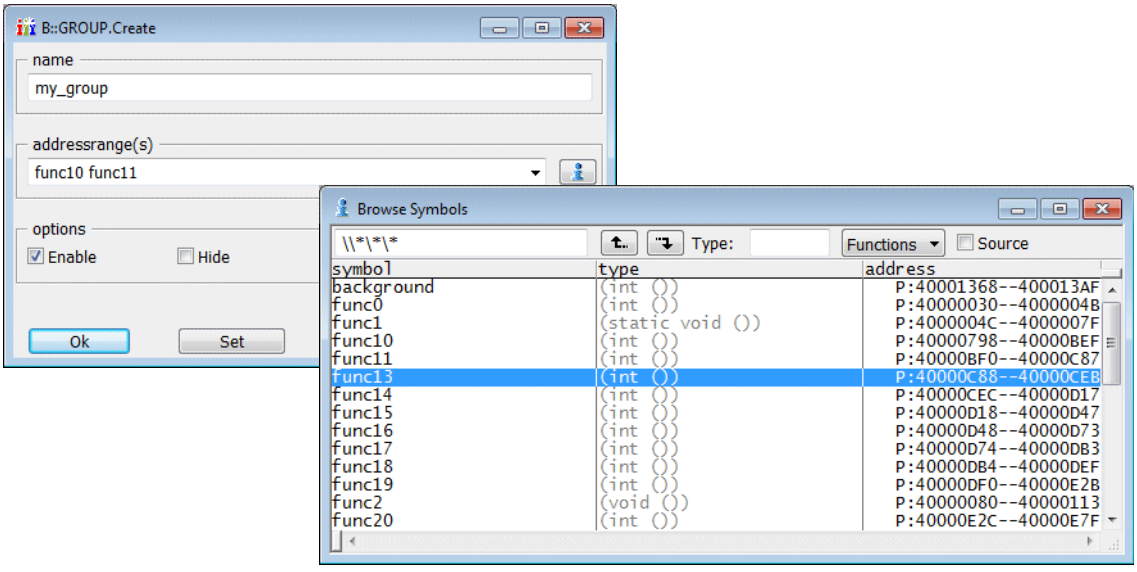
2. Specify the GROUP members.

GROUPS are address ranges, so you can use functions, modules, or programs to specify the group members.



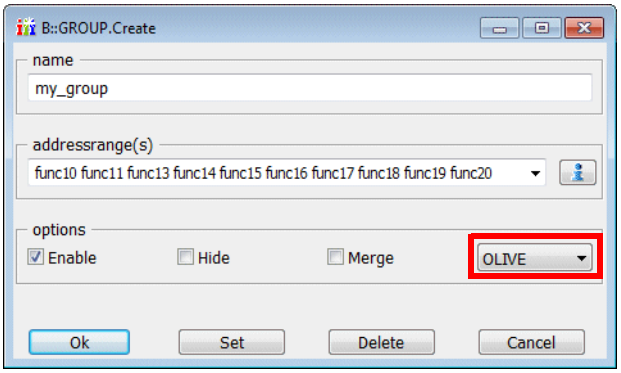
Open the symbol data base to select the group members

A new group member is selected by a double-click.

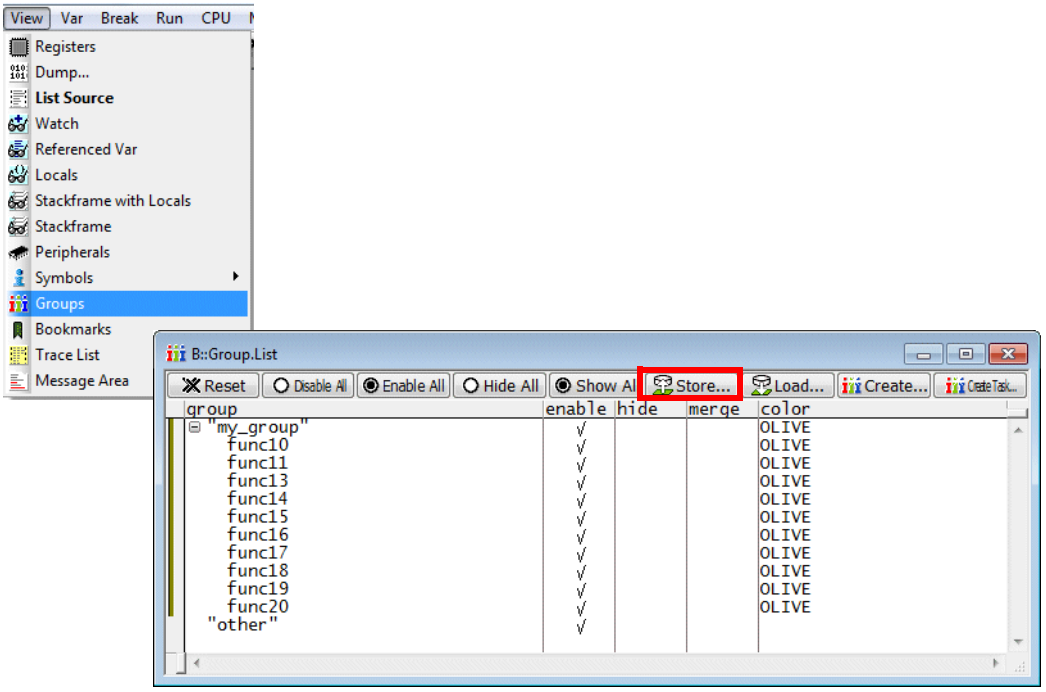


3. **Specify the GROUP color and close the dialog with Ok.**

The GROUP color is used to mark the GROUP members in the trace analysis windows.



4. **Display the GROUP information.**



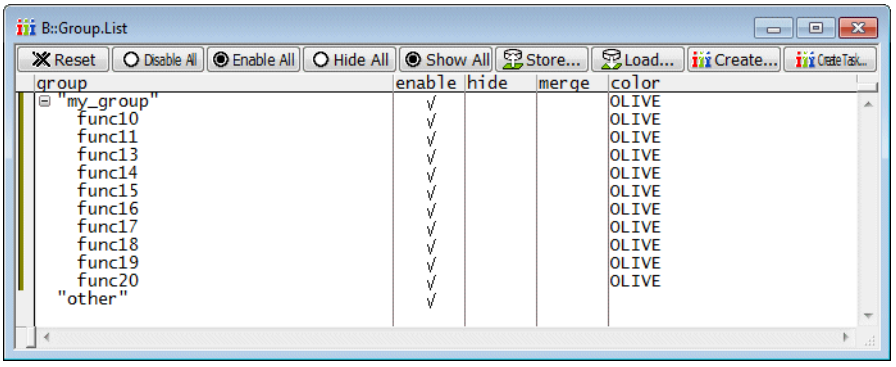
5. **Push the Store… button, if you want to generate a scipt that allows you to re-set the specified groups at any time.**

```
; script group_settings.cmm
B::

 GROUP.RESET
 GROUP.CREATE "my_group" \\diabc\diabc\func10 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func11 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func13 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func14 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func15 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func16 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func17 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func18 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func19 /OLIVE
 GROUP.CREATE "my_group" \\diabc\diabc\func20 /OLIVE

 ENDDO
```

The GROUP status determines the appearance of a GROUP in the trace display and analysis windows. The following three statuses are available:

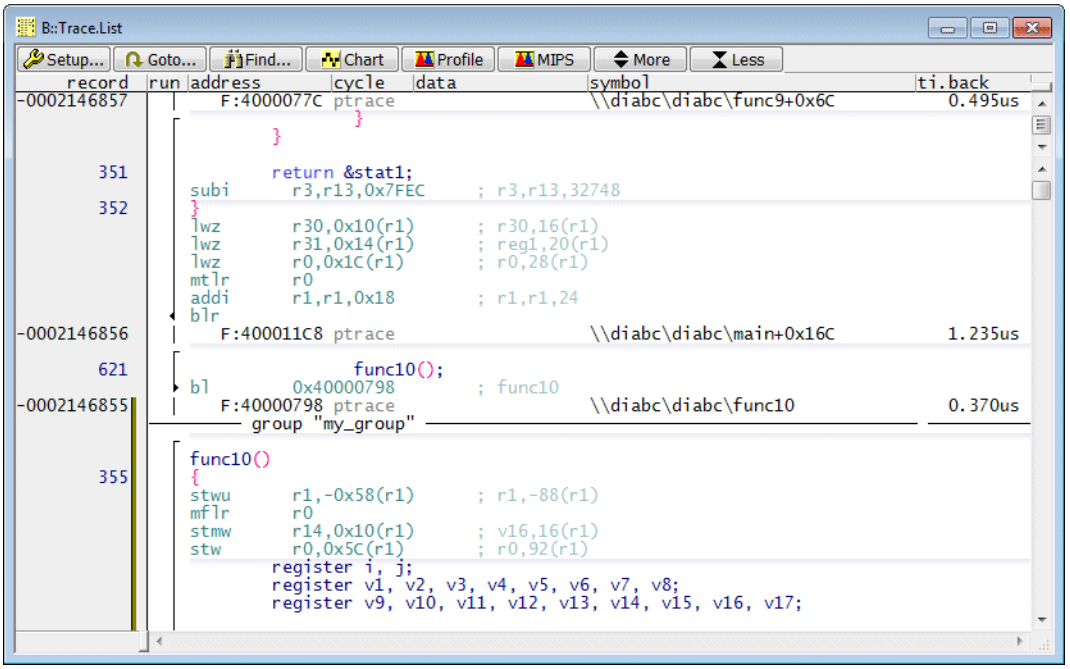
- ENable
- ENable + Merge
- ENable + HIDE

GROUP Status ENable



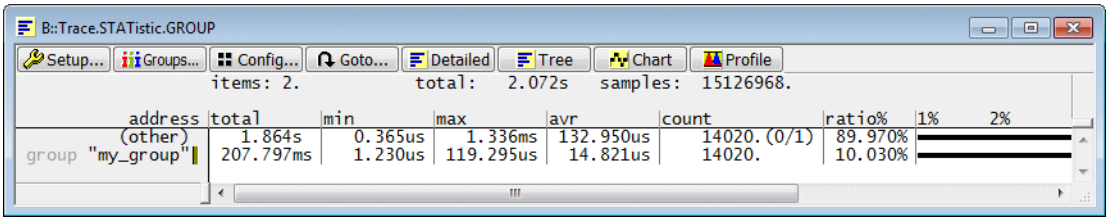
TRACE32 provide the following features if a GROUP has the status ENable:

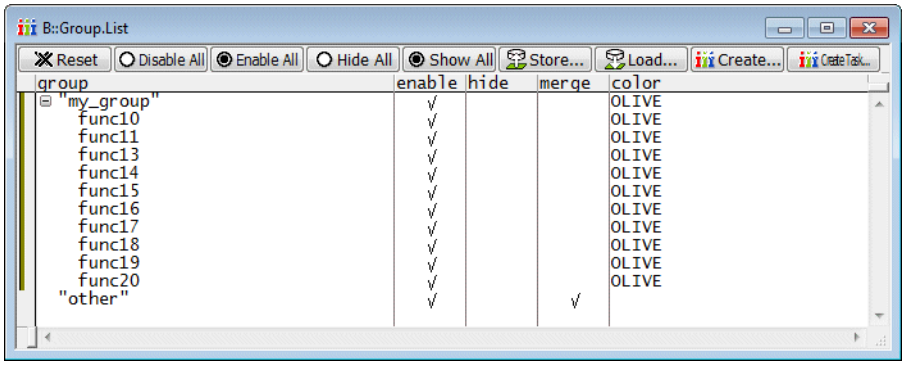
1. **GROUP members are marked in the Trace Listing by their group color.**



2. Special statistic commands are provide for the GROUPS.

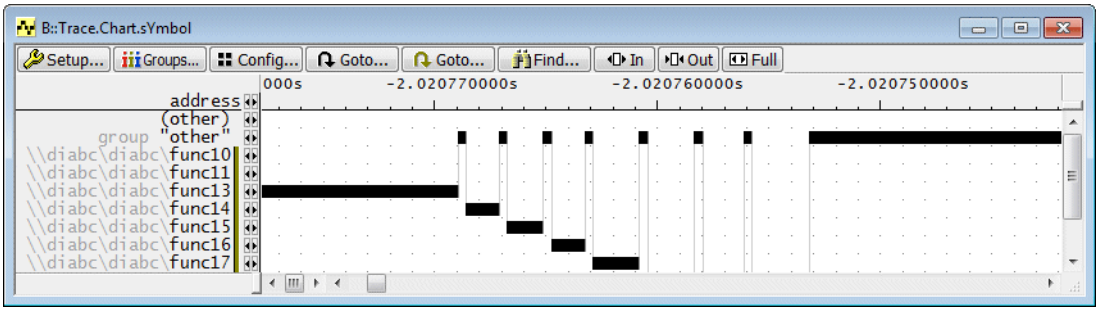
Trace.STATistic.GROUP	Group-based run-time analysis.
Trace.Chart.GROUP	Group-based time chart.

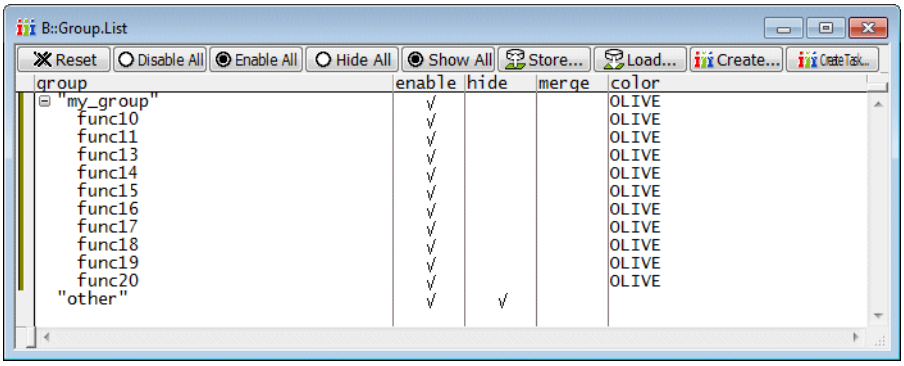




TRACE32 provide the following feature if a GROUP has the status ENable and Merge:

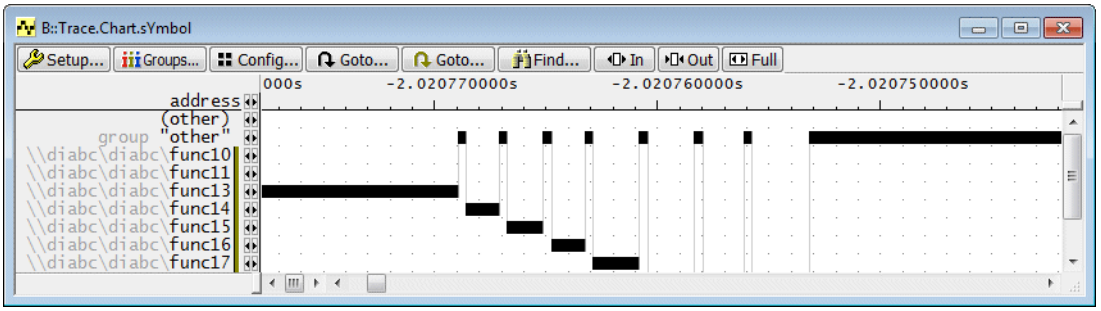
The GROUP represents its members in all trace analysis window. No details about the GROUP members are displayed.



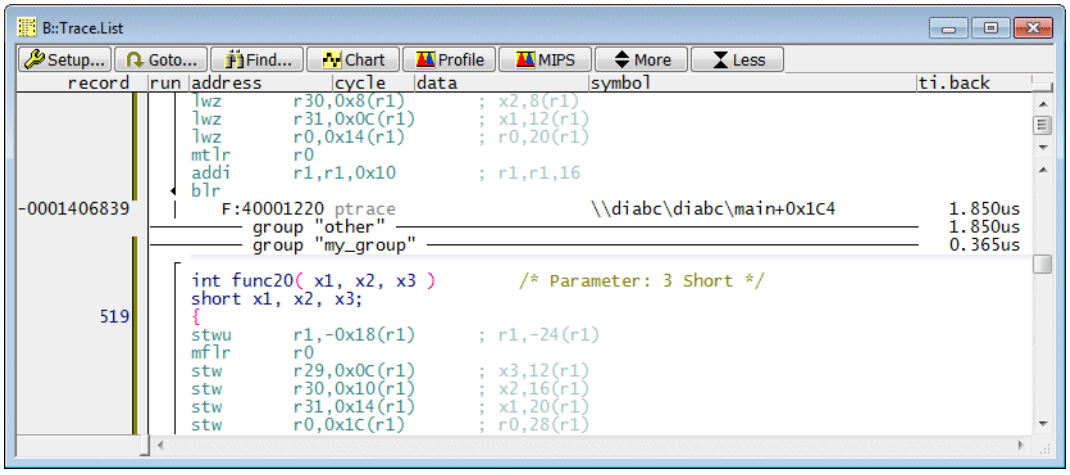


TRACE32 provide the following feature if a GROUP has the status ENable and HIDE:

1. **The GROUP represents its members in all trace analysis window. No details about the GROUP members are displayed.**



2. **The trace information recorded for the GROUP members is hidden in the Trace Listing.**

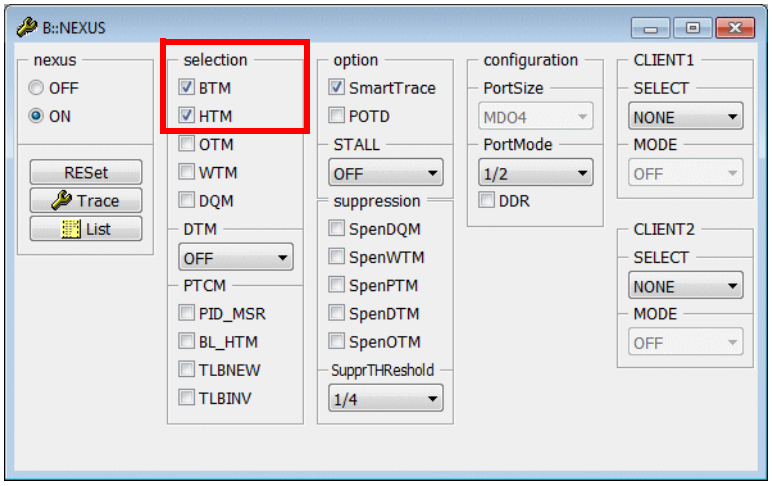


Trace-based Code Coverage

The manual **"Application Note for Trace-Based Code Coverage"** (app_code_coverage.pdf) gives a detailed introduction to the trace-based code coverage.

Since the core information is discarded for trace-based code coverage, there is no difference between single-core and SMP TRACE32 instances with regard to this feature.

It is recommended to enable Branch History Messaging for Code Coverage. This advises the Nexus module to generate the trace messages in a compact way.




```
NEXUS.BTM ON

NEXUS.HTM ON
```

Incremental Code Coverage has to be used in the following situations:

- POWERTRACE/ETHERNET as universal debug and trace hardware
- POWER TRACE PX as universal trace hardware
- On-chip trace memory
- High-bandwidth trace interfaces