ARM-ETM Training

Version 25-Aug-2020

History

24-Aug-2020 Chapter "Code Coverage" was updated. Cross references to **"Application Note for Trace-Based Code Coverage"** (app_code_coverage.pdf) was added. Chapter "Code Coverage" now only contains some hints for the setups for ARM-ETM.

ETM Setup

ETM Versions

The parallel ETM is available in the following versions:

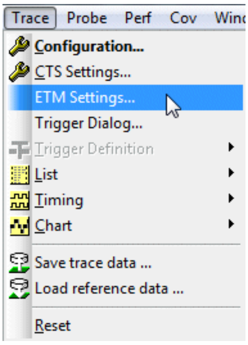
- ETMv1.x for ARM7 and ARM9
- ETMv3.x for ARM11
- ETMv3.x CoreSight Single for ARM9 and ARM11
- ETMv3.x CoreSight for ARM9, ARM11, Cortex-M3/M4/M23, Cortex-R4/R5, Cortex-A5/A7
- PTM for Cortex-A9/A15/A17
- ETMv4 for Cortex-R7/R8/R52, Cortex-M7/M33 and Cortex-A3x/A5x/A7x (Cortex-A32/A35/A53/A55/A57/A72/A73/A75)

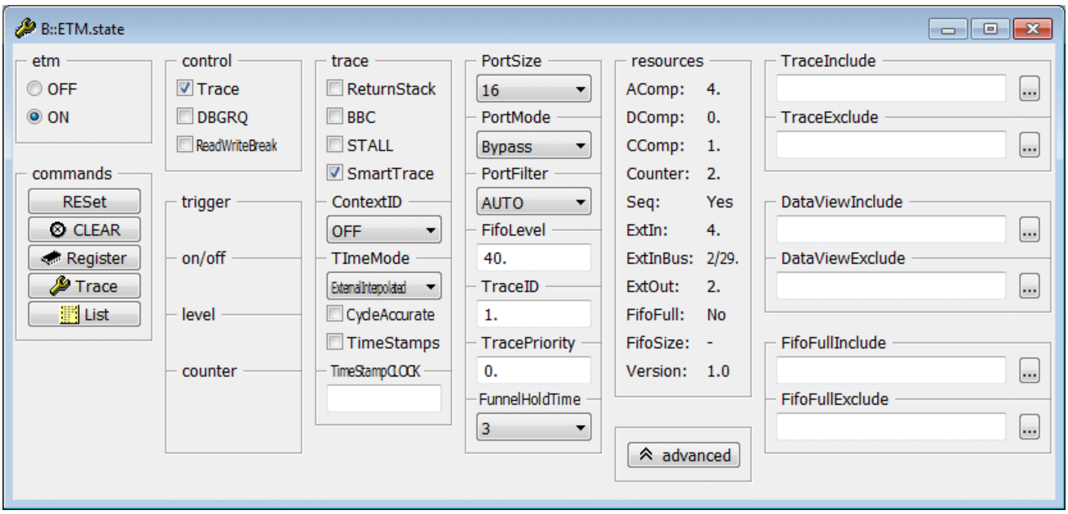
The ETM trace information can be stored internally in an onchip memory (ETB, ETF, ETR) or exported via a trace port interface (TPIU) to an external recording device (PowerTrace, CombiProbe, µTrace).
Using internal memory for trace recording is also called “onchip trace”.
Using an external recording device is also called “offchip trace”

Chips supporting “offchip trace” usually export data via a parallel trace port: Via up to 40 pins on the chip the trace data is transferred to the PowerTrace.
Some modern Cortex-A and Cortex-R chips support also the export of the data via a High Speed Serial Trace Port (HSSTP)

ETM.state Window

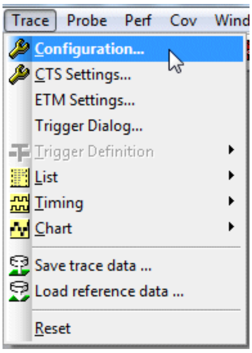
The **ETM.state** window allows to configure the ETM/PTM and the TPIU.

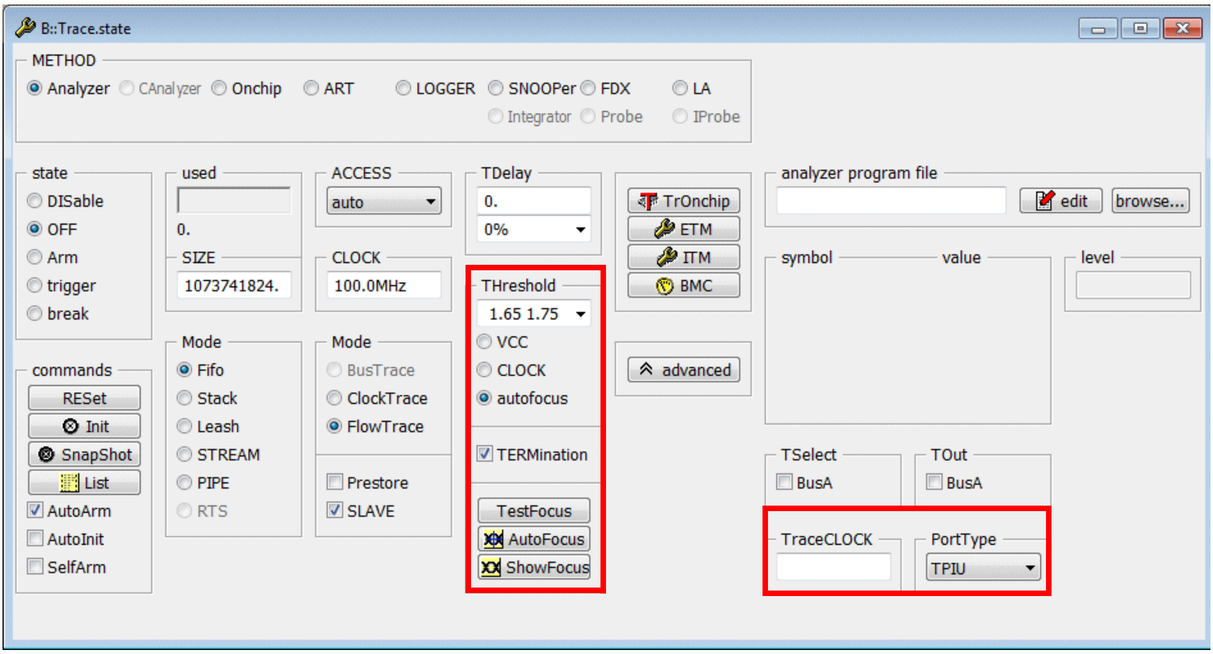




The JTAG interface is use to configure the ETM/PTM and the TPIU.

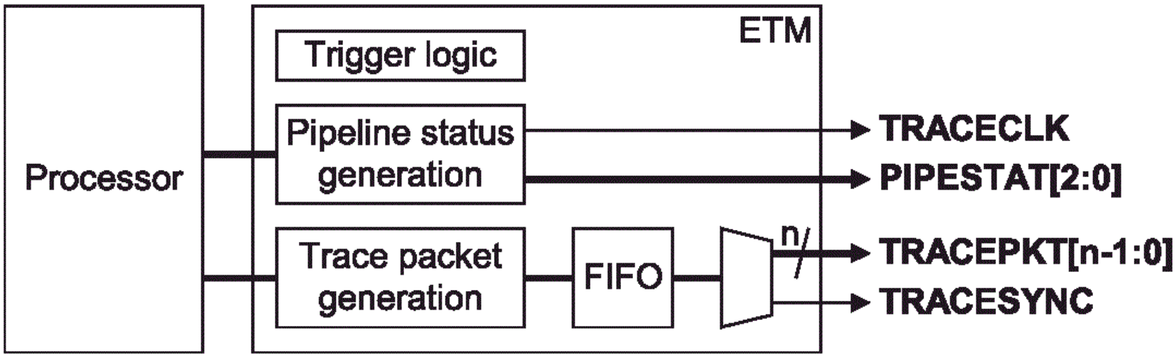
The **Trace.state** window allows to configure the TRACE32 Preprocessor AutoFocus II.





This chapter is only relevant if you have an ARM7 or ARM9 chip with ETMv1.

Interface and Trace Protocol



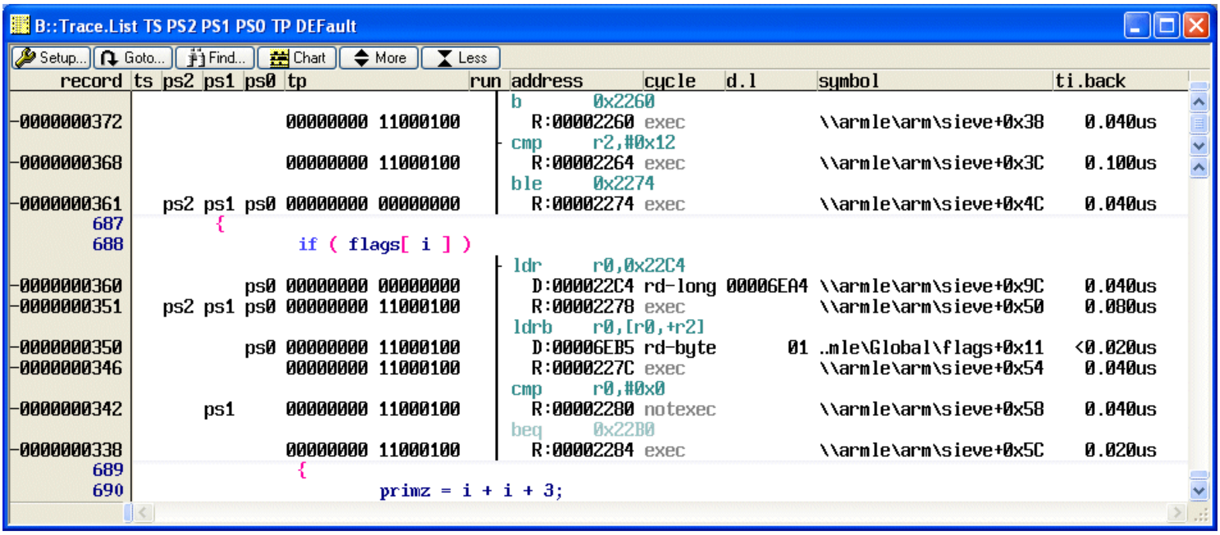
PIPESTAT	Pipeline status pins provide a cycle-by-cycle indication of what is happening in the execute stage of the processor pipeline (instruction executed, branch executed etc.).
TRACEPKT	Trace packets (broadcast address and data information)
TRACECLK	ETM clock

Trace packets contain the following information:

- Address information when the processor branches to a location that cannot be directly inferred from the source code. When addresses are broadcast, high-order bits that have not changed are not broadcast.
- Data accesses that are indicated to be of interest (only data, only addresses, data and addresses).

	The maximum sampling time for the program and data flow trace depends mainly • on the size of the trace buffer • the CPU clock because the pipeline status information has to be sampled every clock cycle.

```
Trace.List TS PS2 PS1 PS0 TP DEFault
```



ETM signals	
TS	Trace synchronization signal
PS[0..2]	Pipeline status
TP **TPH** **TPL**	Trace packets (depending on PortSize) Trace packets high byte (PortSize=16 only) Trace packets low byte (PortSize=16 only)

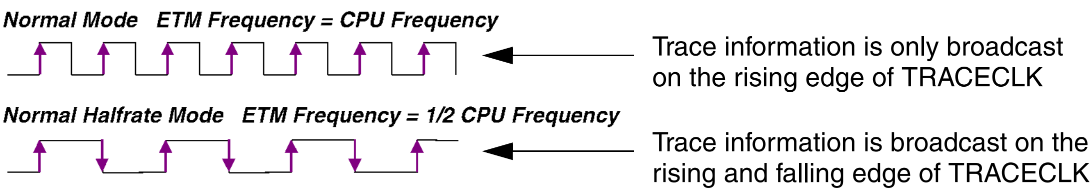
Port Size and Port Mode for ETMv1.x for ARM7/ARM9

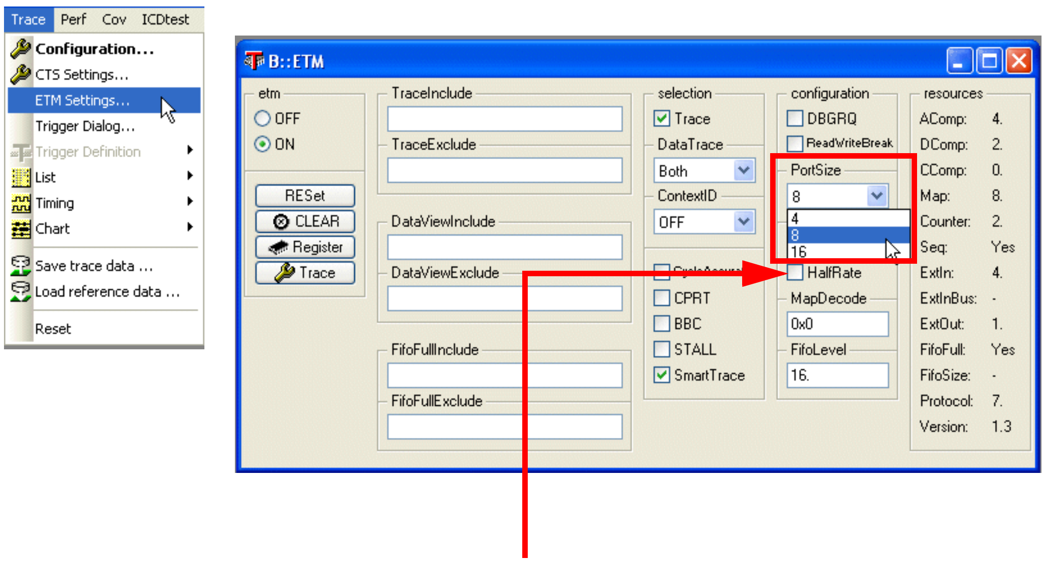
1. **Define the *ETM port size*.**

   TRACEPKT can be either 4, 8 or 16 bit.

2. **Define if your ETM is working in *Halfrate Mode* or not.**

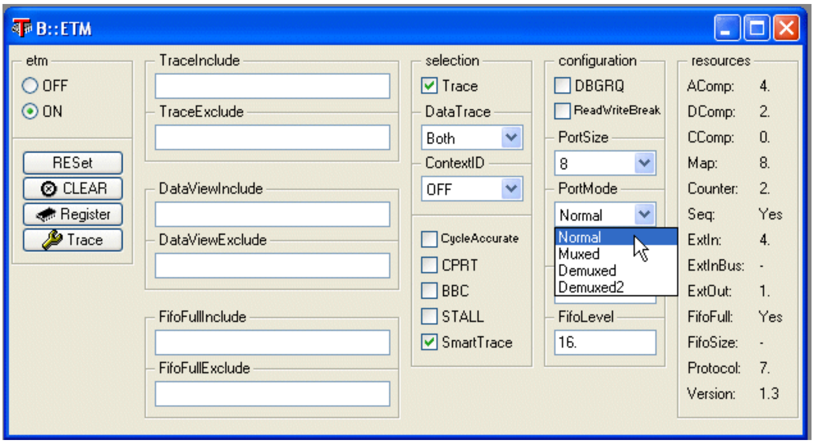
   In Halfrate Mode trace information is broadcast on the rising and falling edge of TRACECLK.





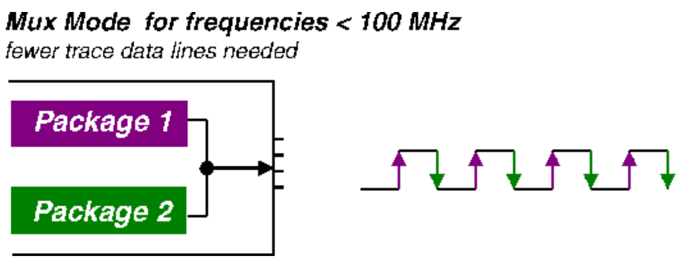
Switch to ON if ETM is working in ***Halfrate*** mode

3. **Define the mode of the ETM port.**



Normal Mode: one trace line is output via one trace port pin. HalfRate mode is supported in normal mode.

Multiplexed Mode: the multiplexed mode can be used to save trace lines; 2 trace lines are multiplexed to a single trace port pin.



Signals sampled on the rising edge of TRACECLK	Signals sampled on the falling edge of TRACECLK
PIPESTAT[0]	TRACESYNC in ETMv1 PIPESTAT[3] in ETMv2
PIPESTAT[1]	TRACEPKT[1]
PIPESTAT[2]	TRACEPKT[2]
TRACEPKT[0]	TRACEPKT[3]

Example for a 4-pin trace connector

Demultiplexed Mode: the demultiplexed mode is used to reduce the switching rate for the trace port; each trace line is split across 2 trace port pins. HalfRate mode is supported in demultiplexed mode.

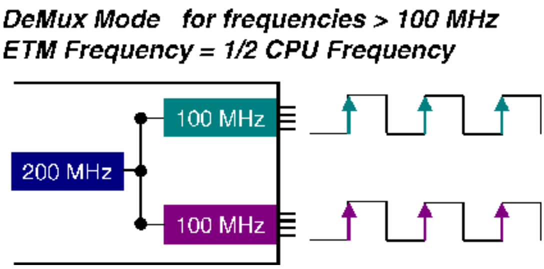
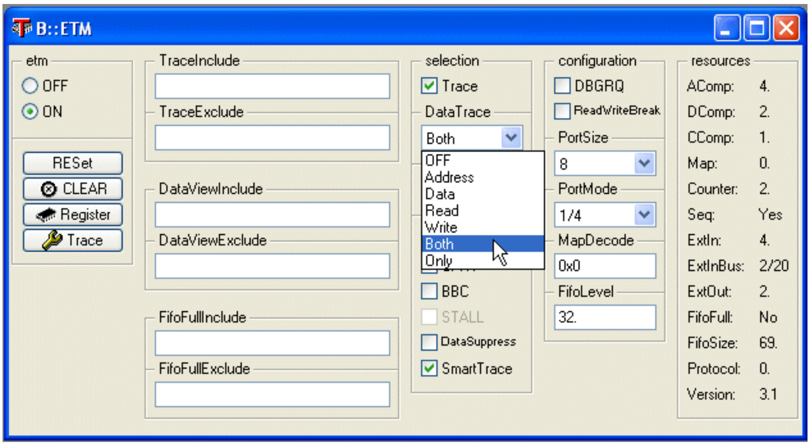


Table 7-6 Demultiplexed four-bit connector pinout

Pin	Signal name	Pin	Signal name
38	**PIPESTAT_A[0]**	37	**PIPESTAT_B[0]**
36	**PIPESTAT_A[1]**	35	**PIPESTAT_B[1]**
34	**PIPESTAT_A[2]**	33	**PIPESTAT_B[2]**
32	**TRACESYNC_A** in ETMv1 **PIPESTAT_A[3]** in ETMv2	31	**TRACESYNC_B** in ETMv1 **PIPESTAT_B[3]** in ETMv2
30	**TRACEPKT_A[0]**	29	**TRACEPKT_B[0]**
28	**TRACEPKT_A[1]**	27	**TRACEPKT_B[1]**
26	**TRACEPKT_A[2]**	25	**TRACEPKT_B[2]**
24	**TRACEPKT_A[3]**	23	**TRACEPKT_B[3]**
22	No connect	21	**nTRST**
20	No connect	19	**TDI**
18	No connect	17	**TMS**
16	No connect	15	**TCK**
14	**VSupply**	13	**RTCK**
12	**VTRef**	11	**TDO**
10	**EXTTRIG**	9	**nSRST**
8	**DBGACK**	7	**DBGRQ**
6	**TRACECLK**	5	**GND**
4	No connect	3	No connect
2	No connect	1	No connect

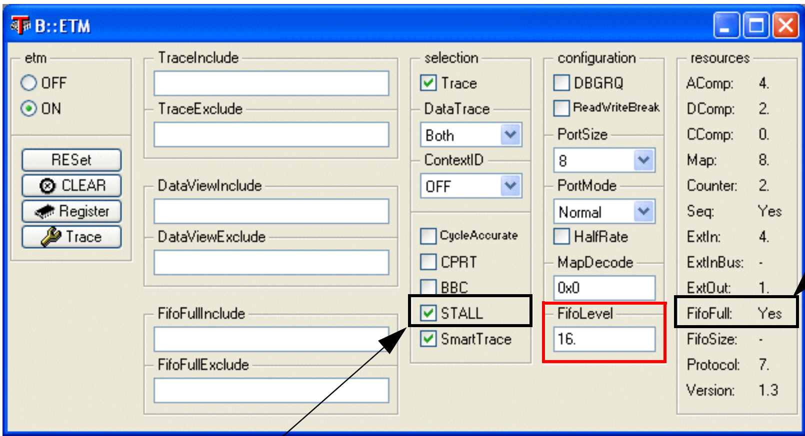
Example for a 4-bit trace connector

Demuxed	4-bit demultiplexed trace port (one mictor connector to target)
Demuxed2	8- and 16-bit demultiplexed trace port (two mictor connectors to target)



DataTrace (ETMv1)	
OFF	No information about data accesses is broadcast.
Address	Only the address information for data accesses is broadcast.
Data	Only the data information for data accesses is broadcast.
Read	The address and data information is broadcast for read accesses.
Write	The address and data information is broadcast for write accesses.
Both	The address and data information is broadcast for all data accesses.

If a FIFOFULL logic is implemented on your ETM, it is possible to stall the processor when a FIFO overflow is likely to happen.



FIFOFULL logic available

Stall the processor if a FIFO overflow is likely to happen

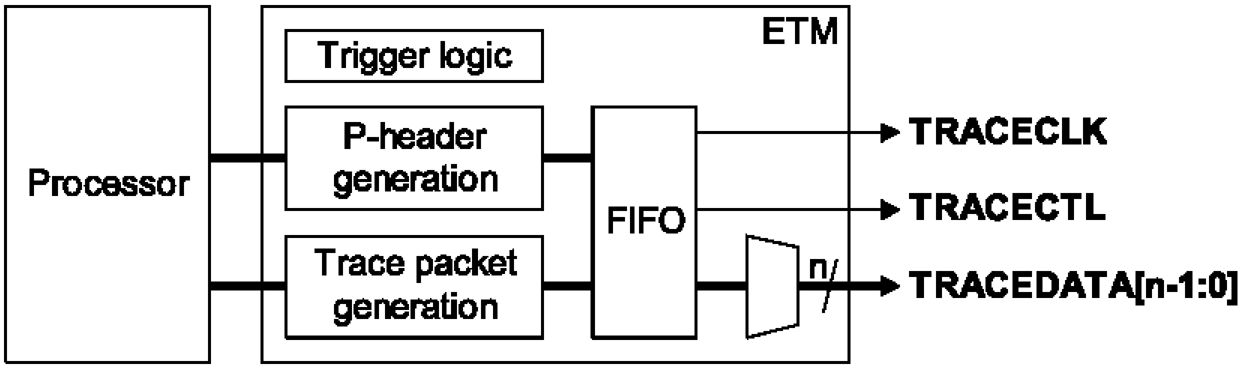
The ETM does not provide any information to help you to find out how big is the performance loss caused by stalling the processor.

ARM7TMI, ARM720T, ARM920T and ARM922T do not support the stalling of the core when the ETM target FiFo is (almost) full. (You can set the option, but it won’t have an effect.)

ETM.STALL ON is supported by ARM926EJ-S, ARM946E-S and ARM966E-S.

This chapter is only relevant if you have a chip with an ARM ETMv3.
This are usually chips with a Cortex-M3/M4/M23, Cortex-R4/R5, Cortex-A5/A7/A8, ARM9 or ARM11 core.

Interface and Protocol

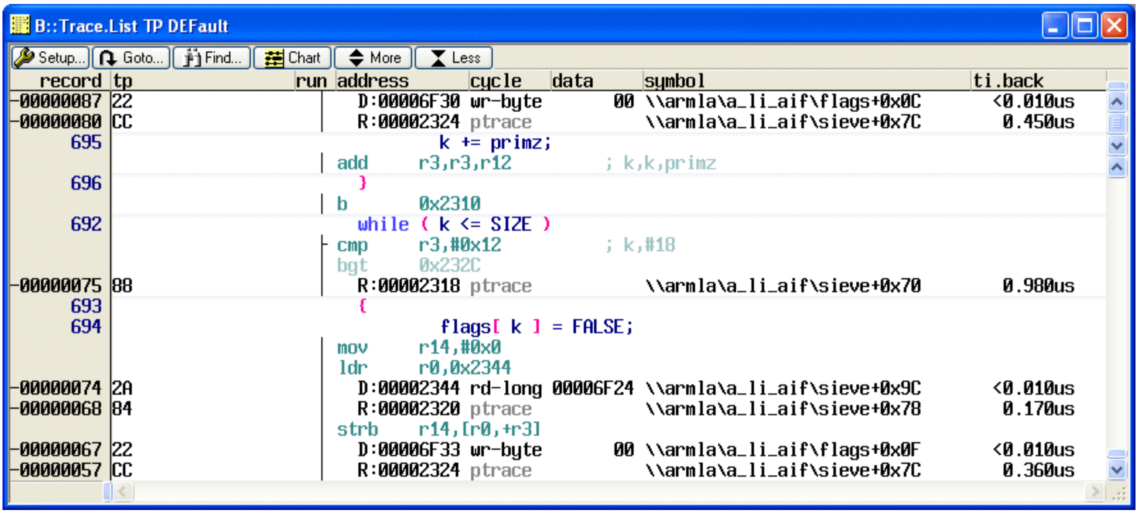


TRACECLK	Trace clock
TRACECTL	Trace control indicates if the trace information is valid.
TRACEDATA[n-1:0]	Trace packets (broadcast pipeline status information, address and data information)

Trace packets contain the following information:

- Information about the execution of instructions.
- Address information when the processor branches to a location that cannot be directly inferred from the source code. When addresses are broadcast, high-order bits that have not changed are not broadcast.
- Data accesses that are indicated to be of interest (only data, only addresses, data and addresses)

Trace.List TP DEFault



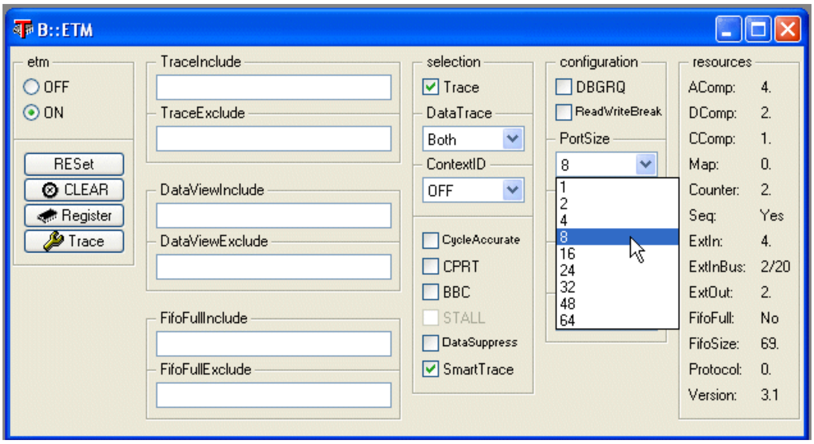
.

	Due to the fact that the information about the execution of instructions is broadcasted by trace packets, the maximum sampling time depends on the number of broadcasted packages.

Port Size and Port Mode for ETMv3.x for ARM11

1. Define the ETM port size

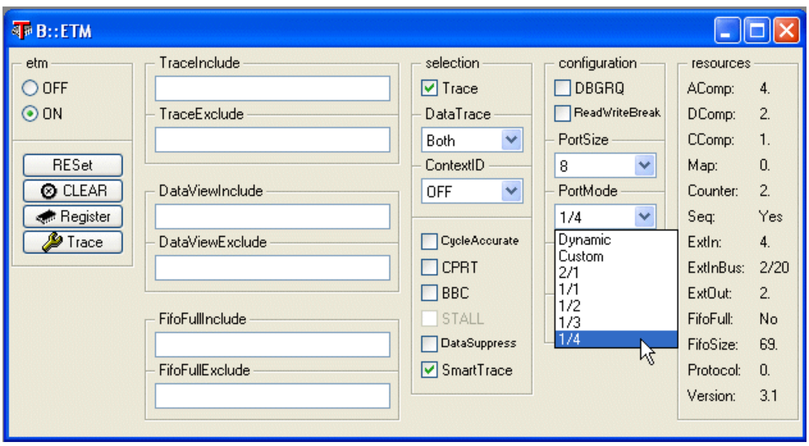
TRACEDATA can use 1..32 pins (48 and 64 pins port size is not supported yet).

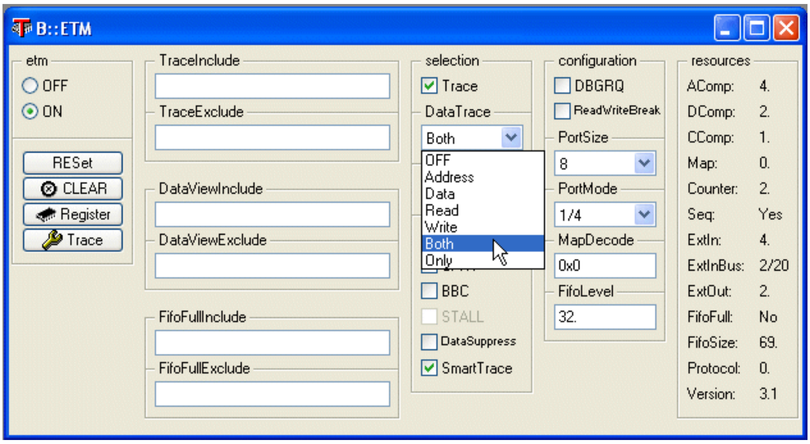


2. Define the mode for the ETM port

The ETMv3.x works always in half-rate mode.

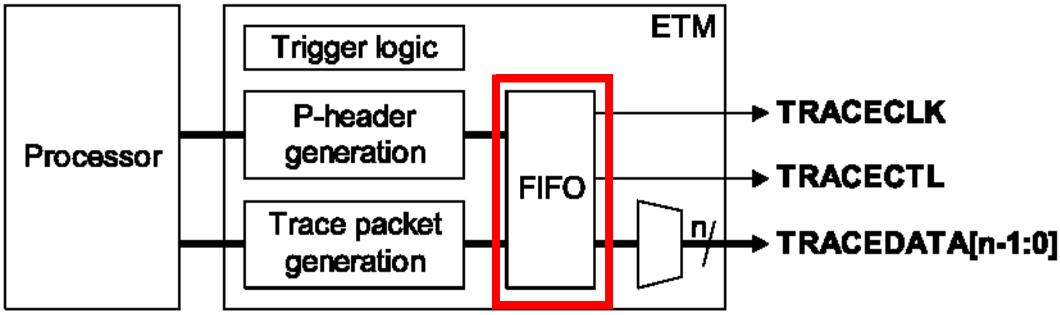
The port mode defines the relation *<etm_clock>/<cpu_clock>.*



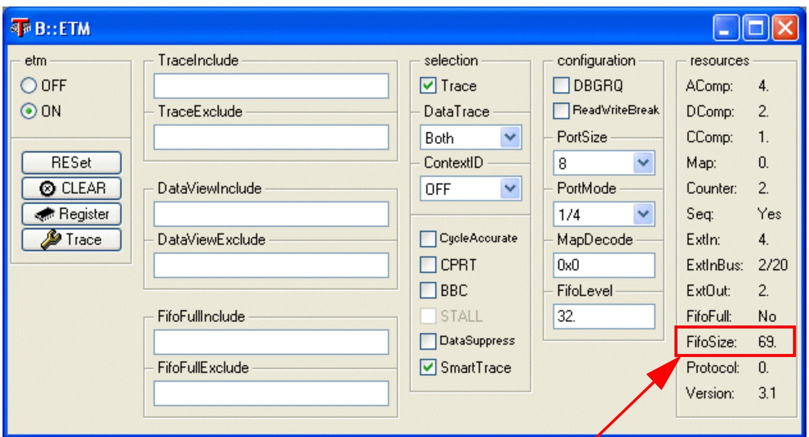


DataTrace (ETMv3)	
OFF	No data accesses are traced. Only program flow is traced.
ON	Both address and data information of for all data accesses (Read and write accesses) are traced (and the program flow).
Read	Both address and data for read accesses are traced (and the prog. flow).
Write	Both address and data for write accesses are traced (and the prog. flow).
Address	The addresses of all read/write accesses are traced (and the program flow), but not the data value, which has been read or written.
ReadAddress	The addresses of all data read accesses are traced (and the prog. flow).
WriteAddress	The addresses of all data write accesses are traced (and the prog. flow).
Data	The data values of all read/write accesses are traced (and the program flow), but not the addresses of the read/write accesses.
ReadData	Data values of all read accesses are traced (and the program flow).
WriteData	Data values of all write accesses are traced (and the program flow).
Only	Only data is traced (address and data of all read/write accesses) but program flow is not traced.
OnlyAddress	Only data addresses are traced (address of all read/write accesses, but no data). Program flow is not traced.
OnlyData	Only data values are traced (data of all read/write accesses, but no addresses). Program flow is not traced.

FIFO Overflow

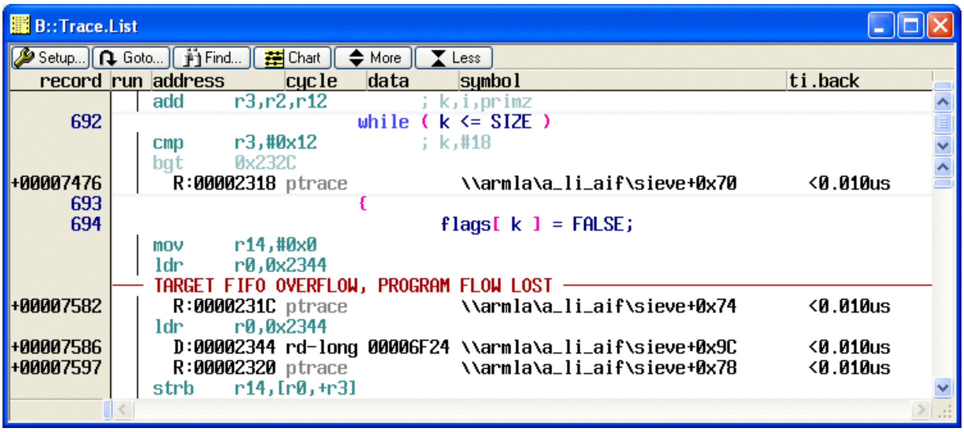


- Broadcasting the program flow requires usually a low bandwidth and can be done without any problem.
- Broadcasting the data flow generates much more traffic on the trace port. In order to prevent an overloading of the trace port a FIFO is connected upstream of TRACEPKT/TRACEDATA. This provides intermediate storage for all trace packets that cannot be output via TRACEPKT/TRACEDATA immediately.

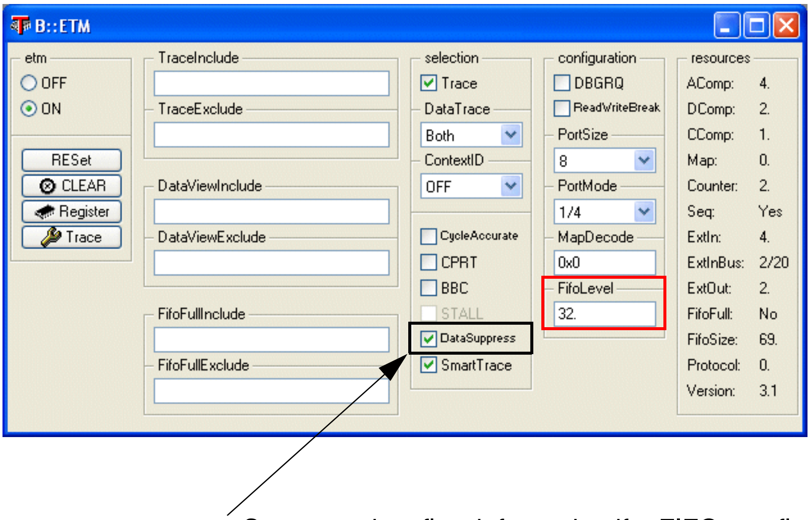


The ETMv3.x provides the FifoSize in a ETM configuration register

Under certain circumstances it is possible that so much trace information is generated, that the FIFO overflows.

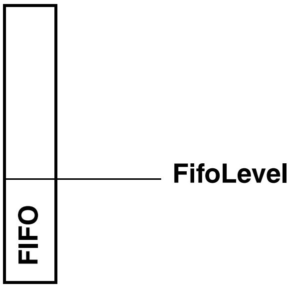


If a FIFOFULL is likely to happen, the ETM suppresses the output of the data flow information.



Suppress data flow information if a FIFO overflow is likely to happen

The data suppression is not indicated in the trace.



You can define a **FifoLevel**. If less the FifoLevel bytes are empty in the FIFO:

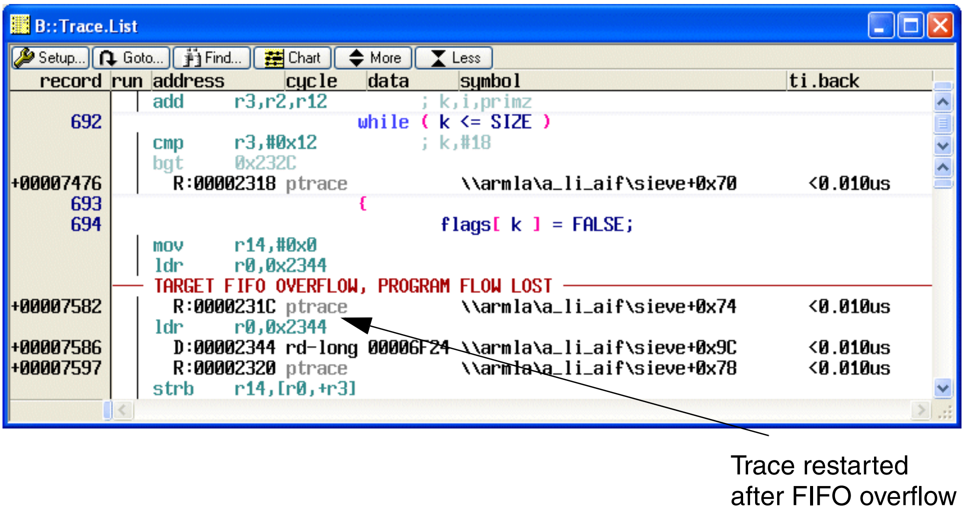
- The processor is stalled until the number of empty bytes is higher the FifoLevel again (ETMv1.x).
- No data information is broadcasted until the number of empty bytes is higher the FifoLevel again (ETMv3.x)

Recommendation for FifoLevel is ~2/3 of the FIFO size.

No further trace packets are accepted by the FIFO if a FIFO overflow occurs.

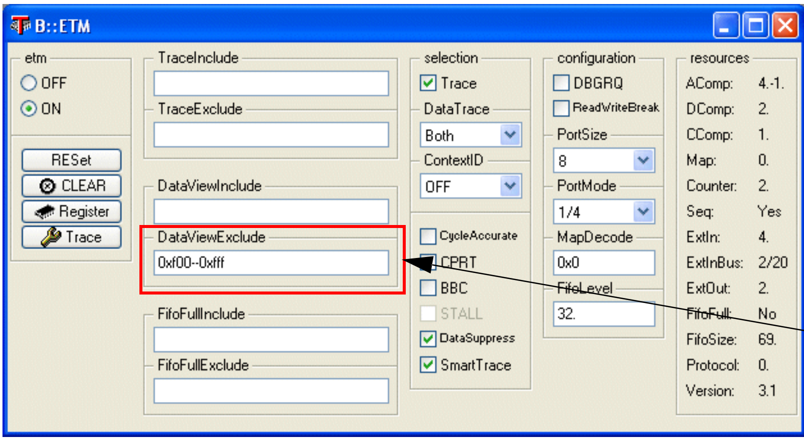
The ETM signals a FIFO overflow via the pipeline information and ensures, that the FIFO is completely emptied. Once the FIFO memory is ready to receive again trace packets:

- A ***Trace restarted after FIFO overflow*** is output via the pipeline information
- The full address of the instruction just executed is output.



What can I do to reduce the risk of a FIFO overflow?

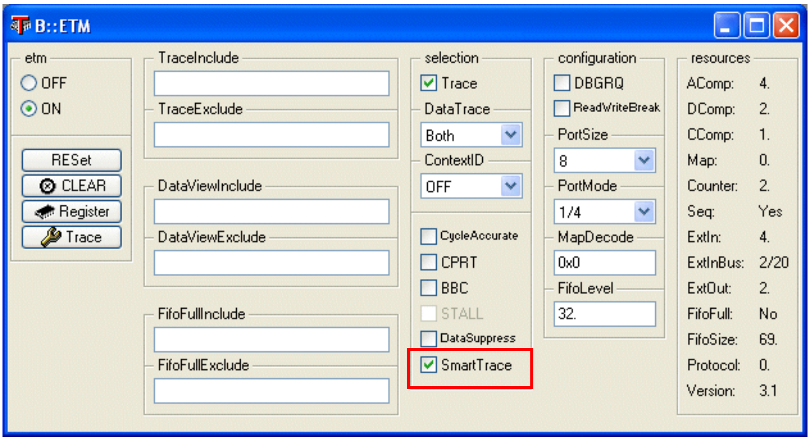
- Set the correct PortSize.
- Restrict DataTrace to read cycles (write accesses can be reconstructed via CTS).
- Restrict DataTrace to write cycles (a FIFO overflow becomes less likely).
- Reduce the broadcast of TraceData information by using a trace filter.
- Exclude the stack area from the data trace.



Exclude the stack area from the data trace

- Stall the core / supress the data flow when a FIFO overflow is likely to happen.

The gaps in the trace recording resulting from FIFO overflow can be reconstructed in most cases by using the SmartTrace and CTS.



It is not possible to reconstruct:

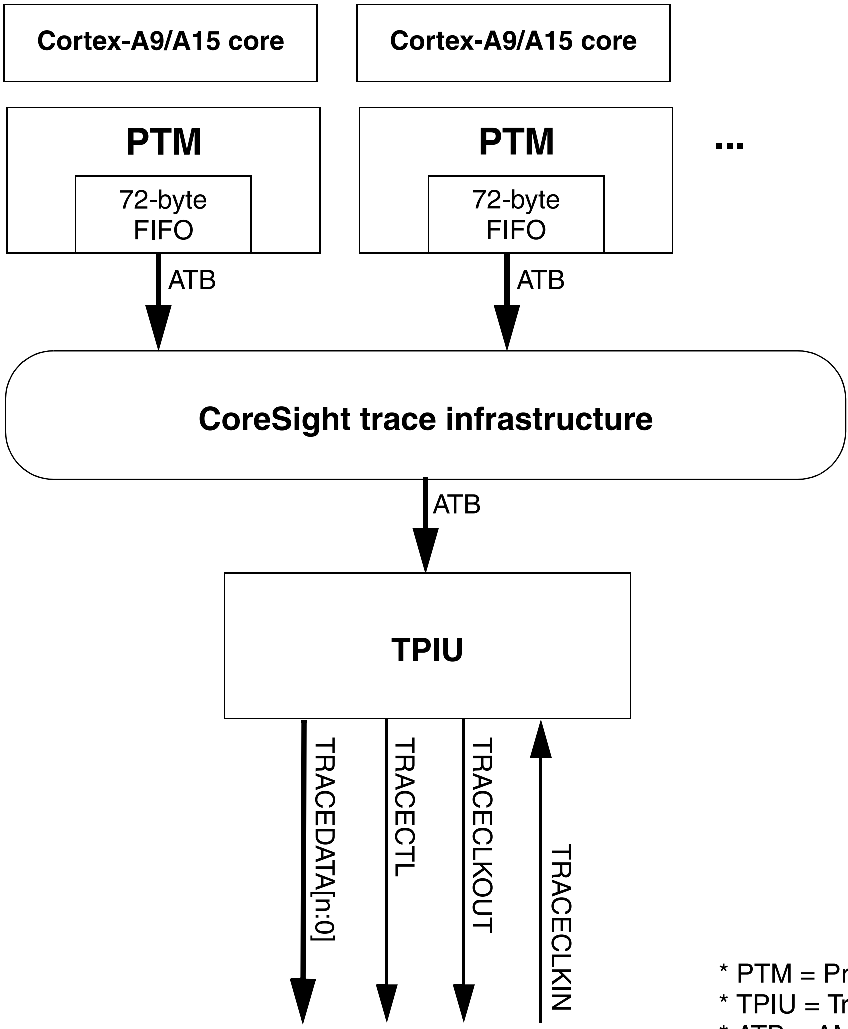
- Exceptions occurring in the trace gaps.
- Port reads occurring in the trace gaps.

PTM stands for Program Trace Macrocell
PTM is also known as PFT. PFT stands for Program Flow Trace.

PTM offers a stronger trace compression comparing to ETMv3 but does not offer any kind of data trace (neither data address nor data value)

The PTM is only used for Cortex-A9, Cortex-A15 and Cortex-A17 processor cores.

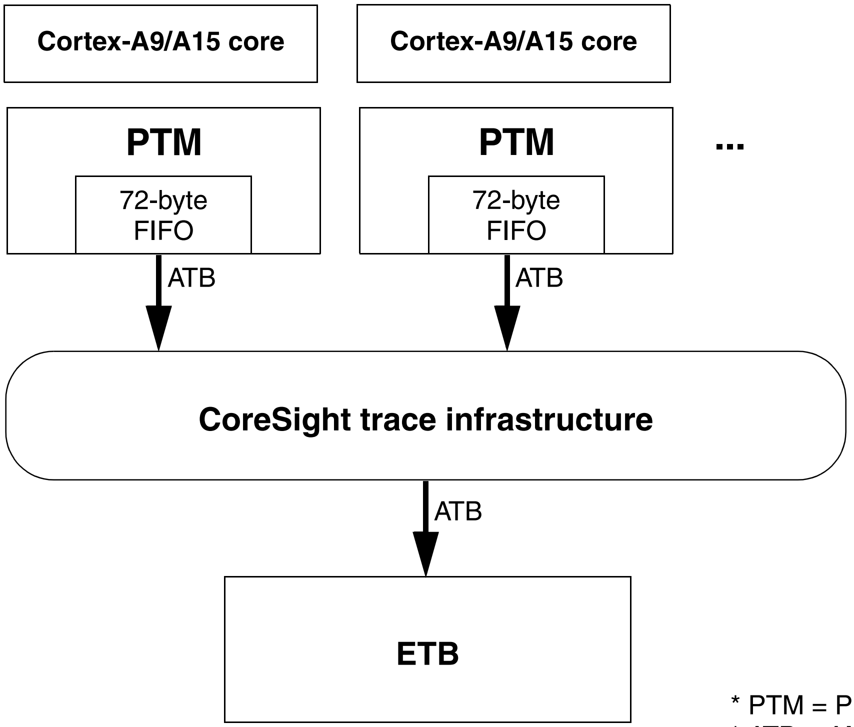
Simplified block diagram for off-chip trace with parallel interface, training-relevant components only:



* PTM = Program Flow Trace Macrocell
* TPIU = Trace Port Interace Unit
* ATB = AMBA Trace Bus

TRACEDATA[n:0]	Trace packets
TRACECLT	Trace Control (optional)
TRACECLKOUT	Trace Clock from Target
TRACECLKIN	not used

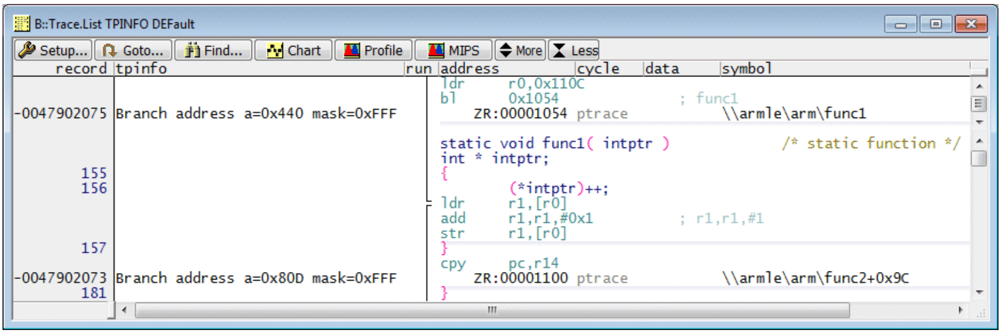
Simplified block diagram for on-chip trace, training-relevant components only:



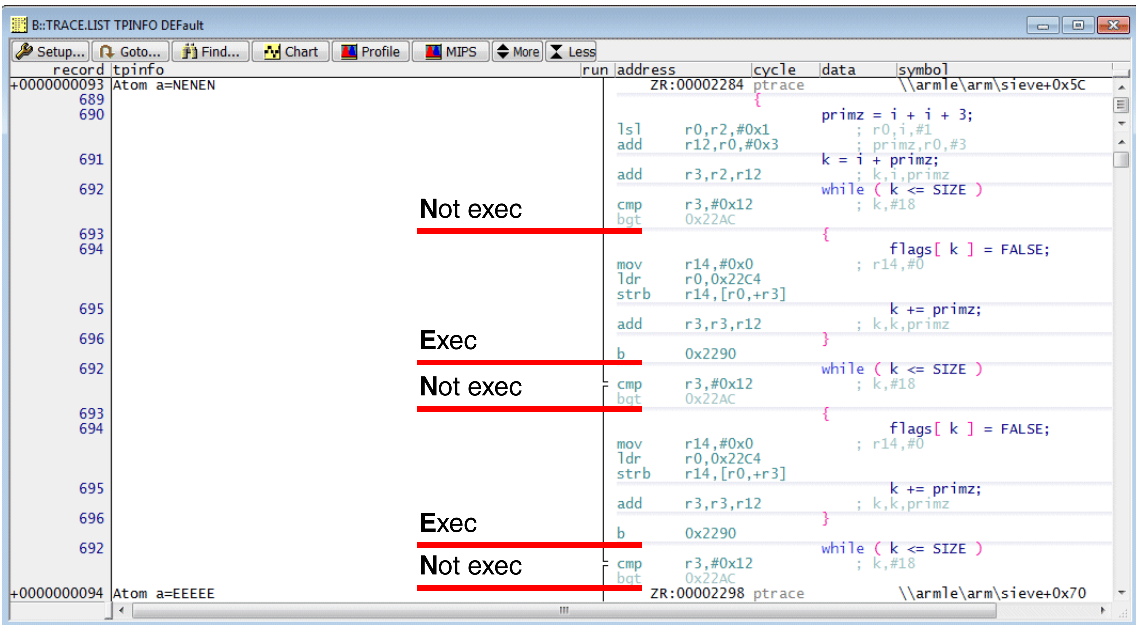
* PTM = Program Flow Trace Macrocell
* ATB = AMBA Trace Bus

The PTM generates trace information on certain points in the program (waypoints). TRACE32 needs for a full reconstruction of the program flow also the source code information. Waypoints are:

- Indirect branches, with branch destination address and condition code



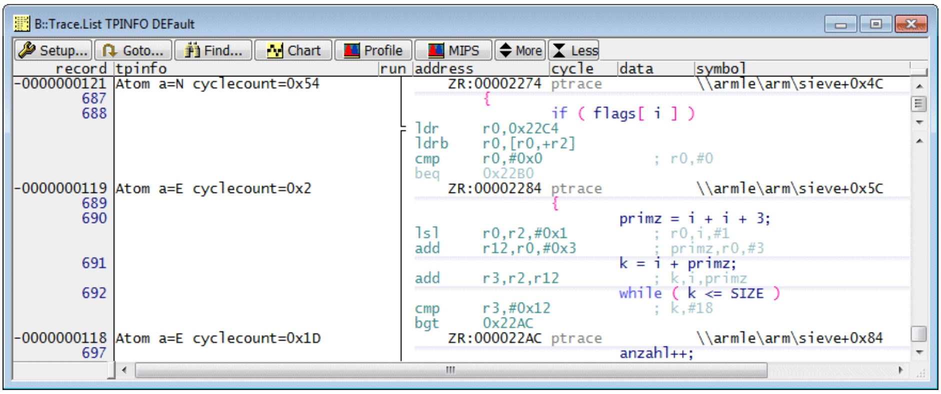
- Direct branches with only the condition code



- Instruction barrier instructions
- Exceptions, with an indication of where the exception occurred
- Changes in processor instruction set state
- Changes in processor security state
- Context ID changes
- Start and stop of the program execution

The PTM can be configured to provide the following additional information:

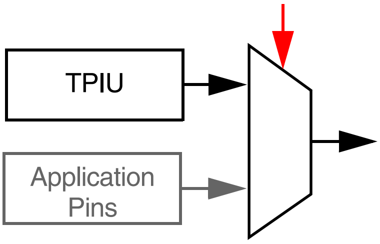
- Cycle count between traced waypoints



- Global system timestamps
- Target addresses for taken direct branches

1. Enable TPIU pins

The TPIU pins need to be enabled, if they are multiplexed with other signals.

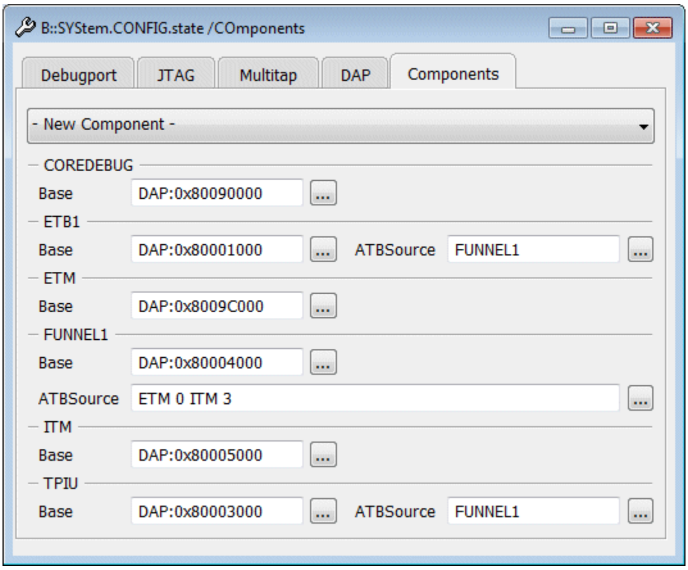


PER.Set.simple *<address>*|*<range>* [%*<format>*] *<string>*	Modify configuration register/on-chip peripheral

```
PER.Set SD:0x111D640 %Word 0x9AA0     ; Enable TPIU functionality on
PER.Set SD:0x111D6A4 %Word 0x2901     ; GPIO's

PER.Set 0x111600D %Long %LE 0x01E     ; Enable CLK
```

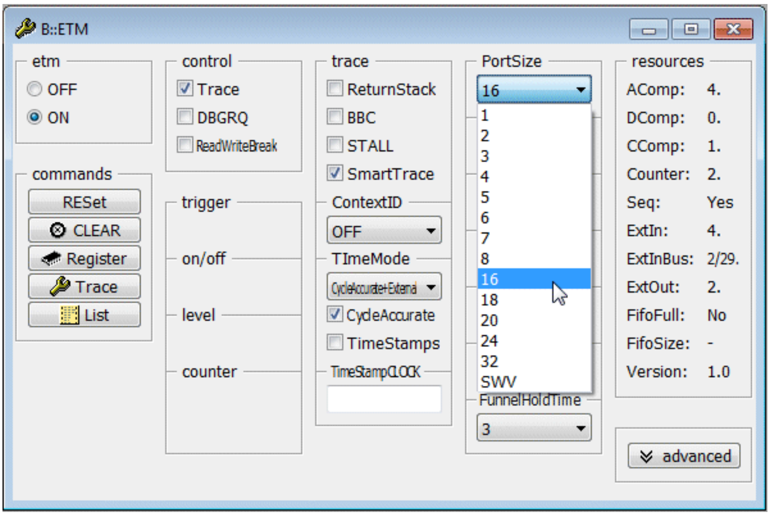
The CoreSight trace infrastructure is automatically configured by TRACE32 PowerView if your chip and its infrastructure is known. Otherwise please contact your local TRACE32 support.

SYStem.CONFIG.state /COmponents	Check configuration of CoreSight infrastructure



3. Specify PortSize

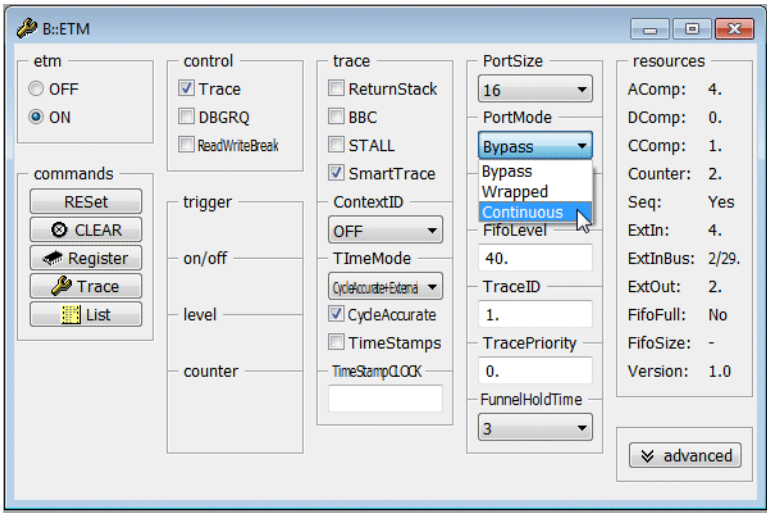
PortSize specifies how many TRACEDATA pins are available on your target to export the trace packets.



ETM.PortSize *<size>*

ETM.PortMode specifies the Formatter operation mode.

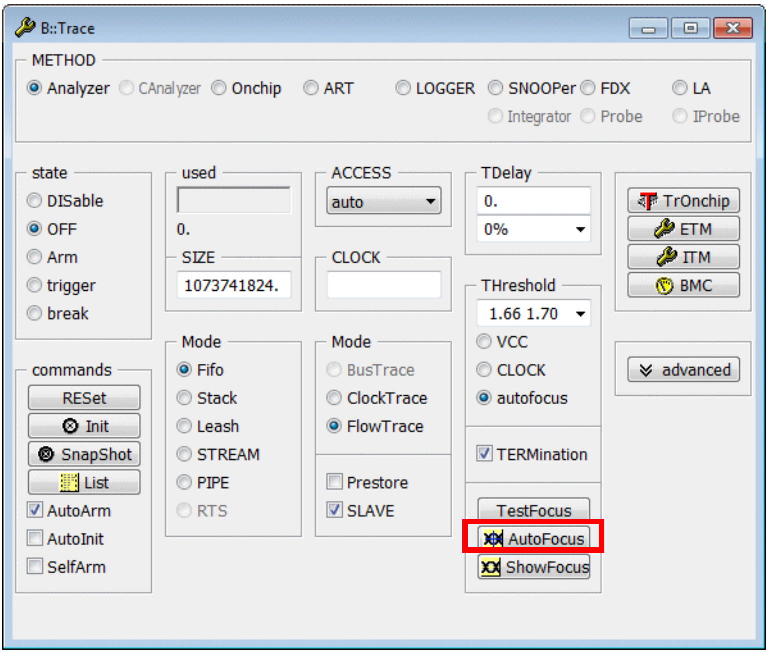
The TPIU/ETB merges the trace information generated by the various trace sources within the multicore chip to a single trace data stream. A trace source ID (e.g **ETM.TraceID**, **ITM.TraceID**, **HTM.TraceID**) allows to maintain the assignment between trace information and its generating trace source. The task of the Formatter within the TPIU/ETB is to embed the trace source ID within the trace information to create this single trace stream.



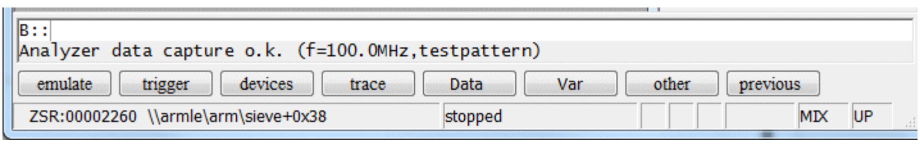
Bypass	There is only one trace source, so no trace source IDs is needed. In this operation mode the trace port interface needs to provide the TRACECTL signal.
Wrapped	The Formatter embeds the trace source IDs. The TRACECLT signal is used to indicate valid trace information.
Continuous	The Formatter embeds the trace source IDs. Idles are generated to indicate invalid trace information. The TRACE32 preprocessor filters these idles in order to record only valid trace information into the trace memory.

ETM.PortMode Bypass | Wrapped | Continuous

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



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



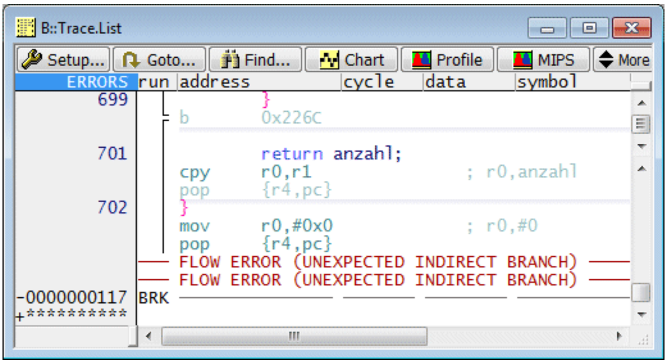
The following setting may be of interest. They are explained in detail later in this training:

ETM.ReturnStack [ON | OFF]	May help to reduce the amount of generated trace information.

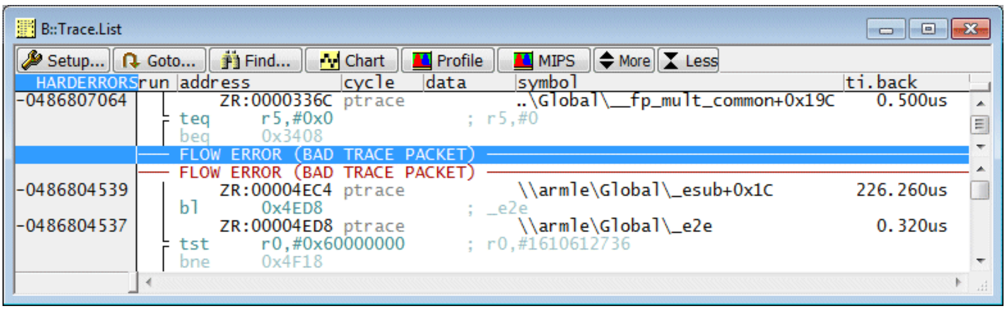
ETM.ContextID 8 | 16 | 32 | OFF	Is required for OS-aware tracing

ETM.TImeMode *<mode>*	Is required for OS-aware tracing
ETM.CycleAccurate [**ON** | **OFF**]	Enable cycle-accurate tracing
ETM.TimeStamps [**ON** | **OFF**]	Enable global timestamp
ETM.TimeStampCLOCK *<frequency>*	Specify clock of global timestamp

FLOWERROR: The trace information is not consistent with the code image in the target memory. This can happen when the code in the target memory was changed or the trace information was corrupted e.g. because of crosstalk on the external trace pins.



HARDERROR: The trace port pins are in an invalid state.
This happens if the trace information was corrupted e.g. because of crosstalk on the external trace pins. In rare cases this can be caused by a bug in the chip.



TRACE32 normally uploads only those records from the physical trace memory to the host, that are required by the current trace display/analysis window. To upload the complete trace contents to the host for diagnosis, the following commands are recommended:

Trace.FLOWPROCESS

Trace.Chart.sYmbol

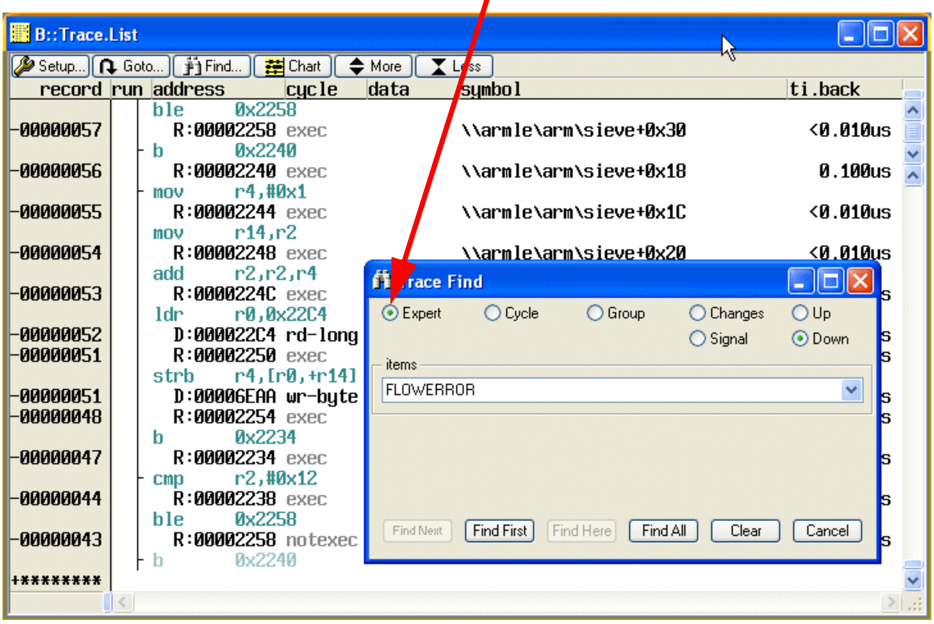
To check the number of FLOWERRORS in the trace use the following command:

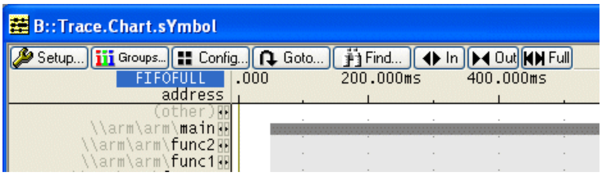
```
PRINT %Decimal Trace.FLOW.ERRORS()
```

If the trace contains FLOWERRORS / HARDERRORS, please try to set up a proper trace recording before you start to evaluate or analyse the trace contents. See also the section "Diagnosis" in "ARM-ETM Trace" (trace_arm_etm.pdf).

To find the FLOWERRORS / HARDERRORS in the trace use the keyword FLOWERROR in the ***Trace.Find*** window.

Use the ***Expert*** Find page to find the FLOWERRORs in the trace



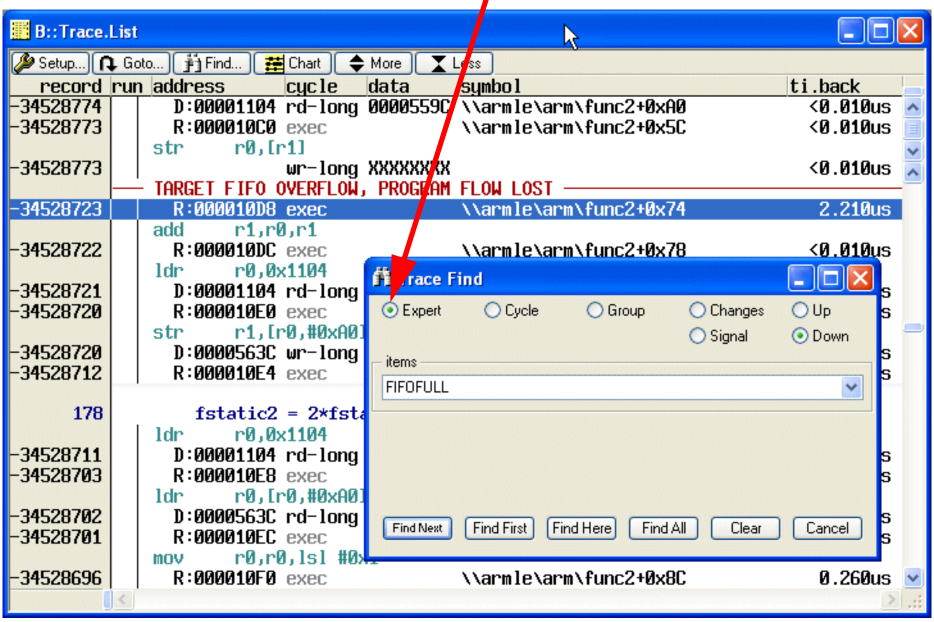


To check the number of FIFOFULLs in the trace use the following command:

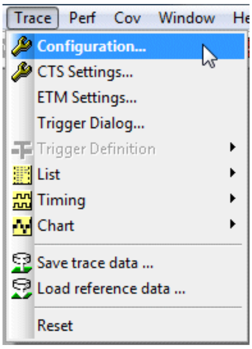
```
PRINT %Decimal Trace.FLOW.FIFOFULL()
```

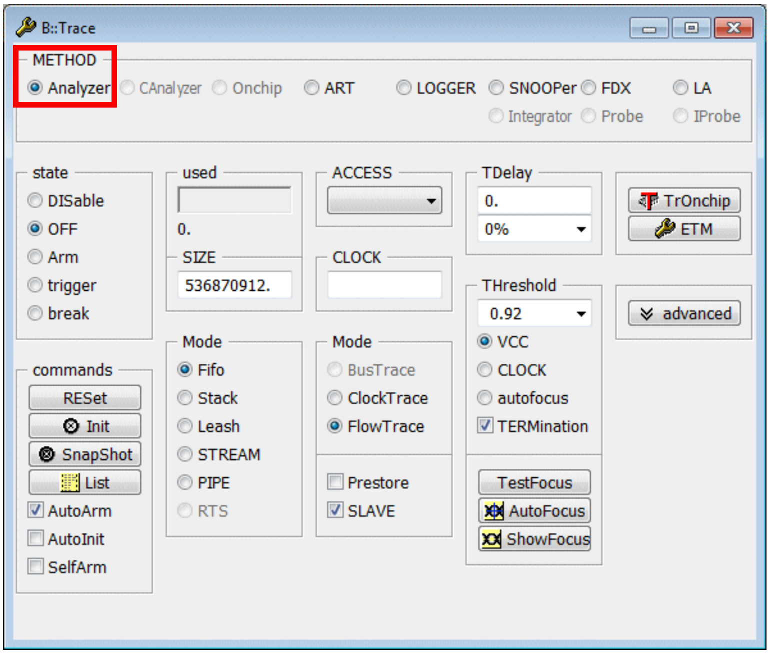
To find the FIFOFULLs in the trace use the keyword FIFOFULL in the ***Trace.Find*** window.

Use the ***Expert*** Find page to find the FIFOFULLs in the trace



Source for the Recorded Trace Information





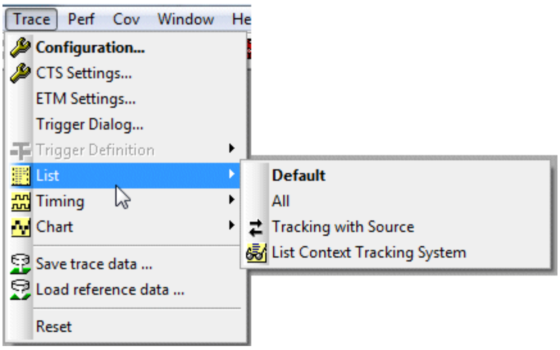
If TRACE32 is started when a PowerTrace hardware and an ETM preprocessor is connected, the source for the trace information is the so-called **Analyzer** (**Trace.METHOD Analyzer**).

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

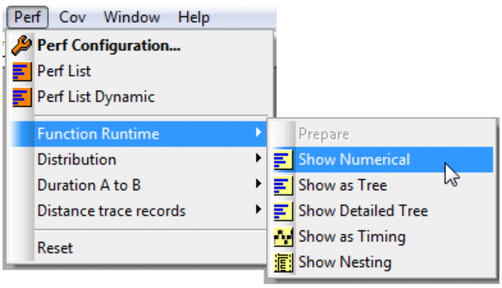
1. **Trace** is an alias for **Analyzer**.

```
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

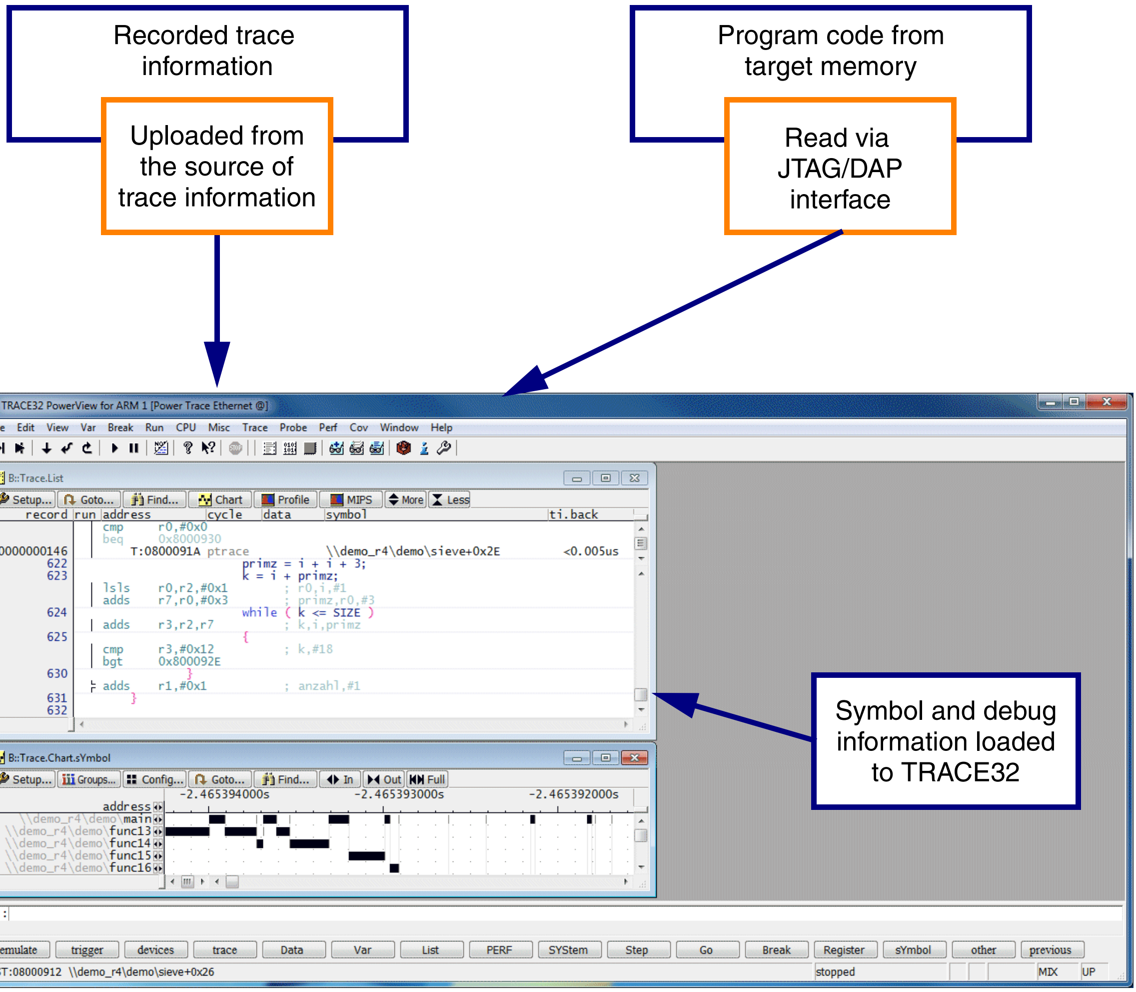
This ETM Training uses always the command group **Trace**. If you are using a CombiProbe or the on-chip ETB as source of the trace information, just select the appropriate trace method and most features will work as demonstrated for the trace method Analyzer.

```
Trace.METHOD CAnalyzer          ; select the CombiProbe as source
                                ; for the trace information

Trace.METHOD Onchip             ; select the ETB as source for the
                                ; trace information
```

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/DAP interface.
- The symbol and debug information already loaded to TRACE32.



Influencing Factors on the Trace Information

The main influencing factor on the trace information is the ETM itself. 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.

An ETM can provide the following trace information:

- **Program:** Instruction flow mainly based on the target addresses of taken indirect branches
- **Non-Branch Conditionals:** The ARM instruction sets allow also non-branch instruction to be conditional. The trace can contain information if the condition of every conditional instruction has been met or only if the condition of the branch instructions have been met.
- **Data Address:** The address to/from a data access is writing/reading data-
- **Data Value:** The data value loaded/stored by a read/write operation
- **Context ID:** Values of Context ID changes, mainly used to indicate task/process changes and changes of the virtual address space
- **Cycle-count (CC):** Number of clock cycles between executed instructions

The table below shows what trace information is generated by the various ARM Cortex cores.

Core	ETM Version	Program Flow	non-branch conditionals	Data Address	Data Value	Context ID	Cycle Count	bits per instruction
ARM7 **ARM9**	ETMv1	■	■	■	■	≥ ETMv1.2	■	≥ 8.00
ARM9	CoreSight ETMv3	■	■	■	■	■	■	~1.20
ARM11	(CoreSight) ETMv3	■	■	■	■	■	■	~1.20
Cortex-M3/M4 **Cortex-M23**	CoreSight ETMv3	■	■	(DWT)	(DWT)	-	-	~1.20
Cortex-M7 **Cortex-M33**	ETMv4 (Cfg. 3)	■	■	(DWT)	(DWT)	-	■	~0.35
Cortex-R4 **Cortex-R5**	CoreSight ETMv3	■	■	■	■	■	■	~1.20
Cortex-R7/R8 **Cortex-R52**	ETMv4 (Cfg. 1)	■	■	■	■	■	■	~0.40
Cortex-A5 **Cortex-A7**	CoreSight ETMv3	■	■	■	■	■	■	~1.20
Cortex-A8	CoreSight ETMv3	■	■	■	-	■	■	~1.20
Cortex-A9 **Cortex-A15** **Cortex-A17**	PTM	■	-	-	-	■	■	~0.30
Cortex-A3x **Cortex-A5x** **Cortex-A7x**	ETMv4 (Cfg. 2)	■	-	-	-	■	■	~0.30

The column "Bits per Instruction" is base on values stated by ARM for program traces without data address/value trace, without conditional non-branch instructions and without cycle-count or internal-timing information (only external tool-base timing).

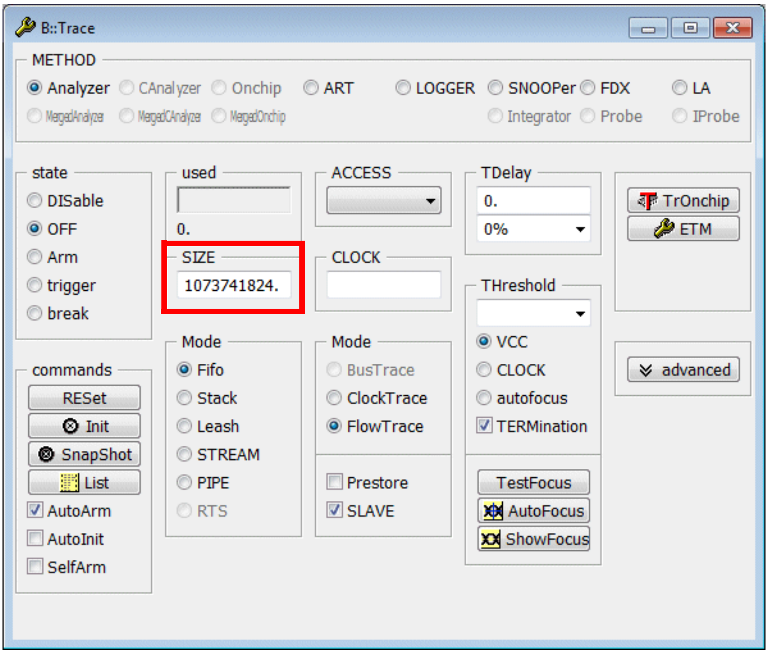
The ETM can provide also a number of comparators to restrict the generated trace information to the information of interest. They are introduced in detail in the section **Trace Control by Filter and Trigger** of this training.

Settings in the TRACE32 Trace Configuration Window

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:** STREAM mode specifies that the trace information is immediately streamed to a file on the host computer. Peak loads at the trace port are intercepted by the TRACE32 trace tool, which can be considered as a large FIFO.

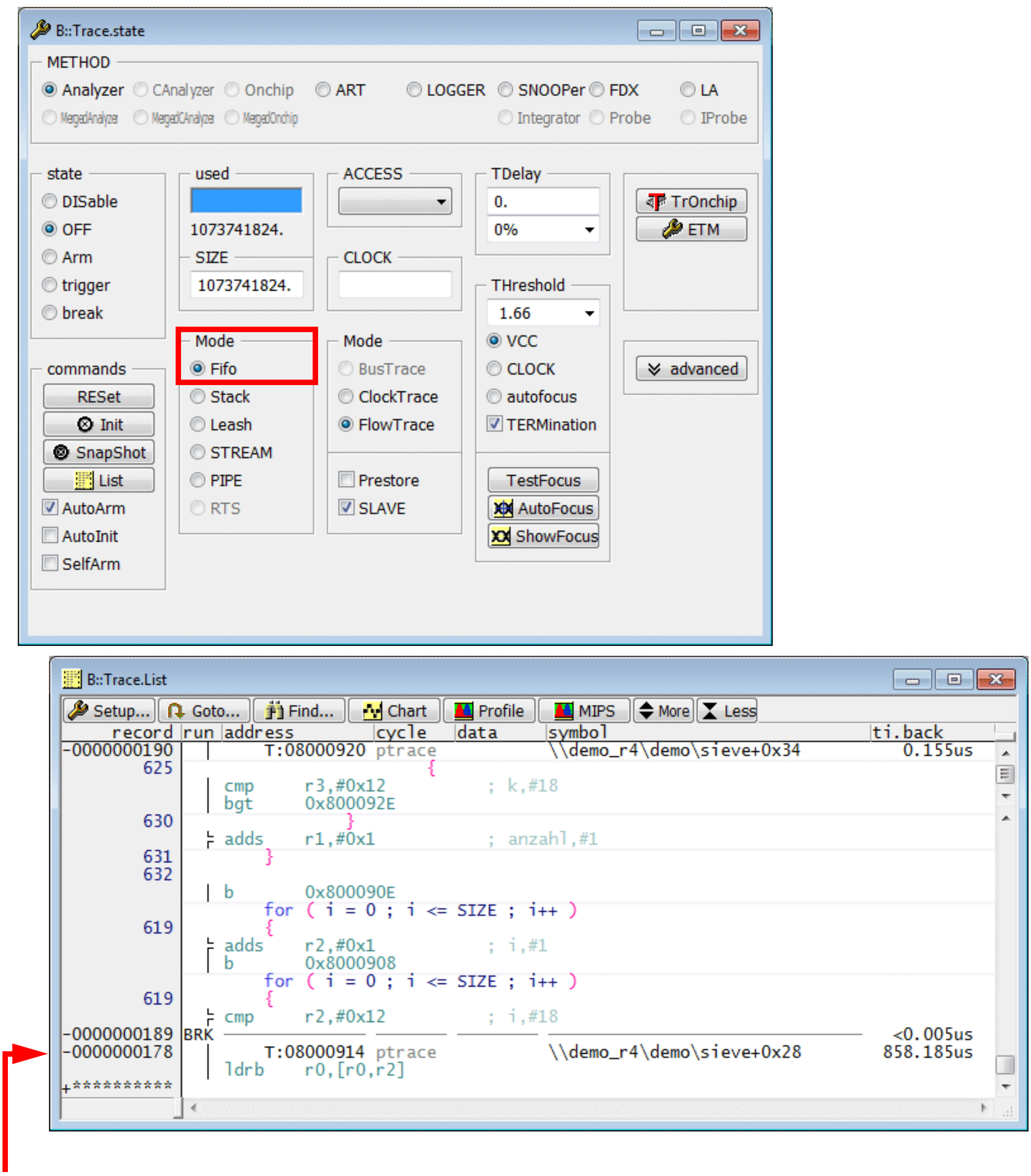
  STREAM mode allows a trace memory of several T Frames.

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

- **PIPE Mode:** PIPE mode specifies that the trace information is immediately streamed to the host computer. There it is distributed to a user-defined trace analysis application. This mode is still under construction and will be used mainly for software-generated trace information.

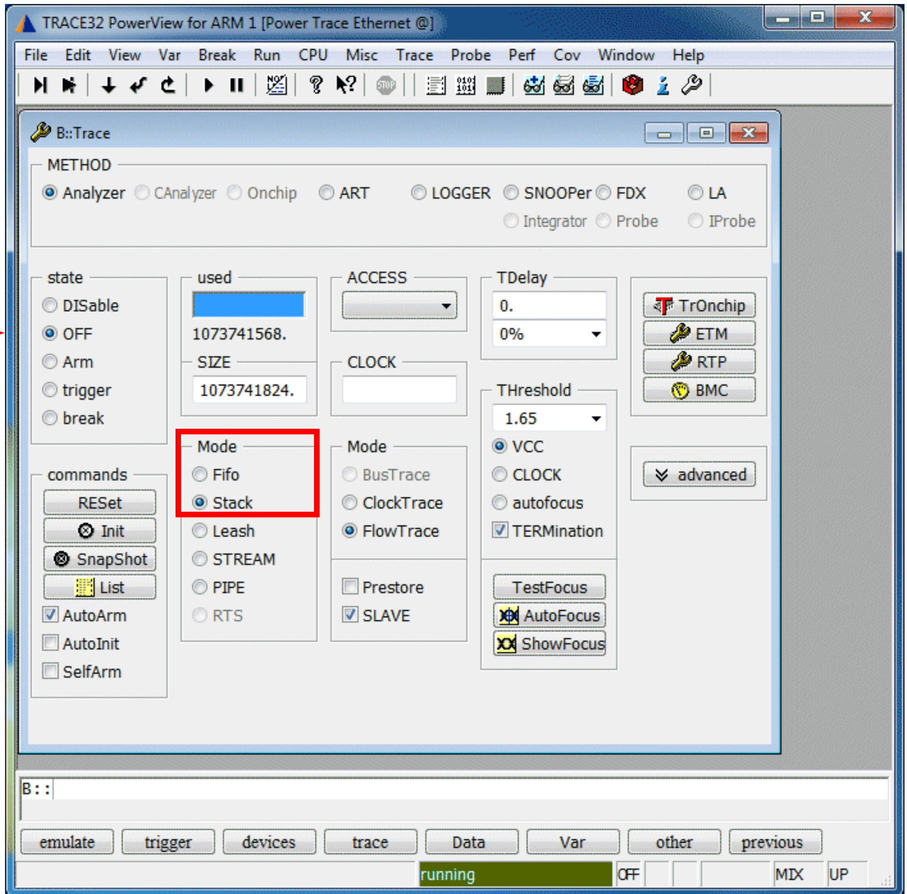
- **RTS Mode**: The RTS radio button in only an indicator that shows if Real-time Profiling is ON or OFF. For details on Real-time Profiling refer to the **RTS** command group.
  RTS is available for ETMv3 and ETMv4.

```
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.

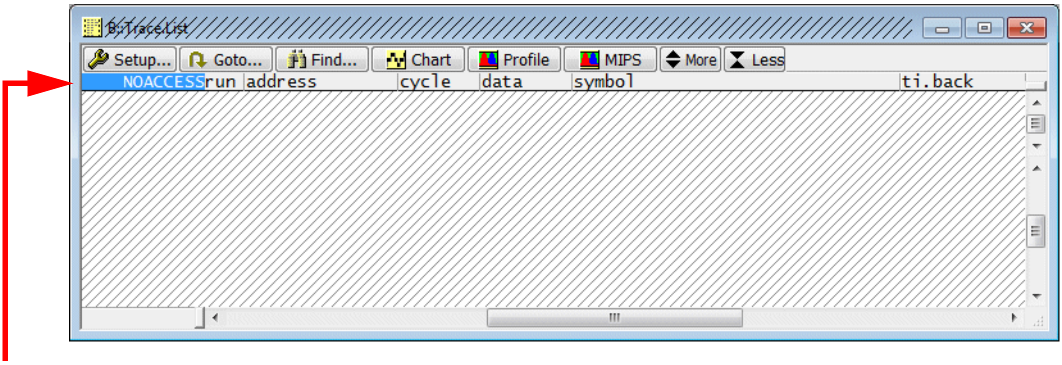
```
Trace.Mode Stack                        ; when the trace memory is full
                                        ; the trace recording is stopped

                                        ; 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** indicates that the program execution is running,

OFF indicates that the trace recording is switched off

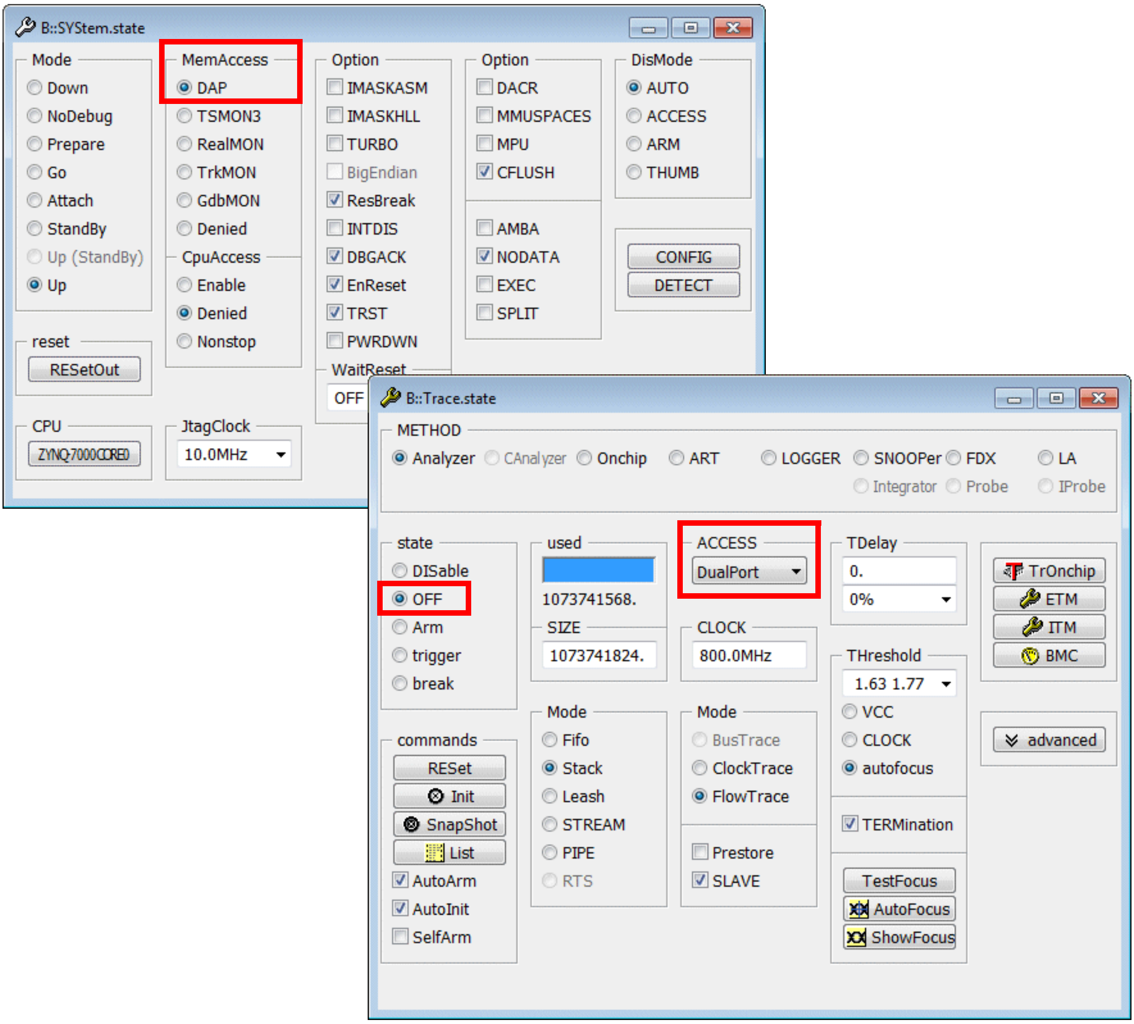


By default, the trace information can not be displayed while the program execution is running. TRACE32 has **NOACESS** to the target memory.

There a three alternative ways to solve the **NOACCESS** issue:

1. **Stop the program execution.** Then TRACE32 has access to the target memory.

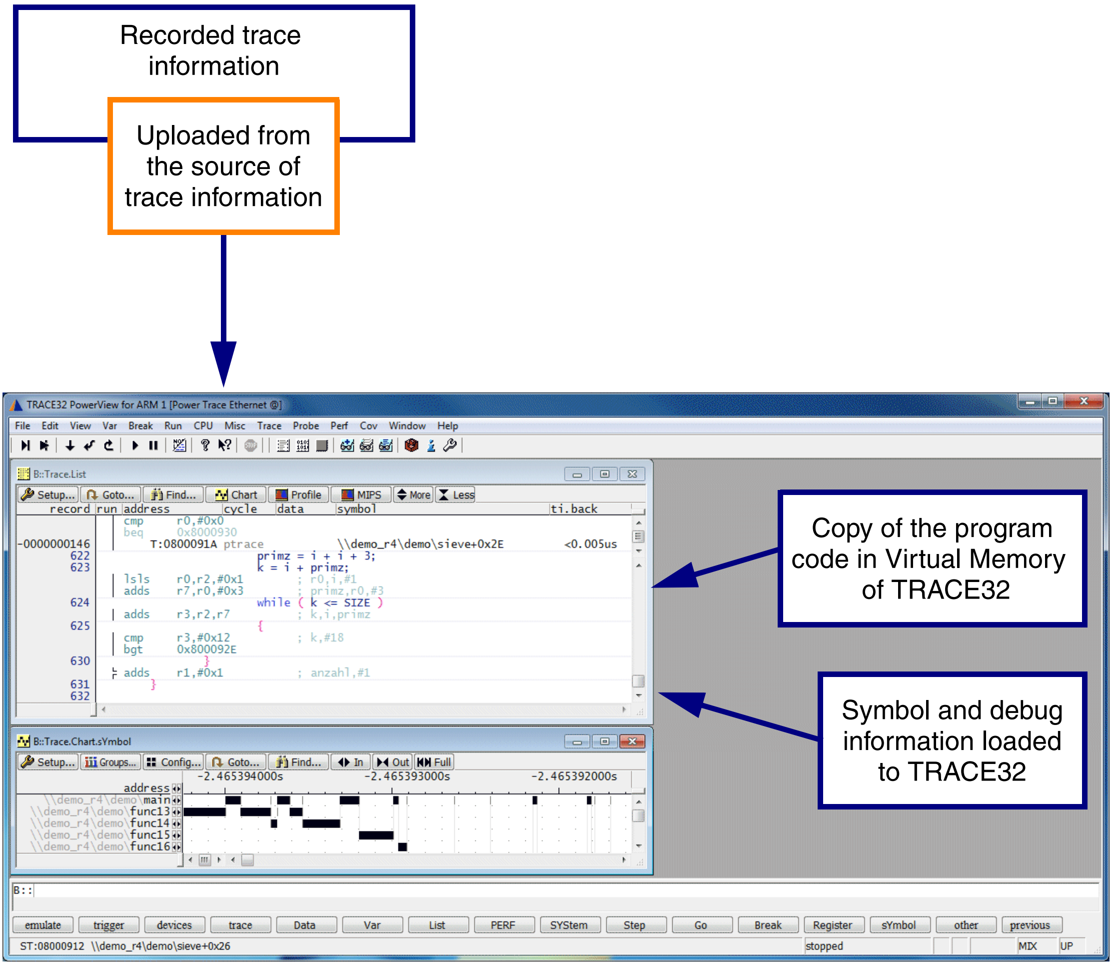
2. **Enable the run-time memory access** (if supported by your chip) and advise TRACE32 to read the target memory via this run-time memory access for trace decoding.



SYStem.MemAccess DAP	Enable run-time memory access.
Trace.ACCESS DualPort	Advise TRACE32 to read the target memory via the run-time memory access for trace decoding.

3. **Provide the program code to TRACE32 via the so-called TRACE32 Virtual Memory.**

If there is a copy of the program code in the TRACE32 Virtual Memory TRACE32 reads the code from there, if an access to the target memory is not possible.



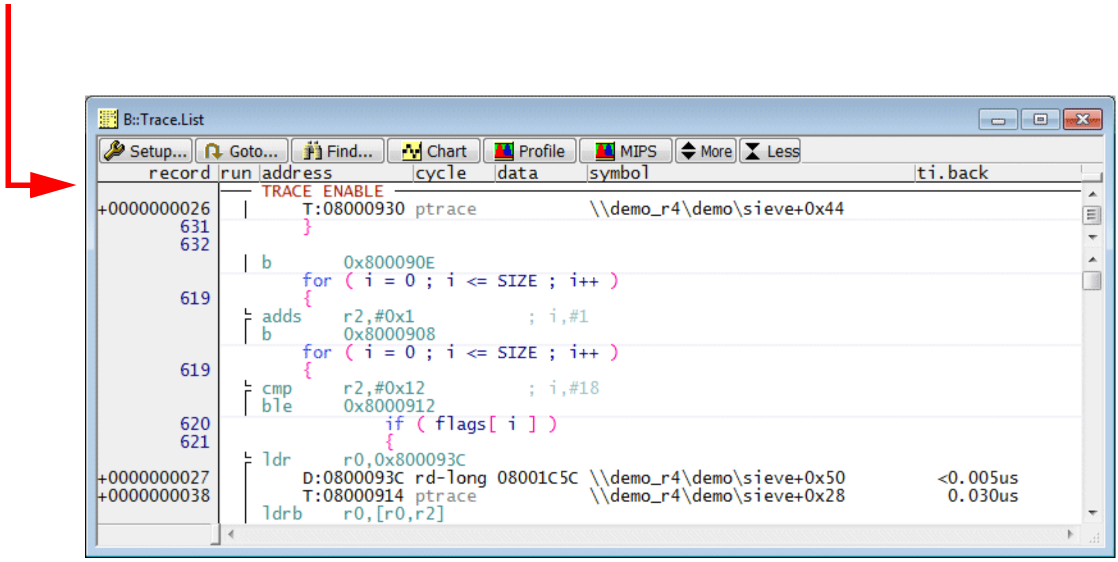

```
; load the code from the file demo_r4.axf to the target and the
; TRACE32 Virtual Memory
Data.LOAD.Elf demo_r4.axf /PlusVM
```

```
Data.LOAD.Elf demo_r4.axf

; …

; load the program code to the TRACE32 Virtual Memory
Data.LOAD.Elf demo_r4.axf /VM /NosYmbol /NoClear
```

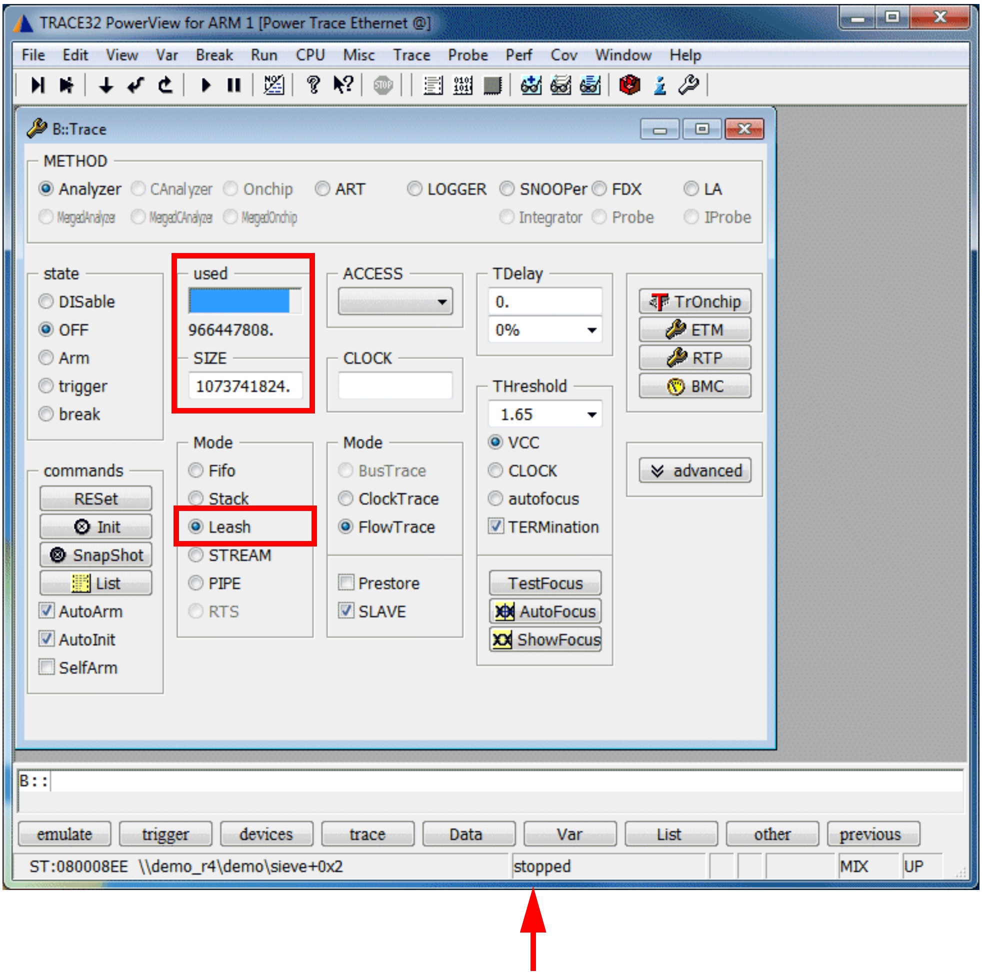
Caution:	Please make sure that the Virtual Memory of TRACE32 always provides an up-to-date version of the program code. Out-of-date program versions will cause FLOW ERRORs.

The trace contents is displayed as soon as TRACE32 has access to the program code.

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 then the maximum **SIZE.**

```
Trace.Mode STREAM                     ; STREAM the recorded trace
                                      ; information to a file on the host
                                      ; computer
                                      ; (off-chip trace only)

                                      ; 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 TFrames.

- Streaming mode required 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 temp directory (**OS.PresentTemporaryDirectory()**).

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

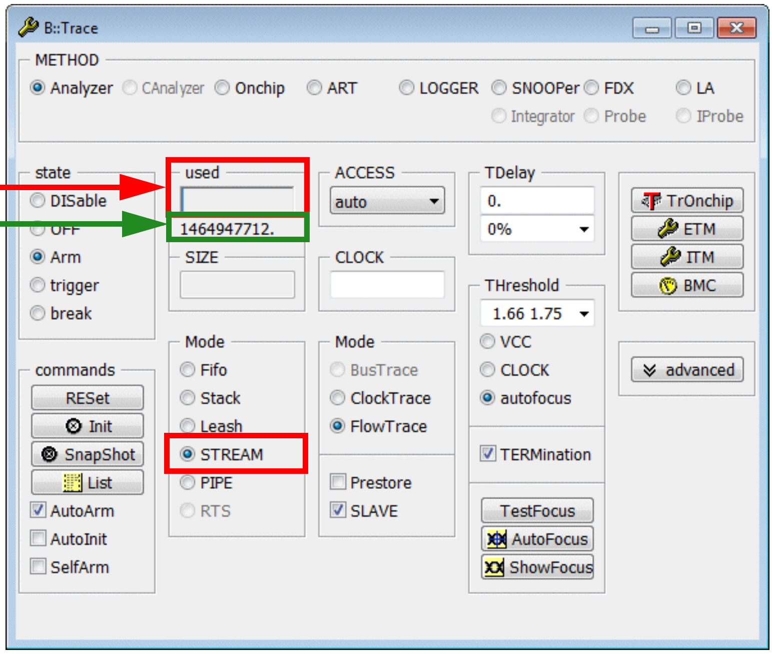
```
Trace.STREAMFILE d:\temp\mystream.t32     ; specify the location for
                                          ; your streaming file
```

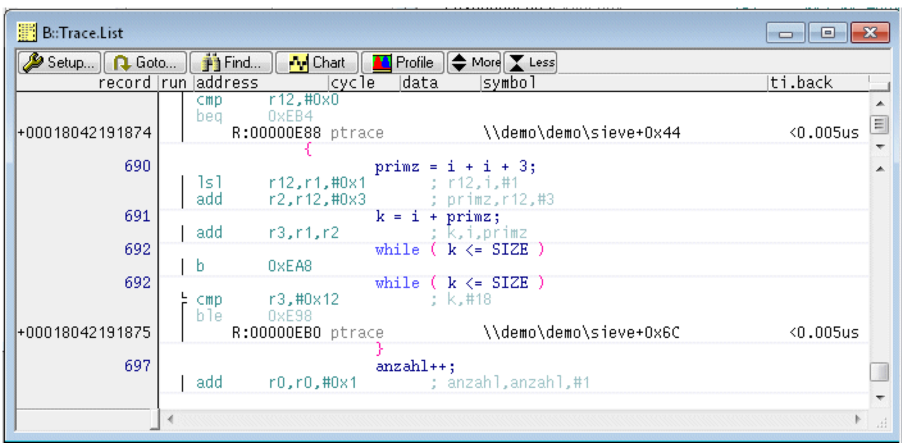
Please be aware that the streaming file is deleted as soon as you de-select the STREAM mode or when you exit TRACE32.

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

used graphically indicates the number of records buffered by the TRACE32 trace memory

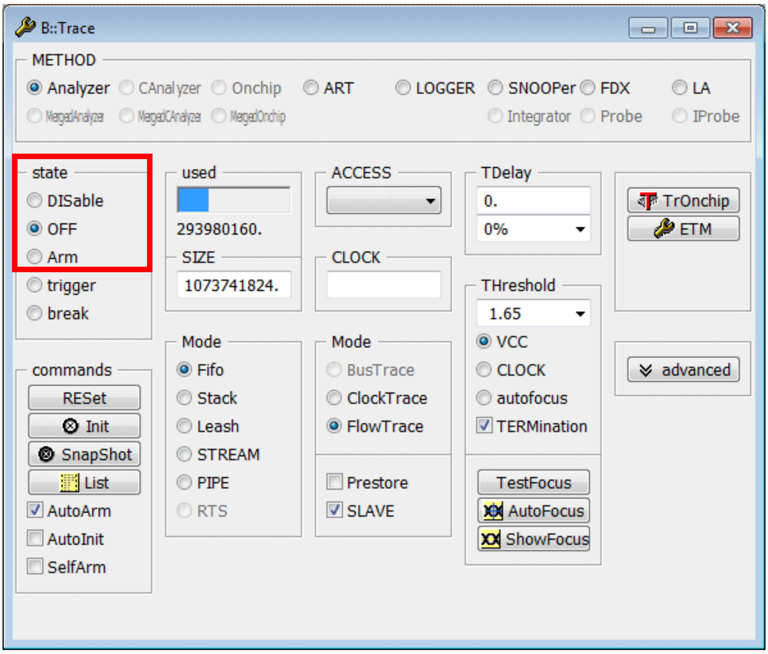
used numerically indicates the number of records saved to the streaming file





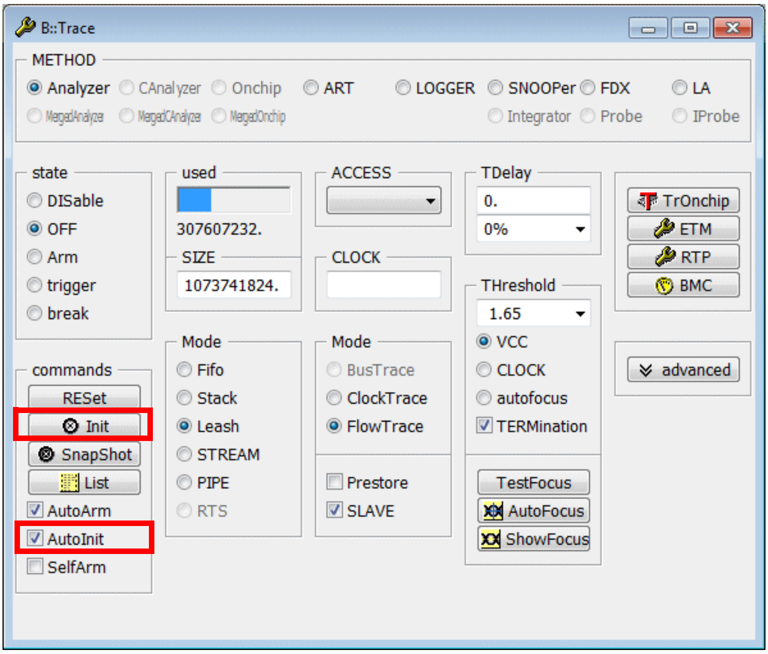
STREAM mode can generate very large record numbers

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



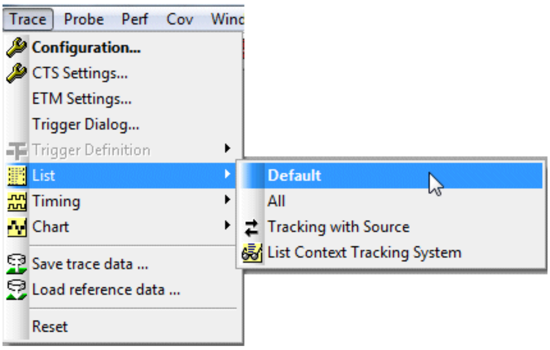
States of the Trace	
DISable	The trace is disabled.
OFF	The trace is not recording. The trace contents can be displayed.
Arm	The trace is recording. The trace contents can not be displayed.

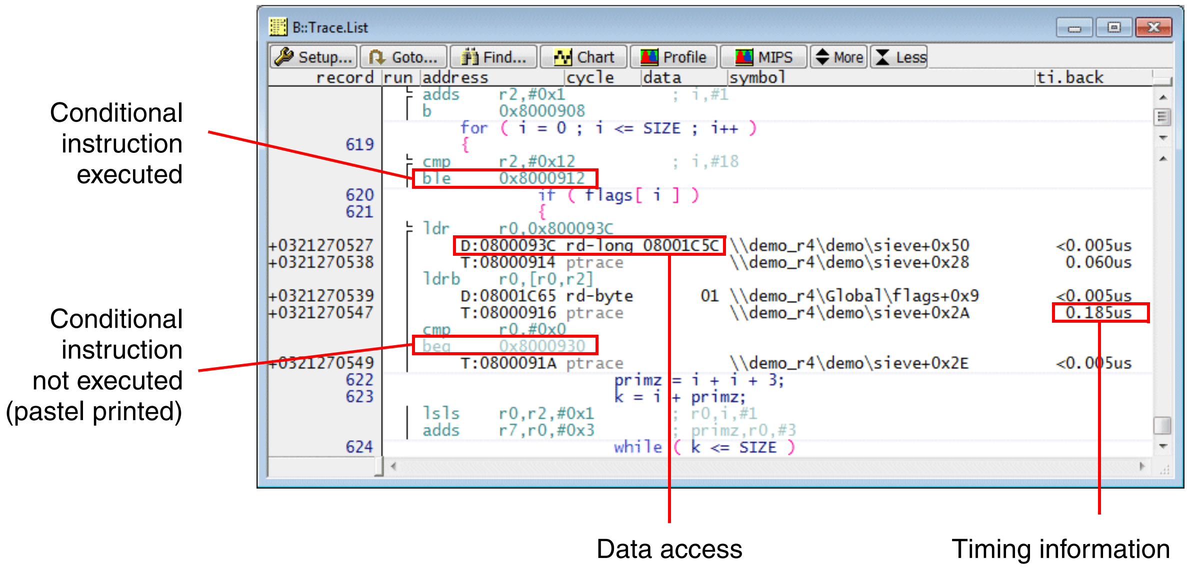
The Trace states **trigger** and **break** are introduced in detail later in this training.

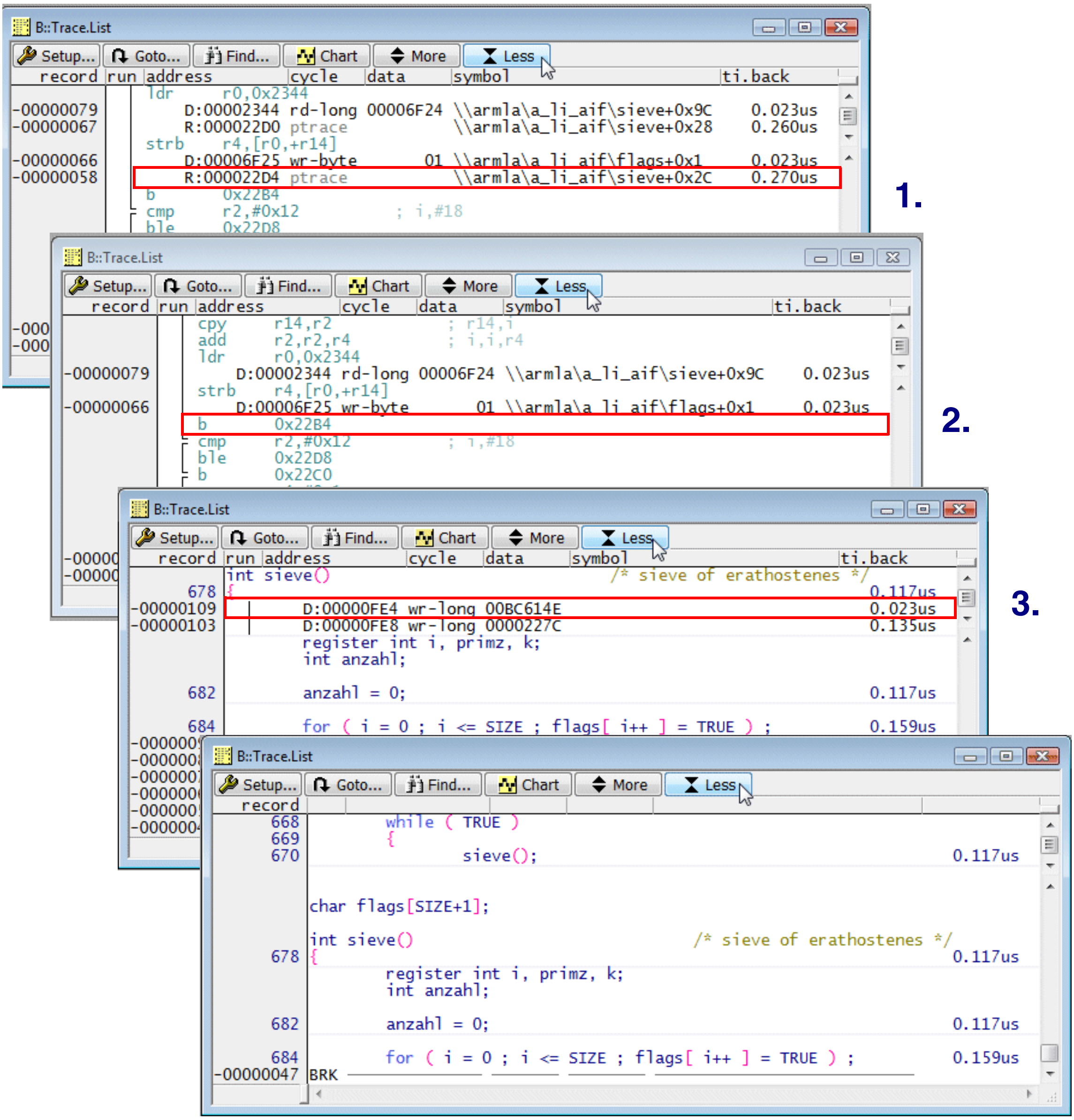


Init Button	Delete the trace memory. All other settings in the Trace Configuration window remain valid.
AutoInit CheckBox	ON: The trace memory is deleted whenever the program execution is started (Go, Step).

Default Listing

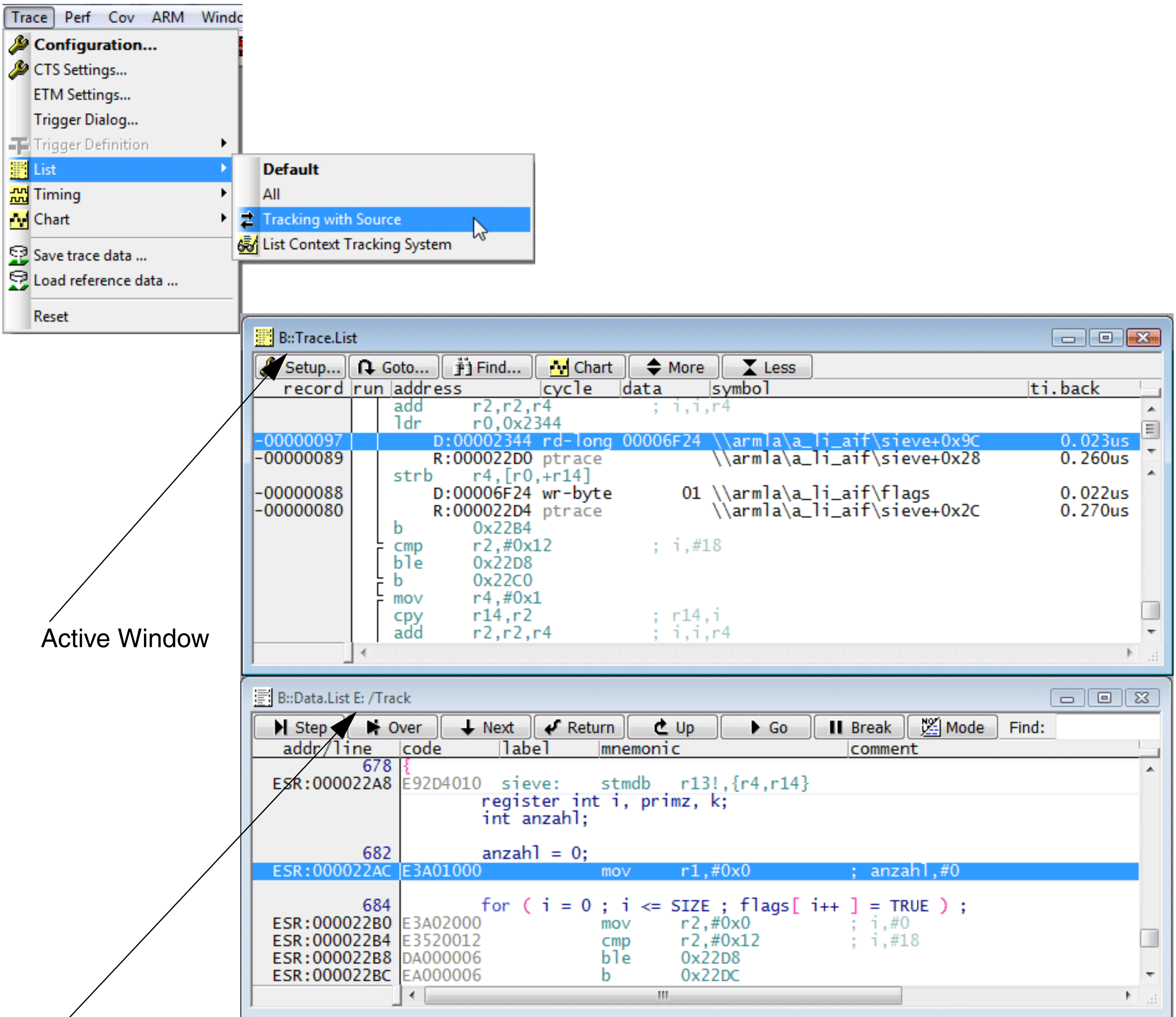






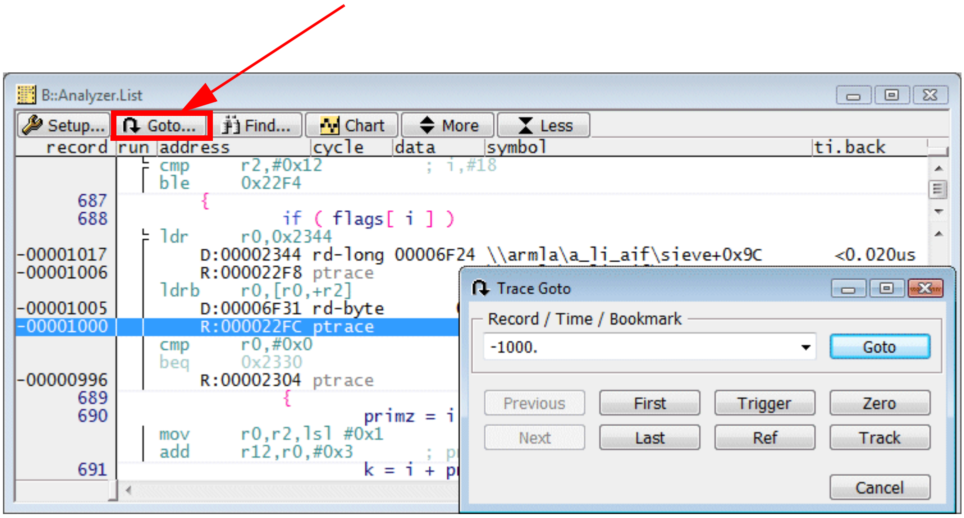
after pushing **'Less'-button** the **1st time**	Suppress the display of the program trace package information (ptrace).
after pushing **'Less'-button** the **2nd time**	Suppress the display of the assembly code.
after pushing **'Less'-button** the **3rd time**	Suppress the data access information (e.g. wr-long cycles).

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



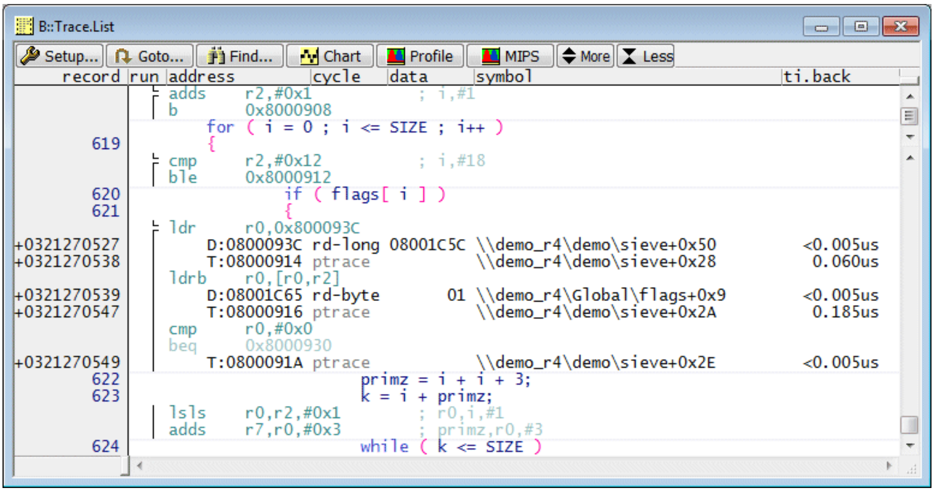
Active Window

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



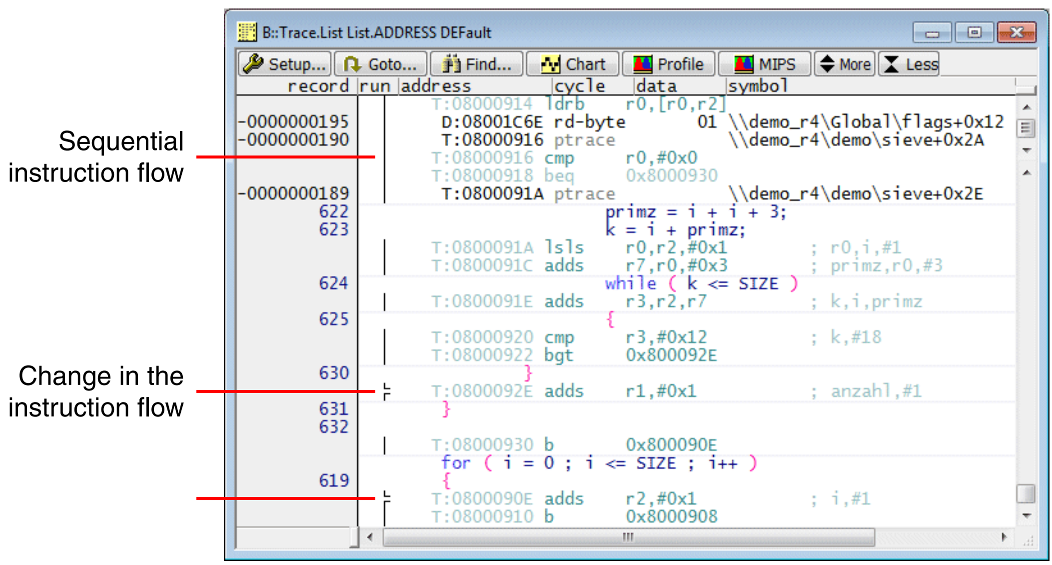
- **Column *record***

  Displays the record numbers

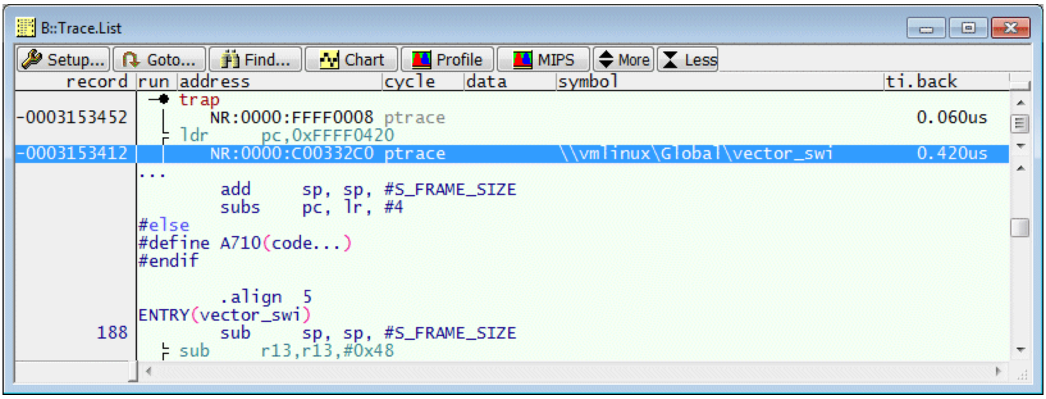
- **Column *run***

  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)

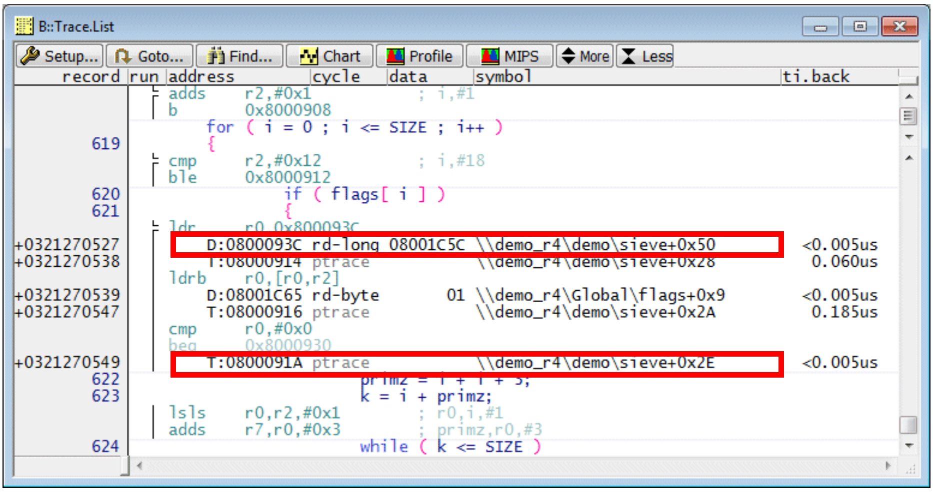
- rd-byte, rd-word, rd-long (read access)

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

- task (Task ID written via Context ID register)

- overlay (Code-Overlay ID written via Context ID register)

- **Column *address/symbol***

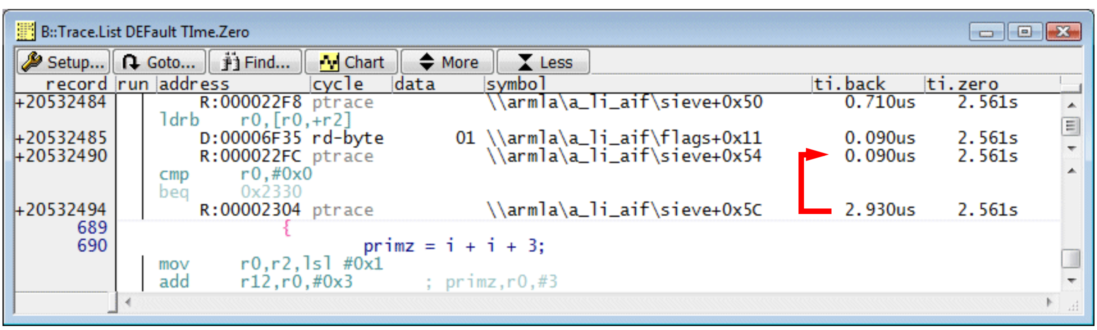


The **address column** shows the following information:
<memory_class>:<logical_address>

Memory Classes	
T	Instruction Thumb mode decoded
R	Instruction ARM mode decoded
D	Data address

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

- **Column *ti.back***

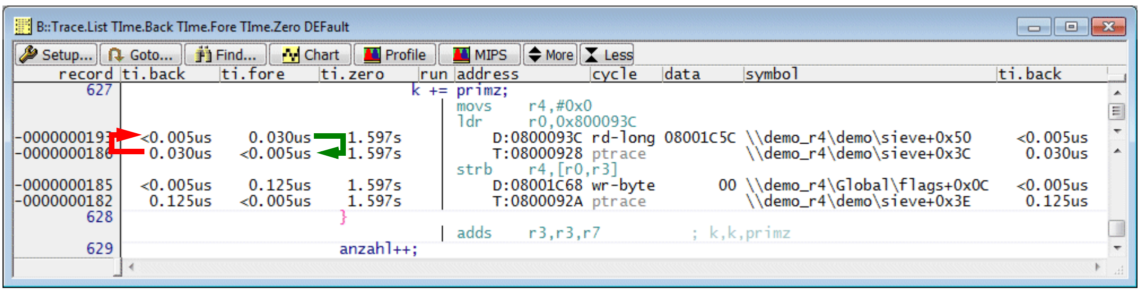


The **ti.back column** shows the time distance to the previous record.

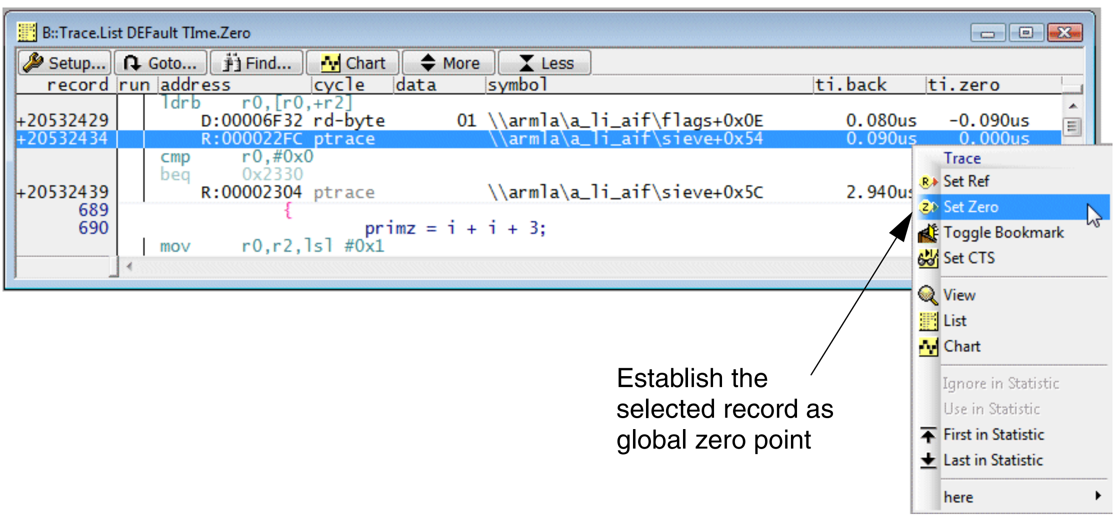
Time Information

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



Time Information for Cycle Accurate Mode

- **Cycle Accurate Mode Pros**

  Provides accurate core clock cycle information.

  Accurate time information can be calculated, if the core clock was constant while recording the trace information.

- **Cycle Accurate Mode Cons**

  Cycle accurate tracing requires up to 4 times more bandwidth.

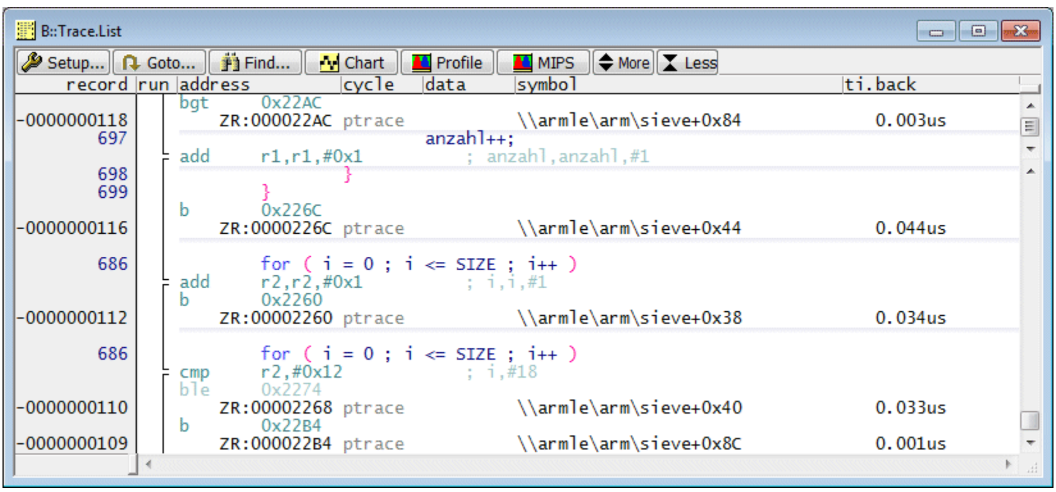
  ETM/PTM trace information can not be correlated with any other trace information.

  Trace information has to be processed always from the start of the trace memory. “Tracking” indicates that the display of the trace information might need some time.

Cycle Accurate Mode and constant core clock while recording

Example for Cortex-A9:

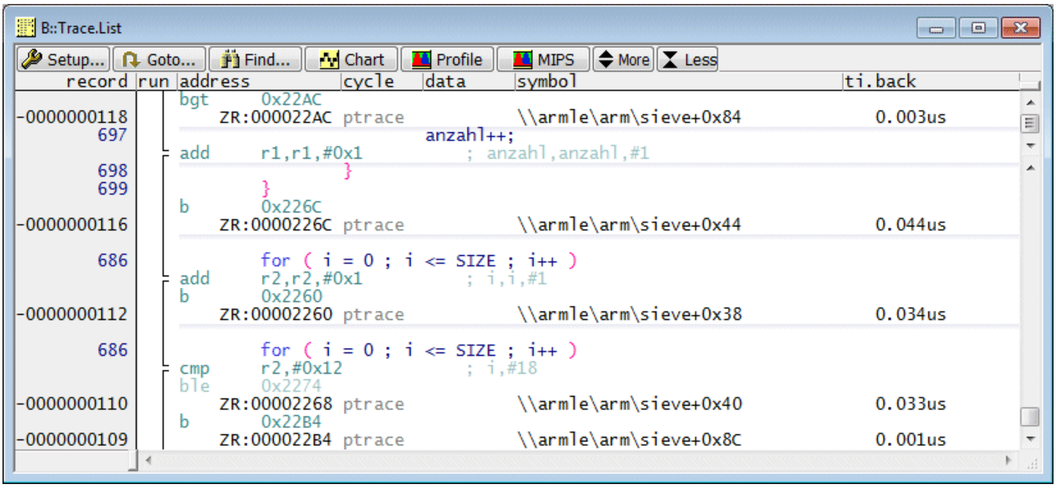
```
ETM.TImeMode CycleAccurate              ; enable cycle accurate mode

Trace.CLOCK 800.MHZ                     ; inform TRACE32 about your core
                                        ; clock frequency
```



Cycle Accurate Mode and changing core clock while recording

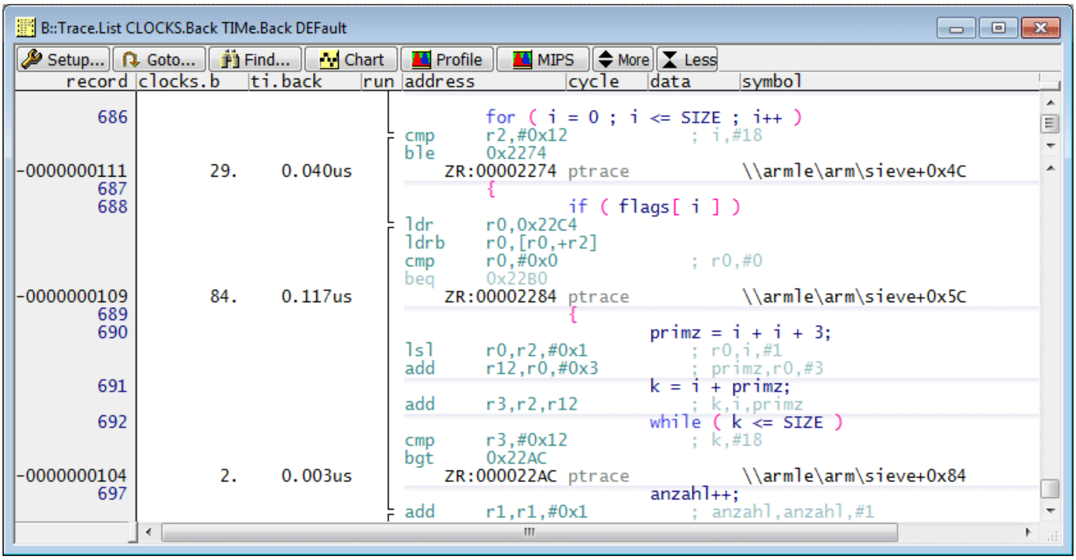
Example for Cortex-A9:

```
; combines cycle accurate mode with TRACE32 global timestamp
ETM.TImeMode CycleAccurate+ExternalTrack
```

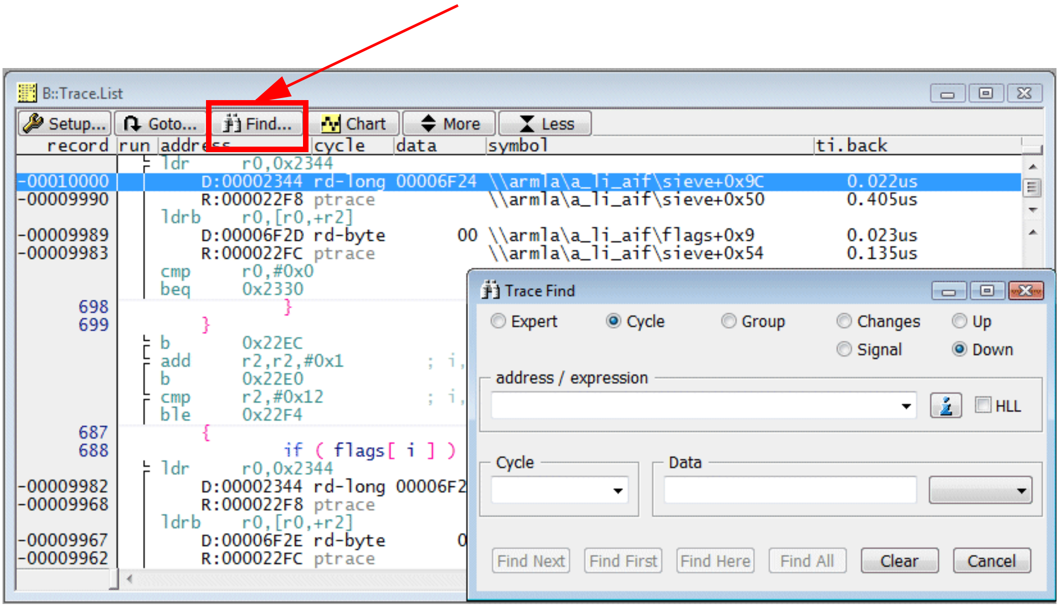


In addition to the timing information the number of clocks needed by an instruction/instructions range can be displayed.

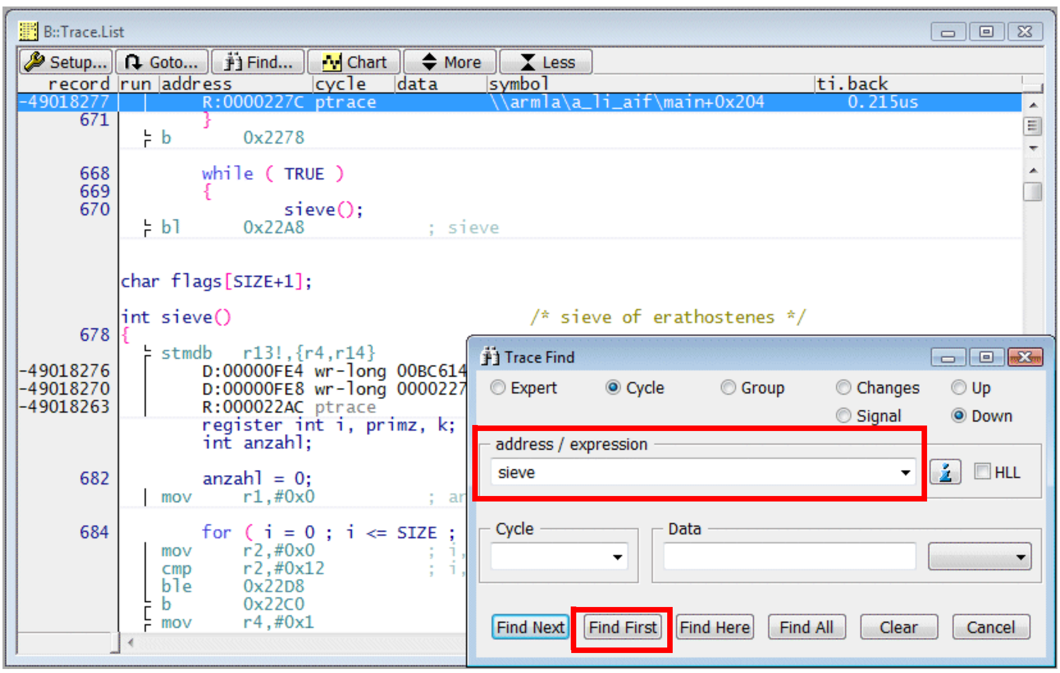
```
Trace.List CLOCKS.Back TIme.Back DEFault
```



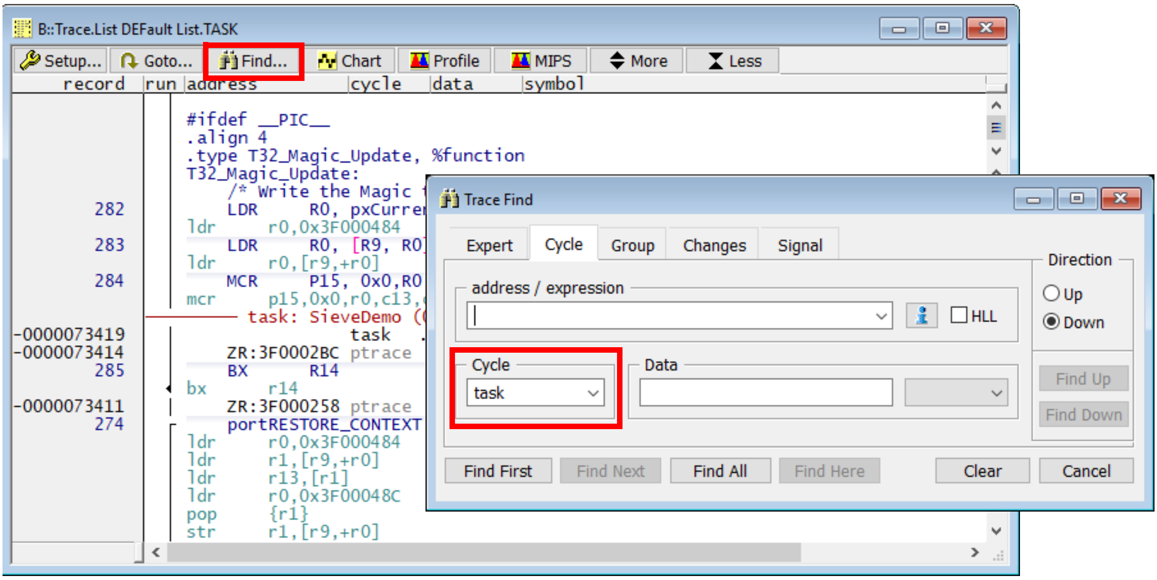
Find a Specific Record



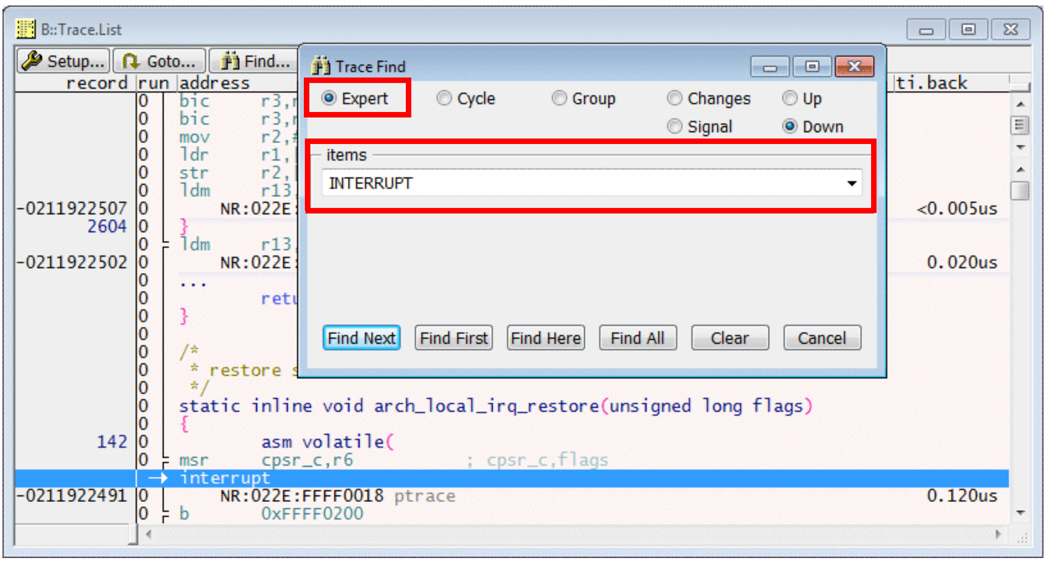
Example: Find a specific symbol address.



Example: Find Context ID



Example: Find Interrupt/Trap



Belated Trace Analysis

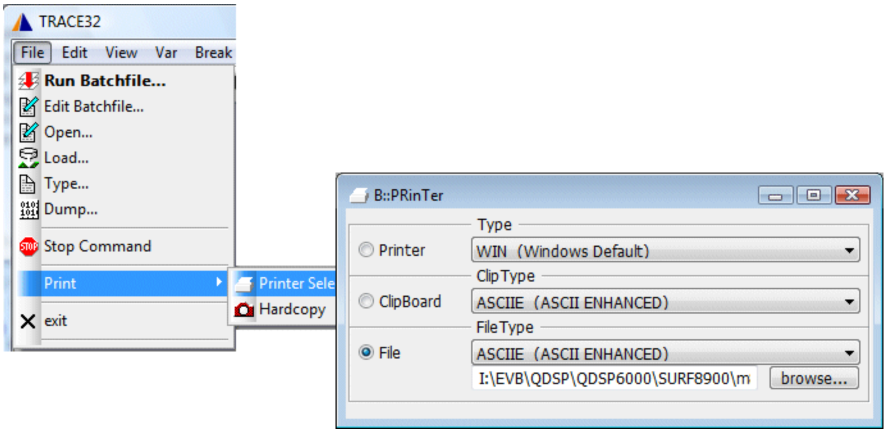
There are several ways for a belated trace analysis:

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

Save the Trace Information to an ASCII File

Saving 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 testrun1.lst               ; specify the file name
```

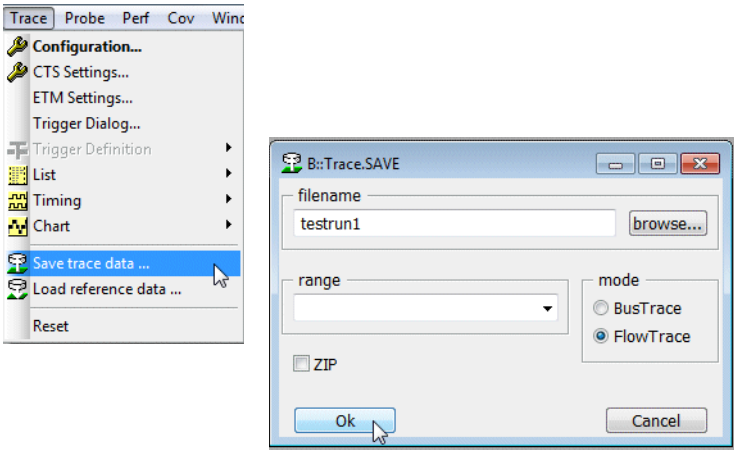
2. It only makes sense to save a part of the trace contents into an ASCII-file. Use the record numbers to specify the trace part you are interested in.

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

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

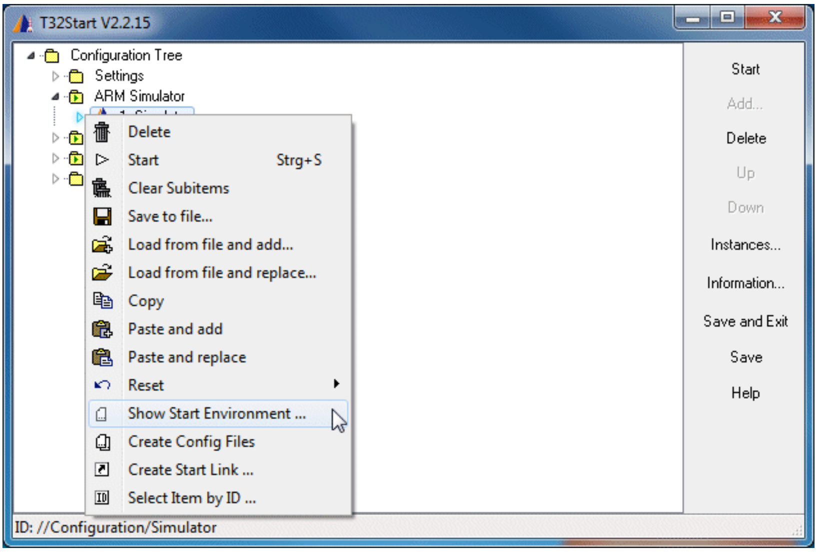
3. Use an ASCII editor to display the result.

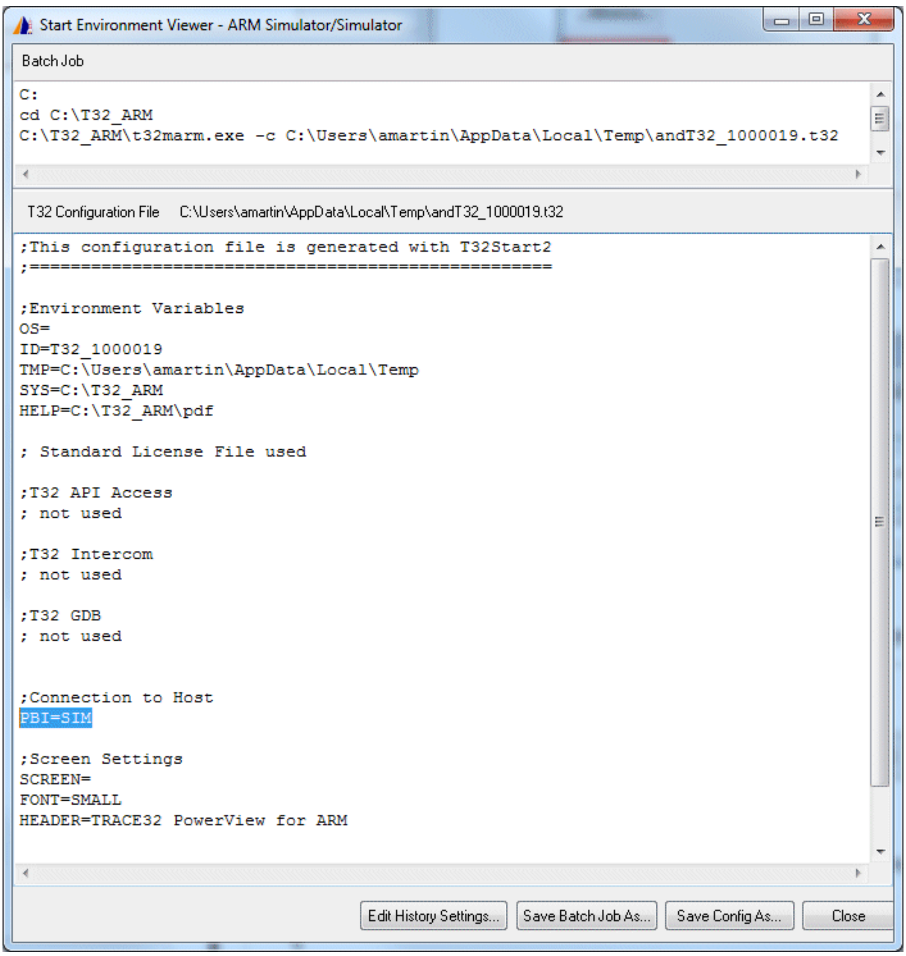
1. **Save the contents of the trace memory into a file.**



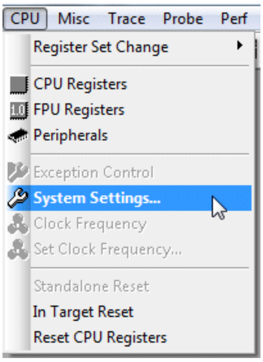
The default extension for the trace file is **.ad**.

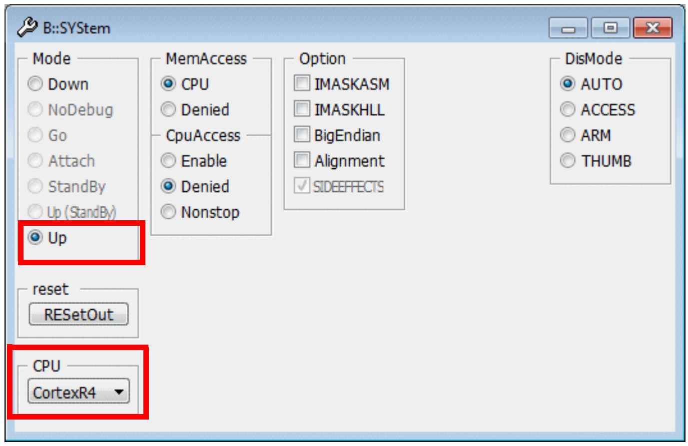
2. Start a TRACE32 Instruction Set Simulator (PBI=SIM).



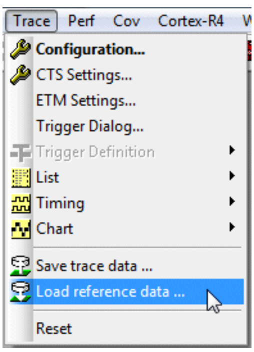


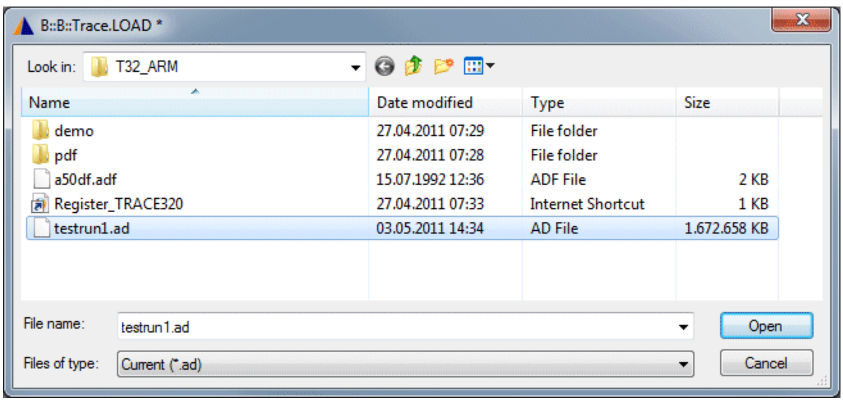
3. **Select your target CPU within the simulator. Then establish the communication between TRACE32 and the simulator.**



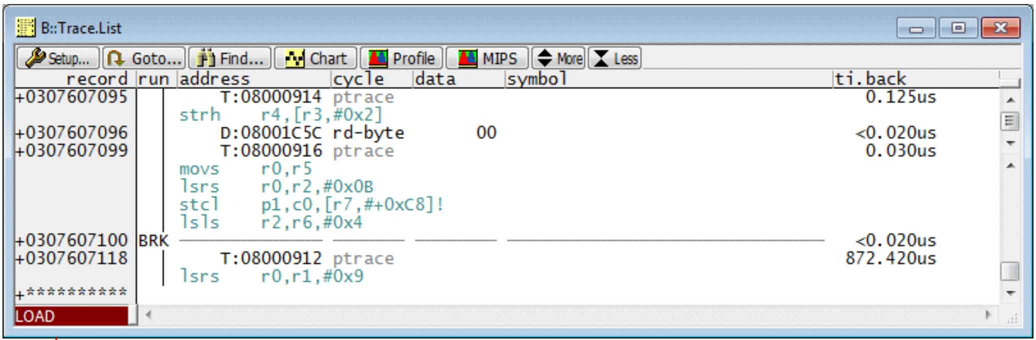


4. **Load the trace file.**





5. **Display the trace contents.**



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

6. **Load symbol and debug information as you need it.**

```
Data.LOAD.Elf demo_r4.axf /NoCODE
```

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

Export the Trace Information as ETM Byte Stream

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

```
Analyzer.EXPORT testrun1.ad /ByteStream

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

In the past it was necessary to spend a lot of time analyzing the trace listing in order to find out which instructions, data or system states had caused malfunctioning of the target system.

Now Trace-based Debugging - also CTS for Context Tracking System - allows the user to recreate the state of the target system at a selected point based on the information sampled in the trace buffer. From this starting point the program steps previously recorded in real-time in the trace memory can be re-debugged again in the TRACE32 PowerView GUI.

Standard Trace-based Debugging requires:

- Continuous instruction flow trace (**Trace.Mode Fifo** or **Leash**)
- Continuous data flow trace (at least read accesses).

  If the read data and the operation on the read data are known, the write accesses can be recreated by CTS.

The ARM Cortex core that allow standard Trace-based Debugging are highlighted by a grey background.

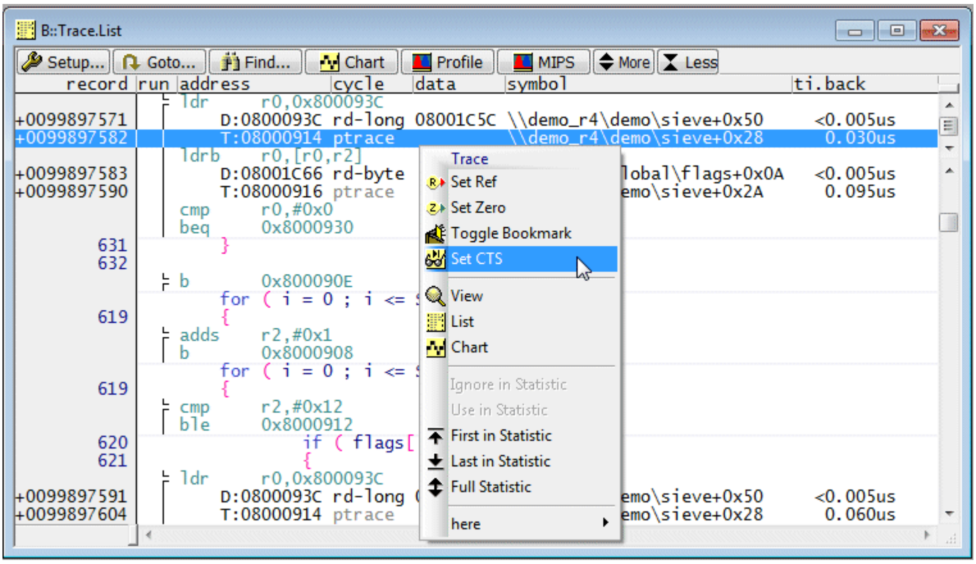
Core	ETM Version	Program	Data Address	Data Value	Context ID	CC
ARM7 **ARM9**	ETMv1	■	■	■	≥ ETMv1.2	■
ARM9	CoreSight ETMv3	■	■	■	■	■
ARM11	(CoreSight) ETMv3	■	■	■	■	■
Cortex-M3 / M4 **Cortex-M23**	CoreSight ETMv3	■	(DWT)	(DWT)		
Cortex-R4 **Cortex-R5**	CoreSight ETMv3	■	■	■	■	■
Cortex-R7 / R8 **Cortex-R52**	CoreSight ETMv4	■	■	■	■	■
Cortex-A5 **Cortex-A7**	CoreSight ETMv3	■	■	■	■	■
Cortex-A8	CoreSight ETMv3	■	■		■	■
Cortex-A9 **Cortex-A15** **Cortex-A17**	CoreSight PTM	■			■	■

Forward and Backward Debugging

Trace-based Debugging allows to re-debug a trace program section.

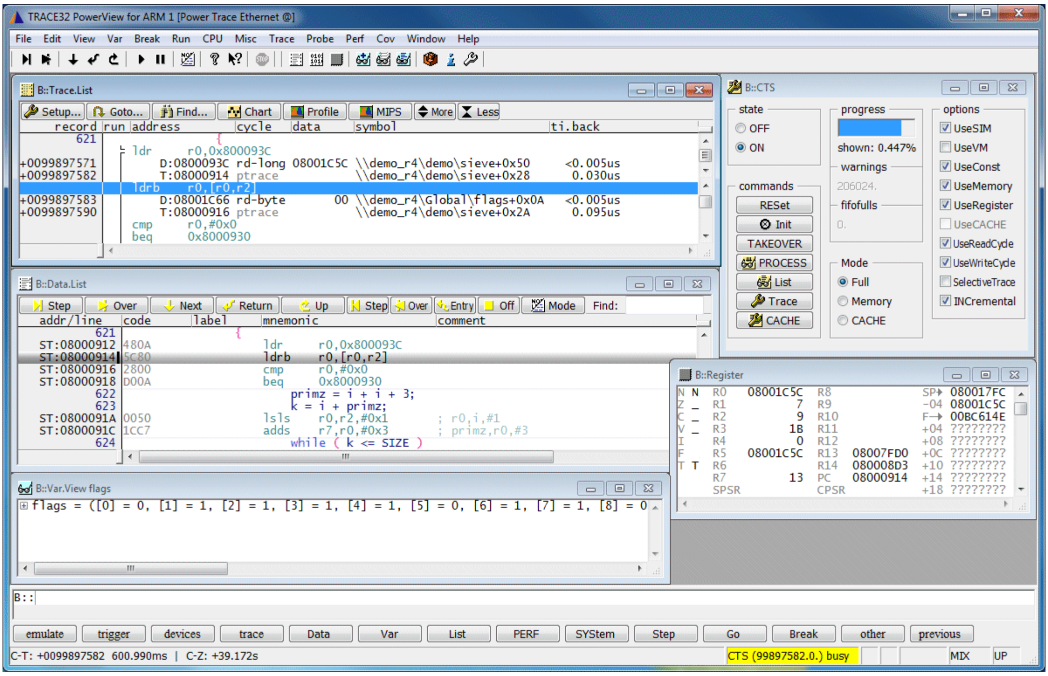
Trace-based Debugging is set up as follows:

1. Select the trace record that should be the starting point for Trace-based Debugging and select **Set CTS** in the **Trace** context menu.

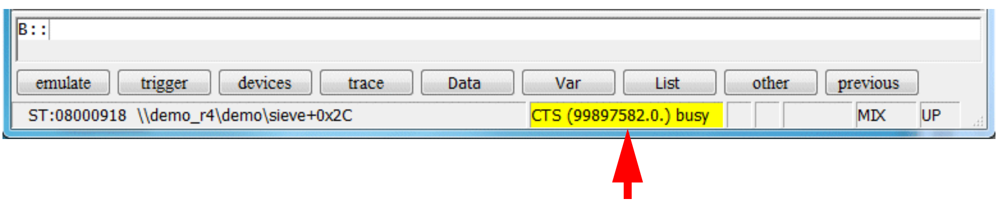


2. TRACE32 PowerView now recreates the state of the target system as it was when the instruction at the starting point was executed. The recreation take a while (CTS busy).

   Please be aware, that CTS recreates the former target state only in the TRACE32 PowerView GUI. This has no effect on the target system.



When CTS is active the TRACE32 PowerView GUI does not show the current state of the target system. It shows a state recreated by the TRACE32 software for the record displayed in the state line.

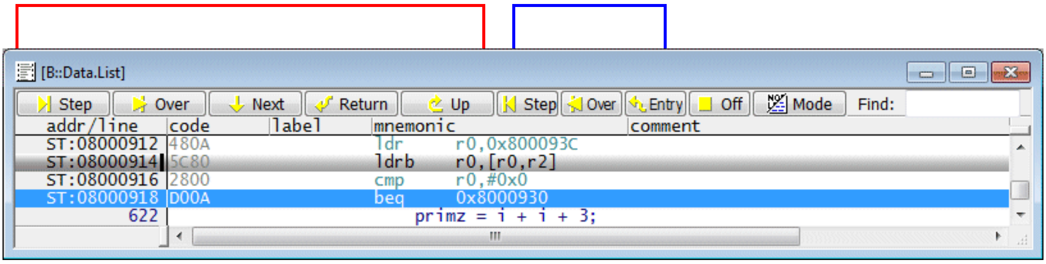


In order to avoid irritations the look-and-feel of TRACE32 PowerView is changed to yellow when CTS is active.

The main subjects of the CTS recreation are:

- Source listing
- Register contents
- Memory contents
- Call stack and local variables
- Variables

Forward debugging commands Backward debugging commands



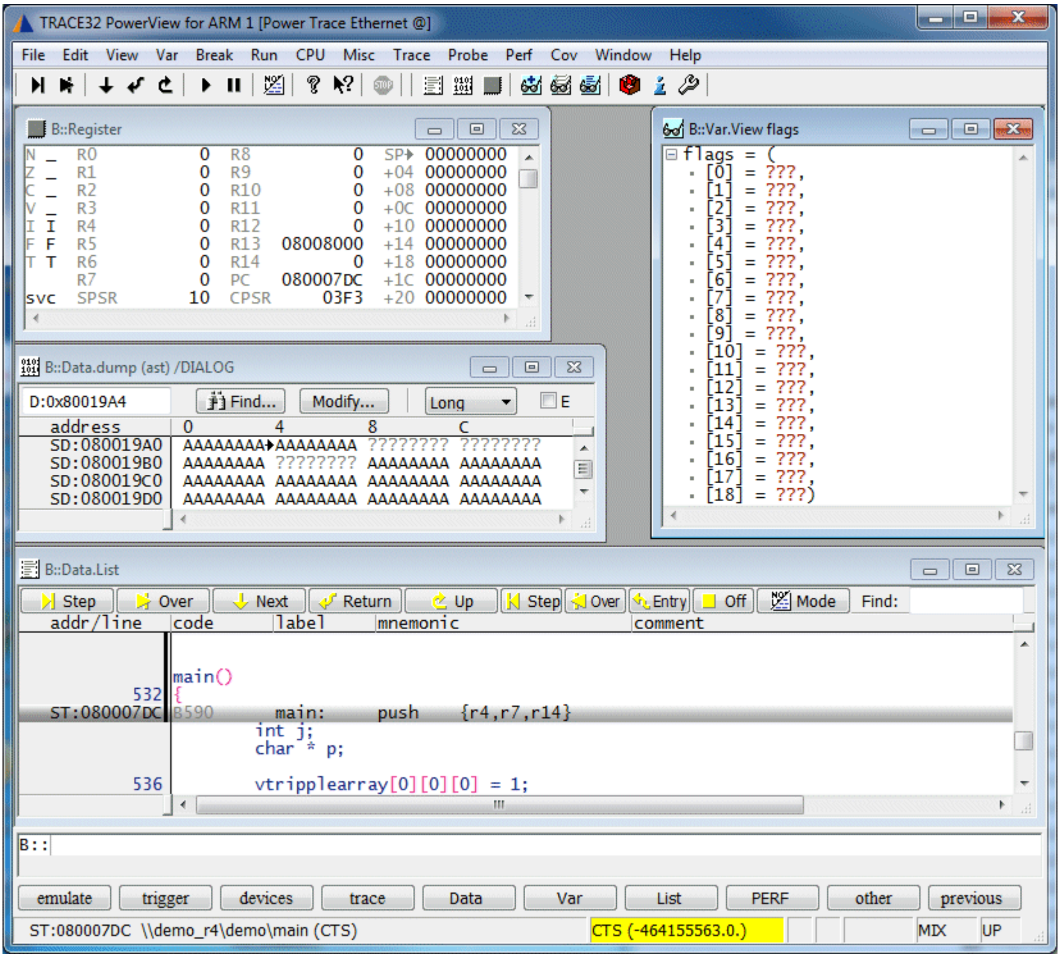
The following **forward debugging** commands can be used to re-debug the traced program section:

- Single step (**Step.single**)
- Step over call (**Step.Over**)
- Single step until specified expression changes (**Step.Change**, **Var.Step.Change**)
- Single step until specified expression becomes true (**Step.Till**, **Var.Step.Till**)
- **Go**
- Start the program execution. Stop it when the next written instruction is executed (**Go.Next**)
- Start the program execution and stop it before the last instruction of the current function is executed (**Go.Return**)
- Start the program execution and stop it after it returned to the calling function (**Go.Up**)

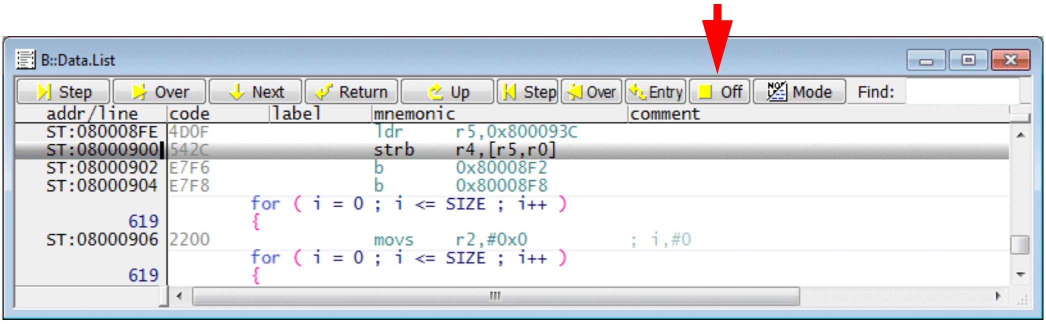
The following **backward debugging** commands can be used to re-debug the traced program section:

- Step backwards (**Step.Back**)
- Step backwards over call (**Step.BackOver**)
- Step backwards until specified expression changes (**Step.BackChange**, **Var.Step.BackChange**)
- Step backwards until specified expression becomes true (**Step.BackTill**, **Var.Step.BackTill**)
- Run the program backwards (**Go.Back**)
- Run the program until the start of the current function (**Go.BackEntry**)

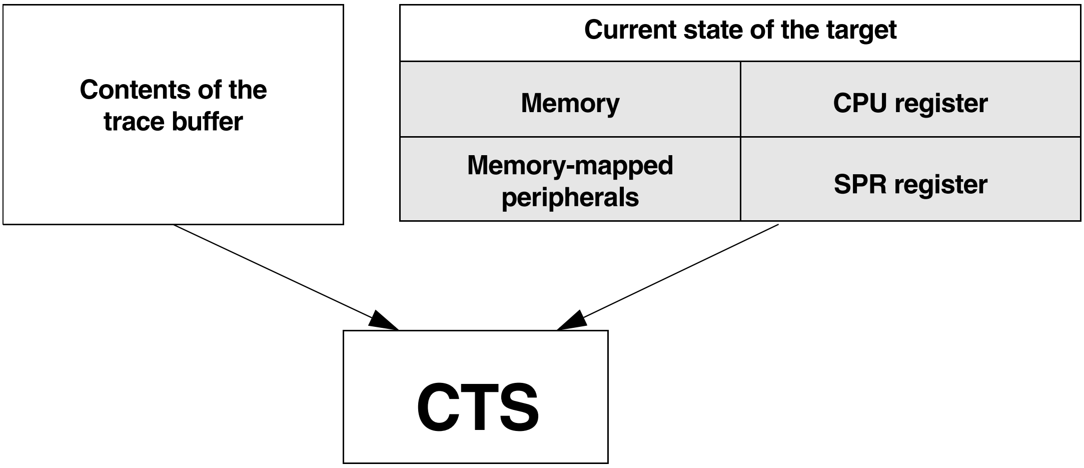
If CTS can not recreate the contents of a memory location or a variable value a ? is displayed.



If you want to terminate the re-debugging of a traced program section use the yellow **Off** button (**CTS.OFF**) in the **Data.List** window.



After CTS is switched off the TRACE32 PowerView GUI displays the current contents of your target system.



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

1. CTS can only perform a correct recreation, if solely the core, for which the data trace information is recorded into the trace buffer, writes to memory. If there are memory addresses, e.g. dual-ported memories or peripherals, that are change otherwise, these addresses have to be excluded by the command **MAP.VOLATILE** from the CTS recreation. These memory addresses are then displayed as unknown (?) if CTS is used.

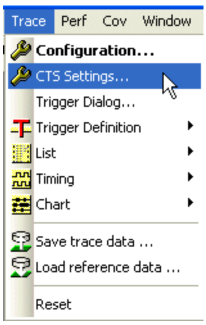
2. CTS performs memory reads while performing a recreation. If read accesses to specific memory-mapped peripherals should be prevented, the addresses have to be excluded by the command **MAP.VOLATILE** from the CTS recreation. These memory addresses are then displayed as unknown (?) if CTS is used.

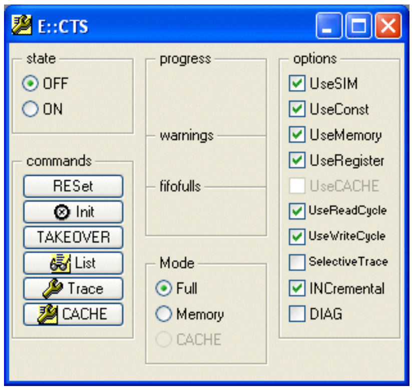
3. Under certain circumstances the reconstruction of the instruction flow can cause BUSERRORS on the target system. If this is the case, it is recommended to load the code to TRACE32 Virtual Memory.

4. CTS has to be re-configured if:

- the program execution is still running while CTS is used.
- not all CPU cycles until the stop of the program execution are sampled to the trace.
- the trace contents is reprocessed with a TRACE32 Instruction Set Simulator.
- only the program flow is sampled to the trace buffer.

In all these cases the current state of the target can not be used by CTS. For more information refer to the command **CTS.state**.





MAP.VOLATILE *<range>*	Exclude addresses from CTS
CTS.state	Reconfigure CTS.

The TRACE32 Instruction Set Simulator can be used for a belated Trace-based Debugging. To set up the TRACE32 Instruction Set Simulator for belated Trace-based Debugging proceed as follows:

1. Save the trace information to a file

```
Trace.SAVE my_file
```

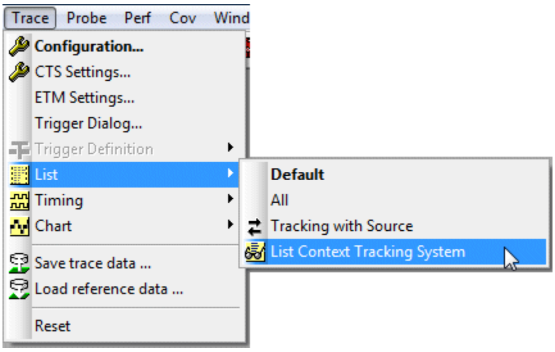
2. Set up the TRACE32 Instruction Set Simulator for a belated Trace-based Debugging:

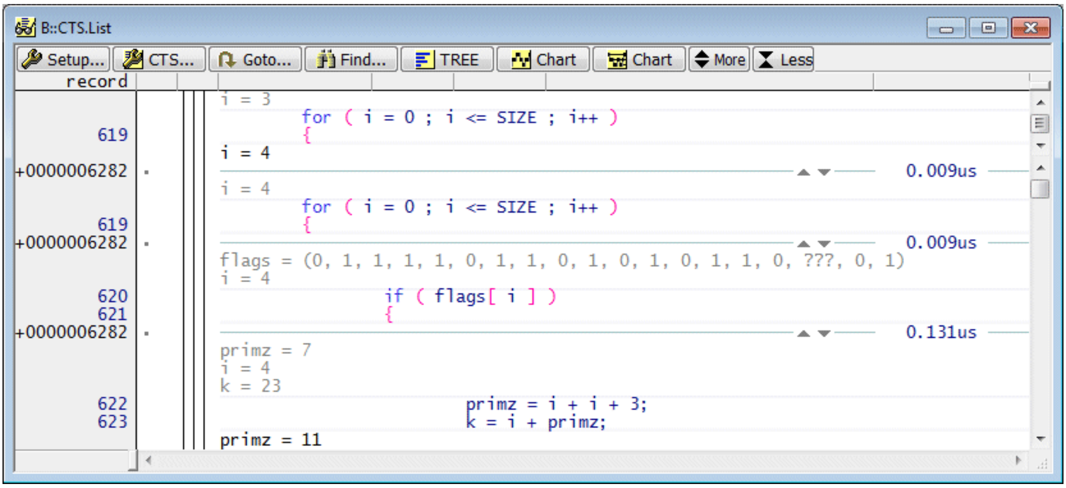
```
SYStem.CPU CORTEXR4                   ; select the target CPU

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

Trace.LOAD my_trace.ad                ; load the trace file

Data.LOAD.Elf demo_r4.axf /NoCODE     ; load the symbol and debug
                                      ; information

Trace.List                            ; display the trace listing

CTS.UseMemory OFF                     ; exclude the current memory
                                      ; contents from CTS

MAP.CONST 0x8000900++0xff             ; inform TRACE32 which memory
                                      ; address range provides
                                      ; constants

; CTS.UseConst ON                     ; include constant address
                                      ; range into CTS

CTS.UseRegister OFF                   ; exclude the current register
                                      ; contents from CTS

CTS.GOTO -293648539.                  ; specify the CTS starting
                                      ; point

…                                     ; start the re-debugging
```

HLL Analysis of the Trace Contents

CTS provides also a number of features for high-level language trace display.

Details on each HLL Instruction

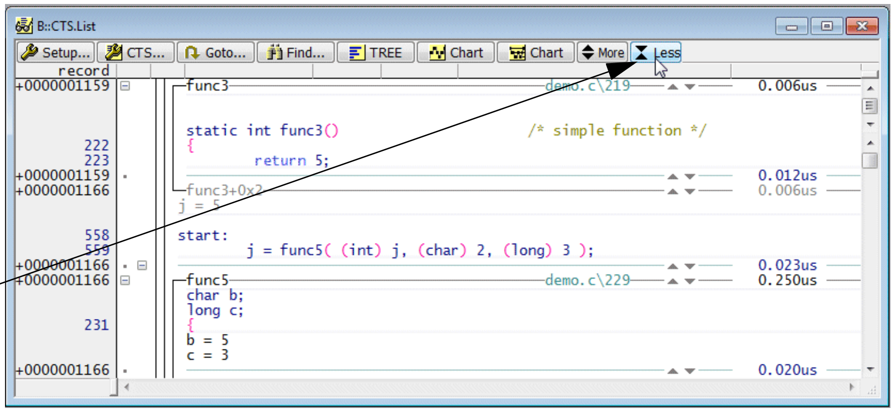




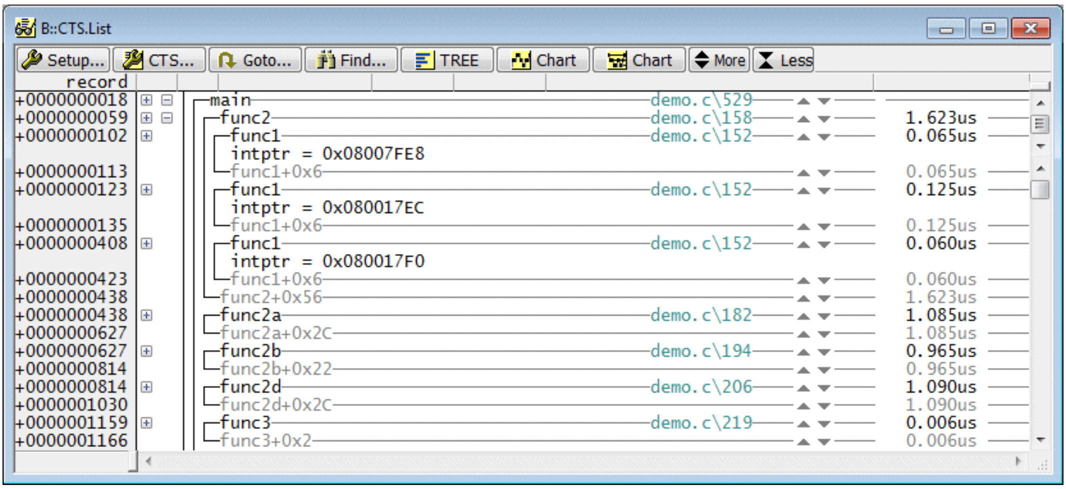
For each HLL step the following information is displayed:

- 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 [<*record*|*range*>] [<*item*> …] [/<*option*>] List pure HLL trace.



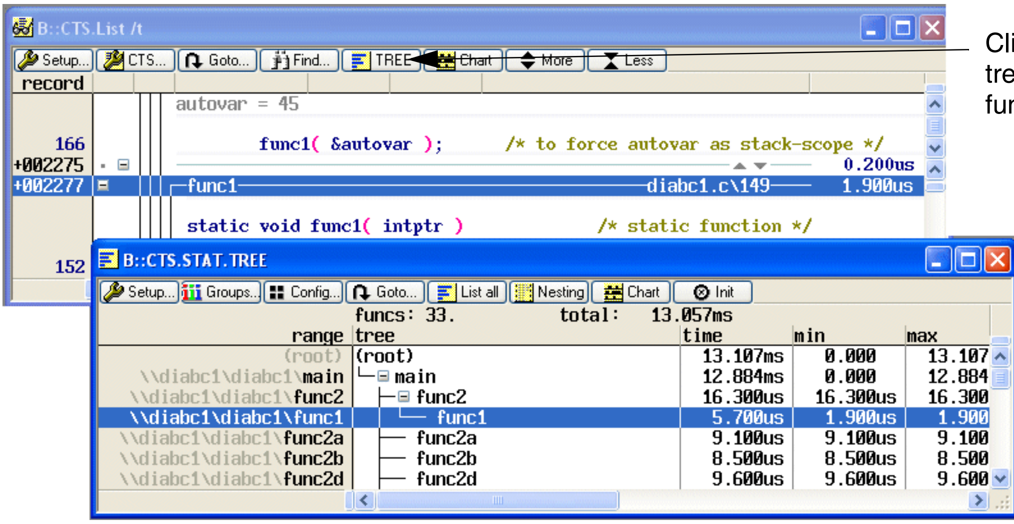
Push the ***Less*** button to get a function nesting analysis



For each function you get the following information:

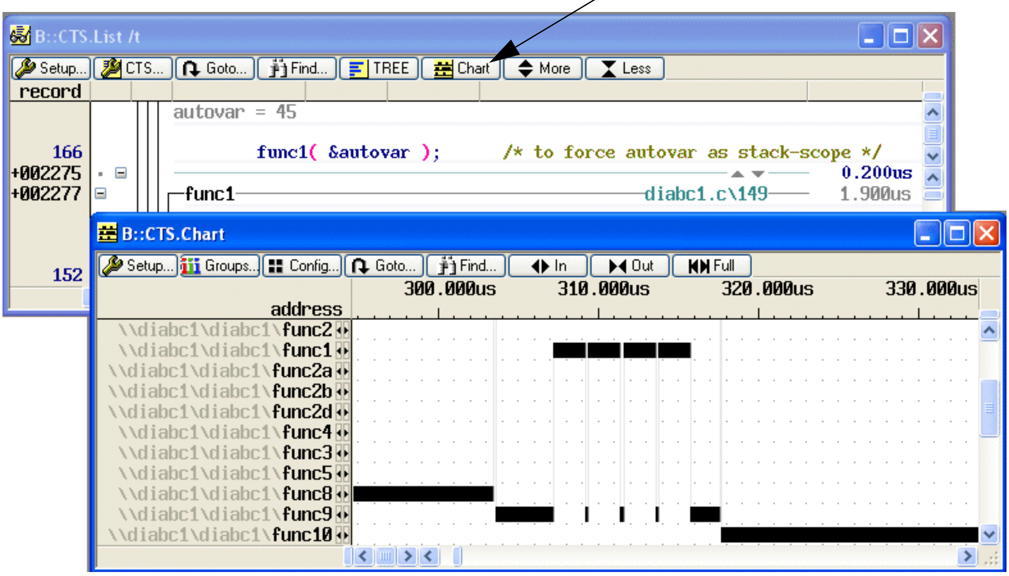
- Function parameters (and return value - not available on all CPUs)
- Time spent in the function

Click here to get a tree analysis of the function nesting



Timing Display

Click here to get a graphical analysis of the function runtime



CTS.STATistic.TREE [**%***<format>*][*<item>* …] [**/***<options>*]	Display trace contents as call tree
CTS.Chart.sYmbol [*<record_range>*] [*<scale>*] [**/***<option>*]	Display trace contents as timing based on the symbol information

Context

An ETM can contain logic (resources) to control the tracing.

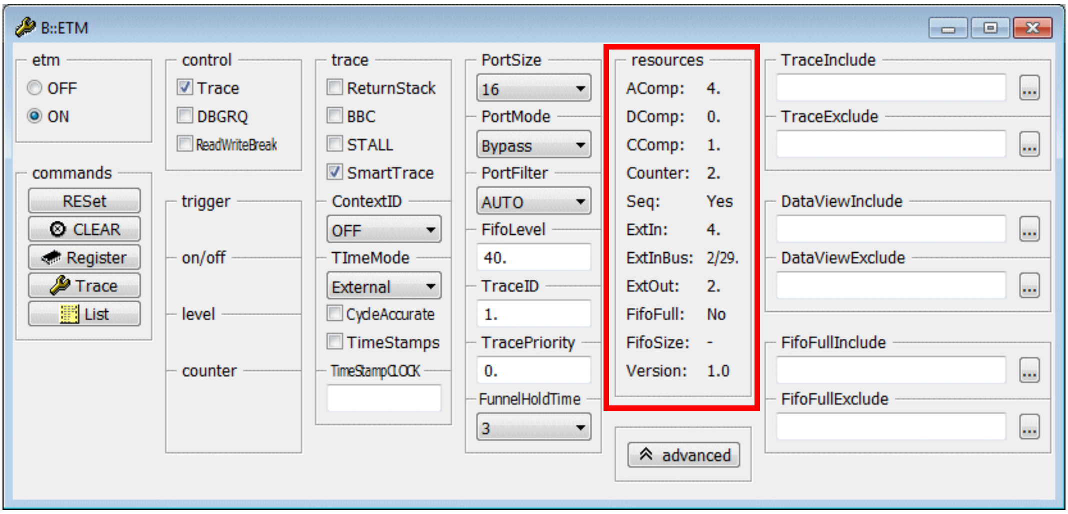
- **Filter:** Address Comparator, Data Comparator and Context ID Comparators are provided to advice the ETM to only generate trace information for events of interest.
- **Trigger:** Address Comparator, Data Comparator, Context ID Comparators, Counters and a Three-level Sequencer can be used to send a Trigger to the trace recording. In most cases the trace recording is stopped when a Trigger occurs.

The table below lists which resources are provided by the various ARM Cortex cores:

Core	ETM Version	Address Comparators	Data Comparators	Context ID Comparator	Counters	Sequencer
ARM7 **ARM9**	ETMv1	■	■	■	■	■
ARM9	CoreSight ETMv3	■	■	■	■	■
ARM11	(CoreSight) ETMv3	■	■	■	■	■
Cortex-M3 **Cortex-M4** **Cortex-M23**	CoreSight ETMv3	(DWT)	(DWT)			
Cortex-R4 **Cortex-R5**	CoreSight ETMv3	■	■	■	■	■
Cortex-R7/R8 **Cortex-R52**	CoreSight ETMv4	■	■	■	■	■
Cortex-A5 **Cortex-A7**	CoreSight ETMv3	■	■	■	■	■
Cortex-A8	CoreSight ETMv3	■	■	■	■	■

Core	ETM Version	Address Comparators	Data Comparators	Context ID Comparator	Counters	Sequencer
Cortex-A9 **Cortex-A15** **Cortex-A17**	CoreSight PTM	■		■	■	■
Cortex-A3x **Cortex-A5x** **Cortex-A7x**	CoreSight ETMv4	■		■	■	■

The ETM Configuration Window provides detailed information on the available resources for your core:

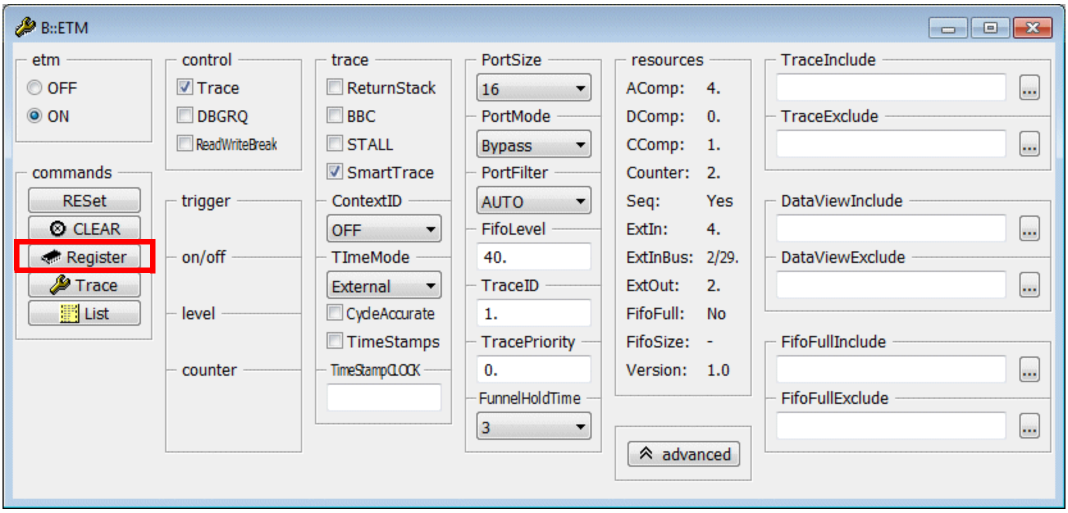


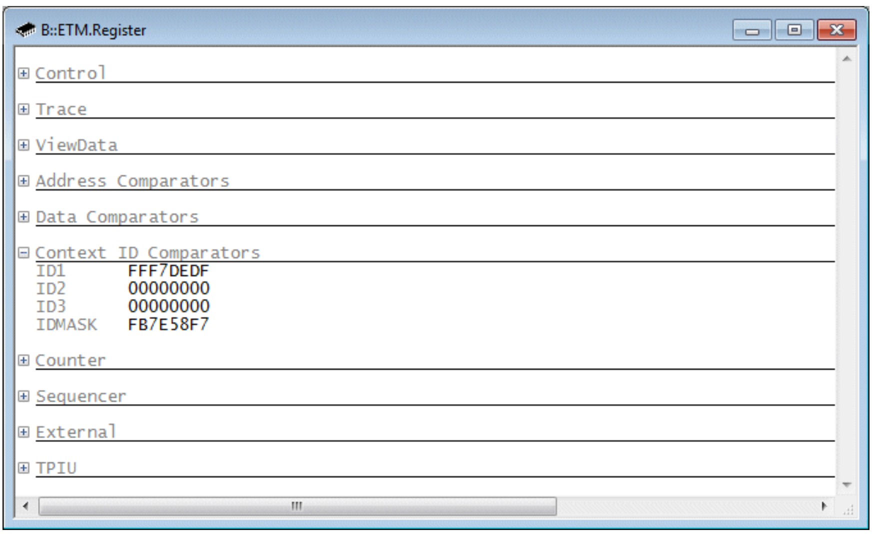
Number of address range comparators	AComp
Number of data comparators (single value or bitmask)	DComp
Number of Context ID comparator	CComp
Number of 16-bit counters	Counter
Three-state sequencer implemented	Seq: Yes

Special values for “DComp”:

- A 0 zero means, you can compare addresses on read/write but you cannot compare data.
- A minus (‘-’) means, the ETM (or PTM) cannot compare addresses for read/write accesses. (The address comparators can only consider program addresses.)

If you push the **Register** button in the ETM Configuration Window, you get a tree display of the control registers for the ETM.

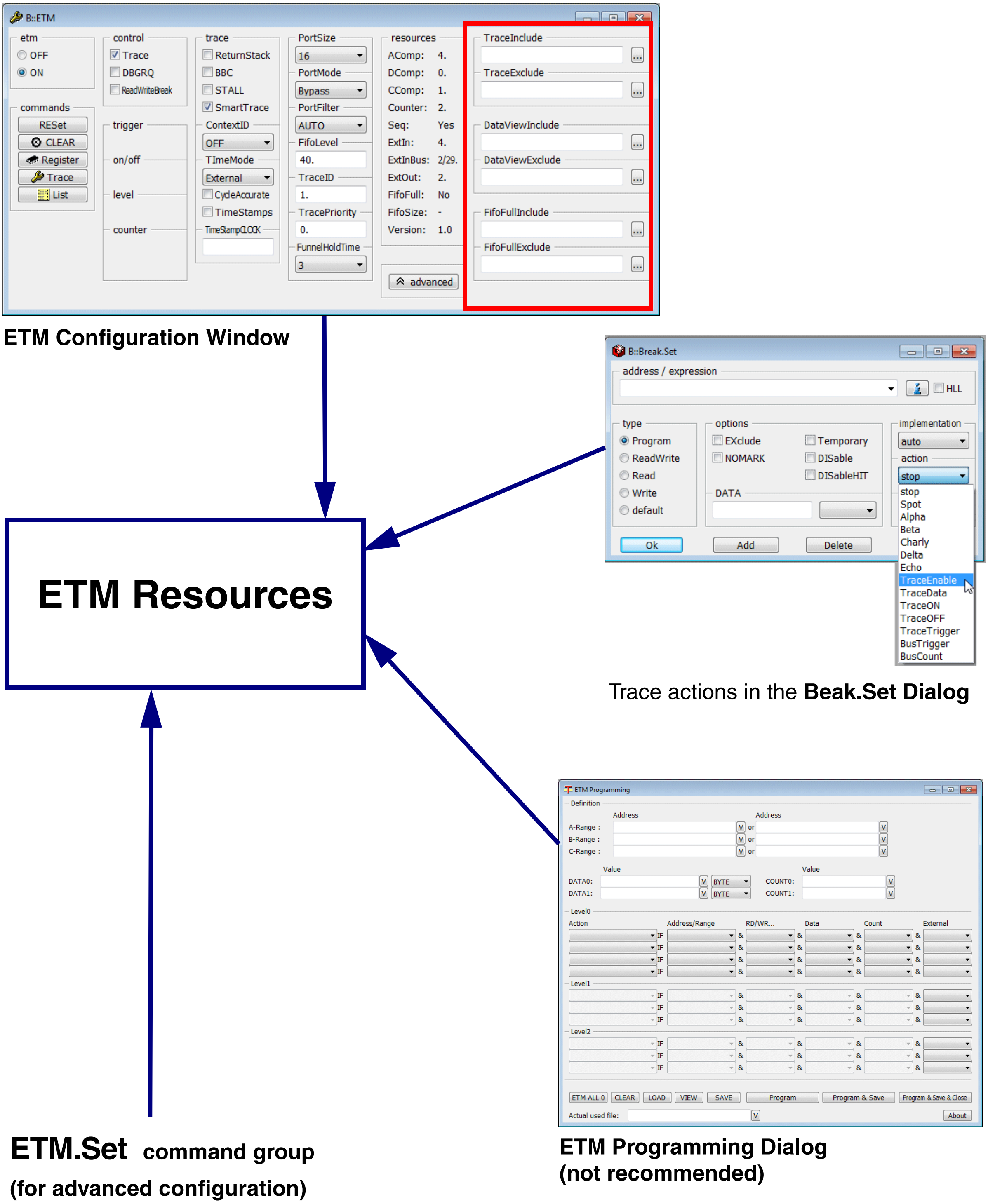




Use the following command, if you want to read the ETM registers while the program execution is running:

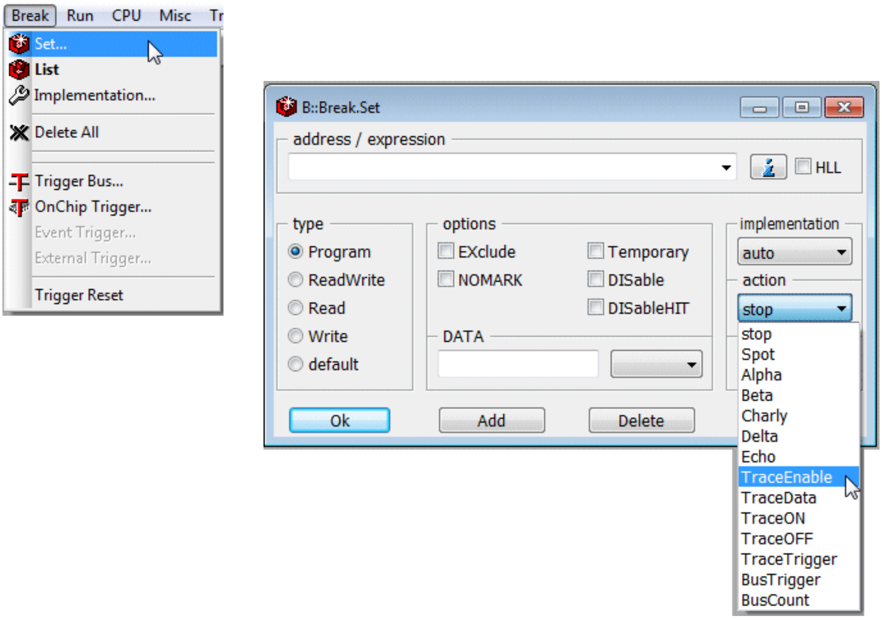
```
ETM.Register , /DualPort
```

The following TRACE32 components can be used to control the ETM resources.



For advanced configurations via command **ETM.Set** see **"ARM-ETM Trace"** (trace_arm_etm.pdf)

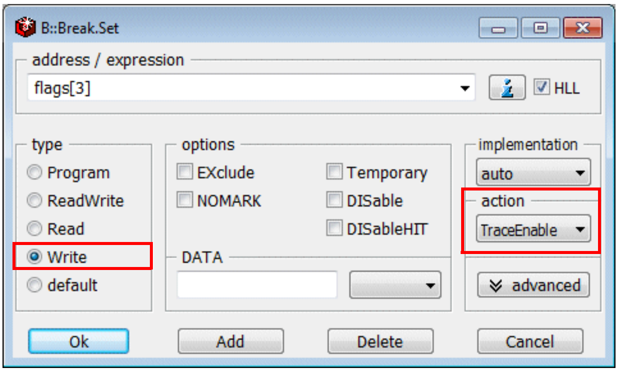
Filters and Trigger by Using the Break.Set Dialog



TraceEnable	Advise the ETM to generate trace information only for the specified data or program event(s).
TraceData	Advice the ETM to generate trace information for the complete instruction flow and additionally for the specified data event.
TraceON	Advise the ETM to start with the generation of trace information at the specified event.
TraceOFF	Advise the ETM to stop the generation of trace information at the specified event.
TraceTrigger	Advise the ETM to generate a Trigger for the trace recording at the specified event. If the ETM Trigger is not already connected to the TPIU in your CoreSight system, this connection has to be configured manually by the corresponding CTI (Cross Trigger Interface) setup.
BusTrigger	Advise the ETM to generate a pulse at ETM External Output 1 at the specified event.
BusCount	Advise the ETM to decrement an ETM counter at the specified event.

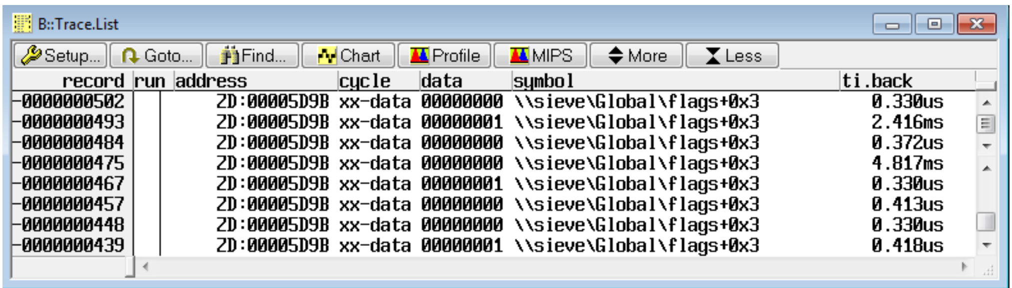
Example 1: Advise the ETM to generate only trace information for the write accesses to the variable flags[3].

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

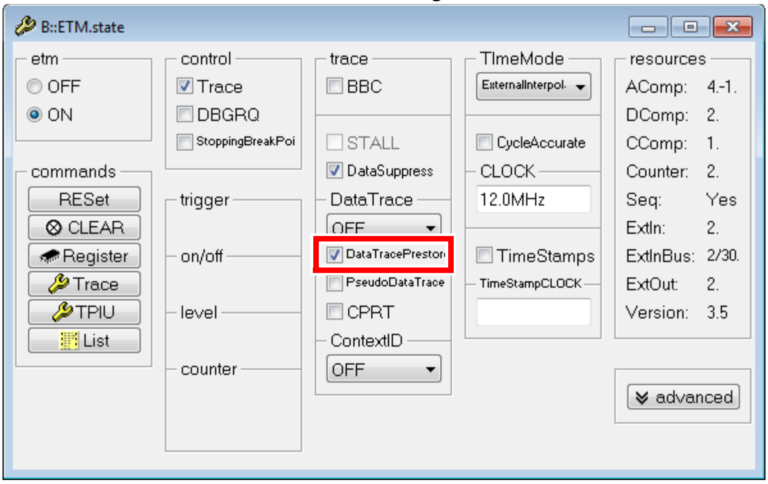


2. Start the program execution and stop it.

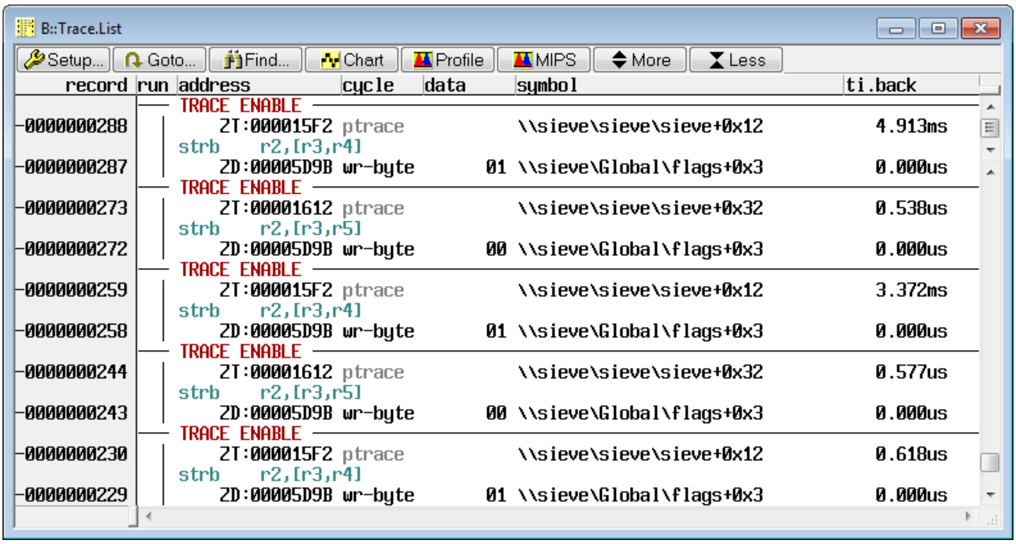
3. Display the result.



4. If you'd like to have details about the instruction that performed the write access, enable DataTracePrestore in the ETM Configuration Window.



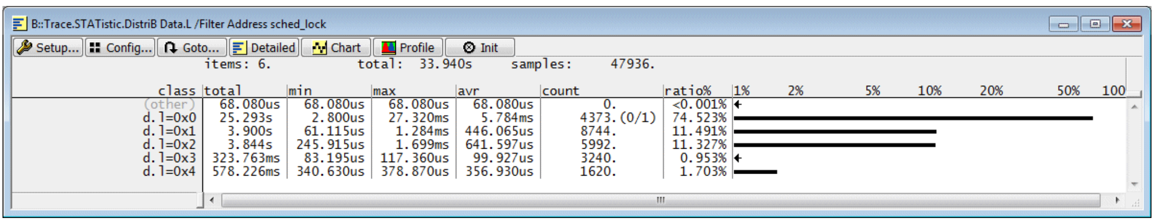
5. With **ETM.DataTracePrestore ON** you see in Trace.List for every data-cycle also the associated program trace cycle (ptrace):



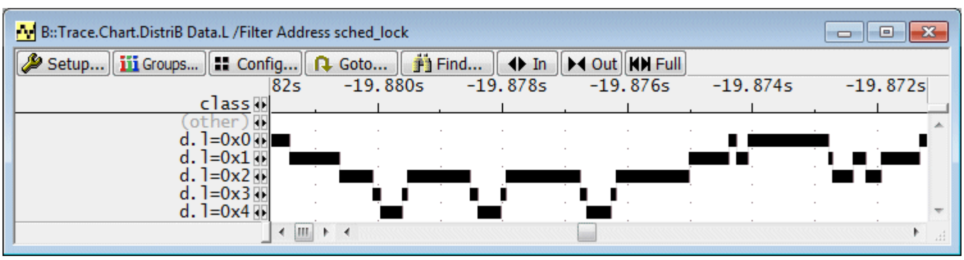
Using DataTracePrestore generates additional trace data with ETMv3.
With ETMv1 and ETMv4 any data trace cycle is always generated together with a program trace cylce. Thus for ETMv1/v4 DataTracePrestore just controls if the debugger *displays* the program trace cylce.

Example 2: Advise the ETM to generate only trace information for the write accesses to the variable sched_lock. Perform various statistical analysis on the trace contents.

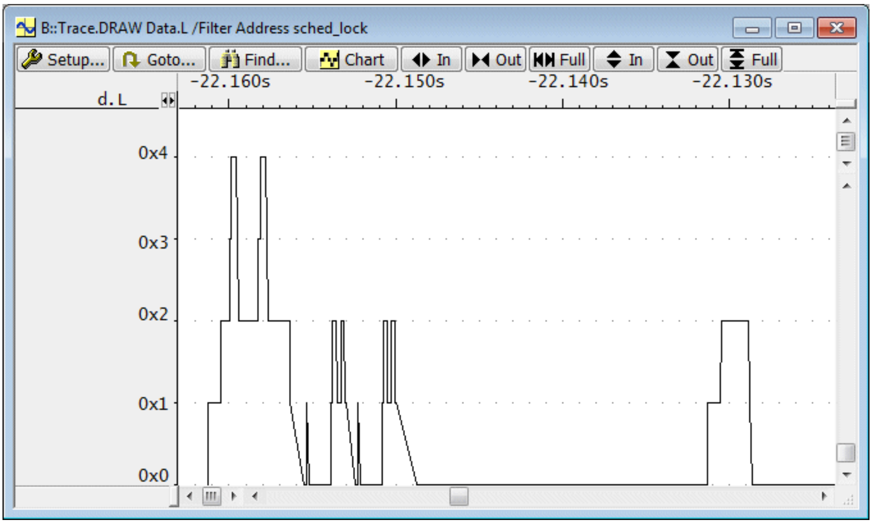
```
; advise the ETM to only generate trace information for the write
; accesses to the variable sched_lock
Var.Break.Set sched_lock /Write /TraceEnable

; start and stop the trace recording to fill the trace memory
Go

…

Break

; display the trace memory contents
Trace.List

; analyse the contents of the variable sched_lock statistically
Trace.STATistic.DistriB Data.L /Filter Address sched_lock
```




```
; display a timing diagram that illustrates the contents changes of
; the variable sched_lock
Trace.Chart.DistriB Data.L /Filter Address sched_lock
```

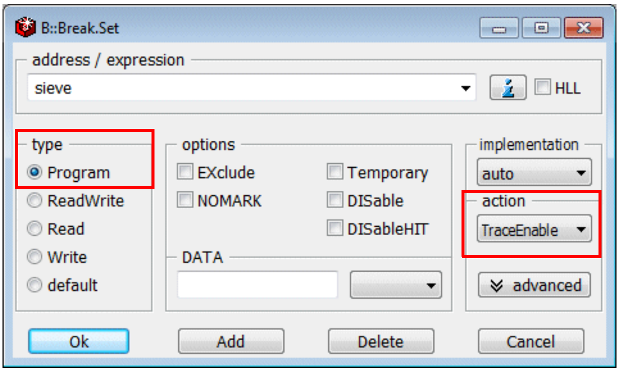


```
// display a graph over time of the variable "sched_lock"

Trace.DRAW.Var %DEFault sched_lock

// or

Trace.DRAW.channel Data.L /Filter Address sched_lock
```



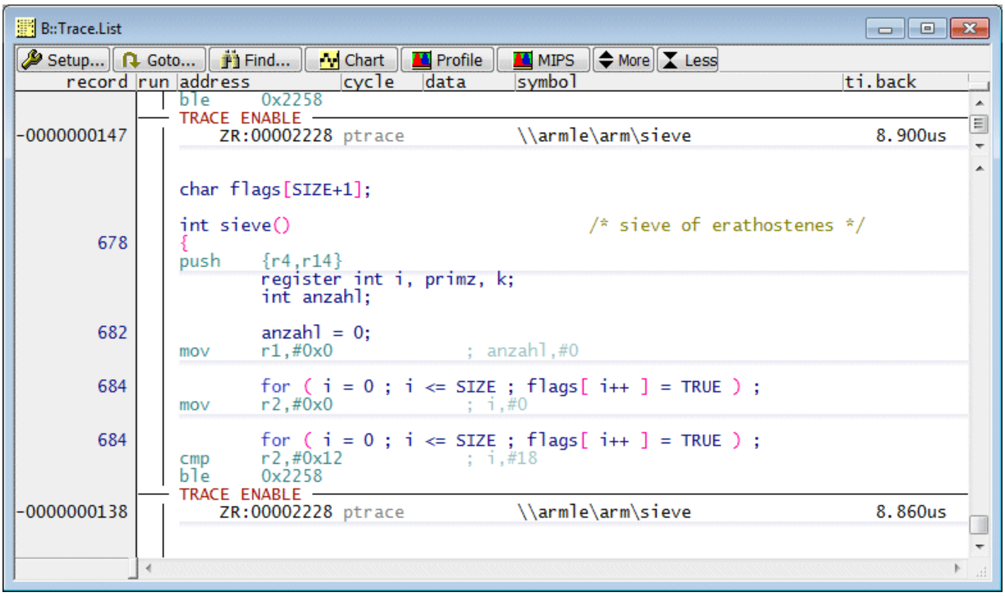
Example 1: Advise the ETM to generate trace information only for the entry to the function sieve.

1. Set a program breakpoint to the entry of the function sieve and select the action TraceEnable.



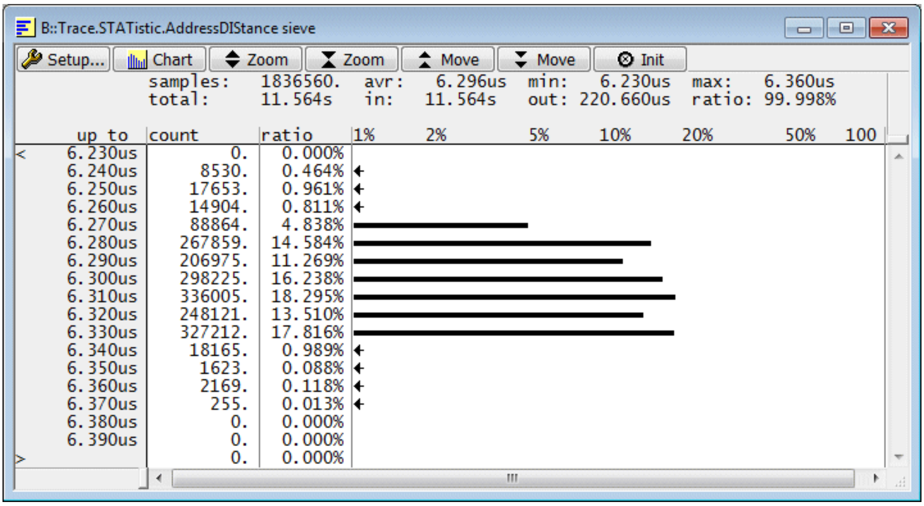
2. Start the program execution and stop it.

3. Display the result.



Use the following command, if you want to perform a statistical evaluation of this event.

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

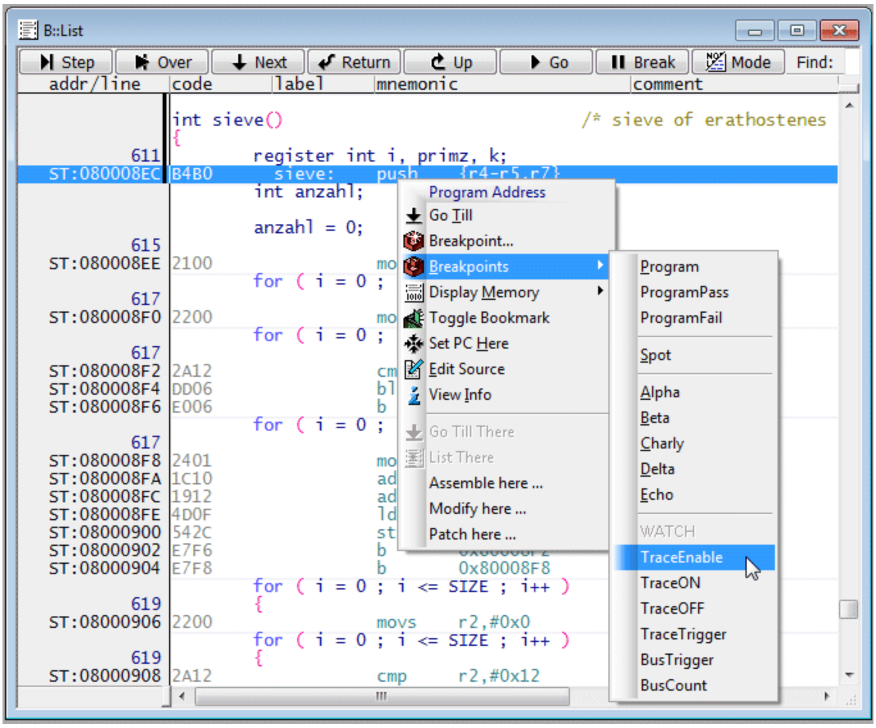


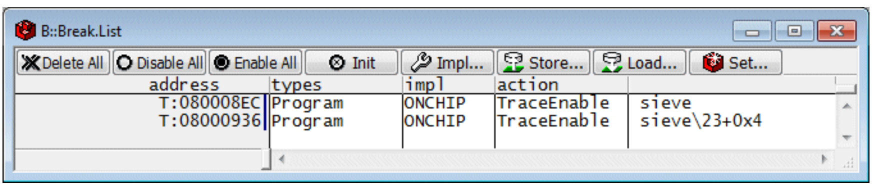
	For ARM Cortex CPUs: When measuring execution times while only parts of the program flow have been traced (by using trace filters TraceEnable, TraceON or TraceOFF) **you should not use external (tool based) time stamps**. On ARM Cortex CPUs already generated trace packets might not leave your chip when the trace filter disables the ETM/PTM. The packets are then recorded when the ETM/PTM gets re-enabled, which gives wrong external timestamps. Best is to use **ETM.TImeMode CycleAccurate**, **when using trace-filters**. Alternatively you can use **ETM.TImeMode SyncTimeStamps** or **ETM.TImeMode AsyncTimeStamps** (if an internal chip-internal time stamper is available) See command **ETM.TImeMode** in **“ARM-ETM Trace”** (trace_arm_etm.pdf) for more details.

	If you use external (tool based) time stamps (**ETM.TImeMode External**) to measure execution times while only parts of the program flow have been traced (by using trace filters TraceEnable, TraceON or TraceOFF), ensure that the port-filter of you trace preprocessor (or CombiProbe or µTrace) is set to ON. (**Trace.PortFilter ON**) Otherwise trace packets might be highly delayed in your trace tool.

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

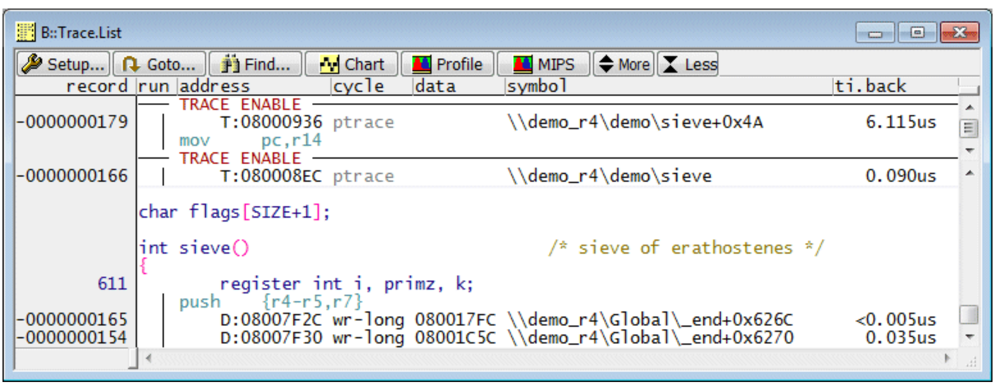
1. Mark the entry and the exit of the function sieve with an TraceEnable breakpoint.





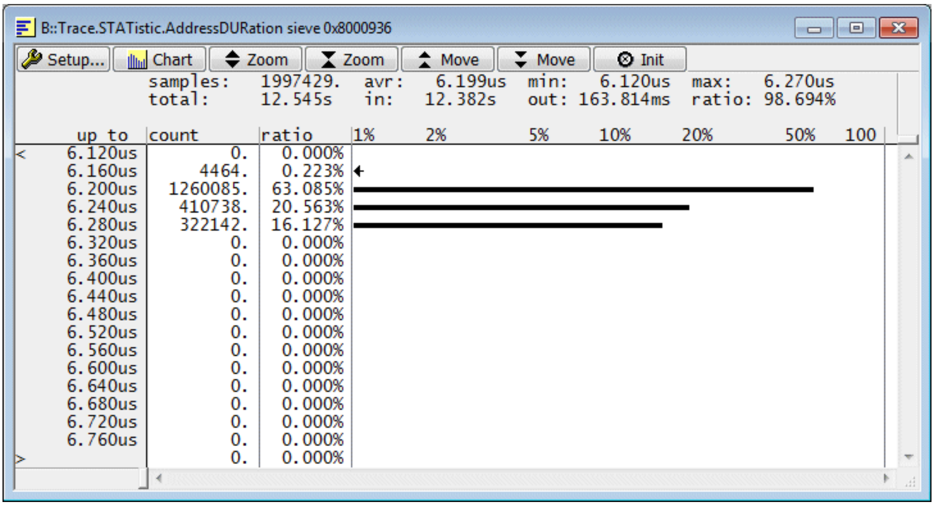
2. Start the program execution and stop it.

3. Display the result.



Use the following command, if you want to get a statistical analysis of the time spent between the entry and exits of the function sieve. (The function sYmbol.EXIT(*<func>*) returns the exit address of a function.)

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



For ARM Cortex CPUs:
When measuring execution times while only parts of the program flow have been traced (by using trace filters TraceEnable, TraceON or TraceOFF) **you should not use external (tool based) time stamps**. On ARM Cortex CPUs already generated trace packets might not leave your chip when the trace filter disables the ETM/PTM. The packets are then recorded when the ETM/PTM gets re-enabled, which gives wrong external timestamps.

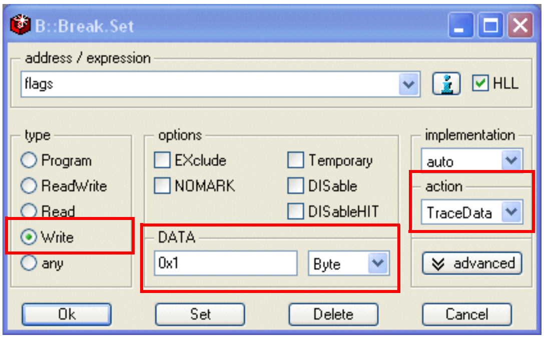
Best is to use **ETM.TImeMode CycleAccurate**, **when using trace-filters**. Alternatively you can use **ETM.TImeMode SyncTimeStamps** or **ETM.TImeMode AsyncTimeStamps** (if an internal chip-internal time stamper is available)

See command **ETM.TImeMode** in **“ARM-ETM Trace”** (trace_arm_etm.pdf) for more details.

If you use external (tool based) time stamps (**ETM.TImeMode External**) to measure execution times while only parts of the program flow have been traced (by using trace filters TraceEnable, TraceON or TraceOFF), ensure that the port-filter of you trace preprocessor (or CombiProbe or µTrace) is set to ON. (**Trace.PortFilter ON**) Otherwise trace packets might be highly delayed in your trace tool.

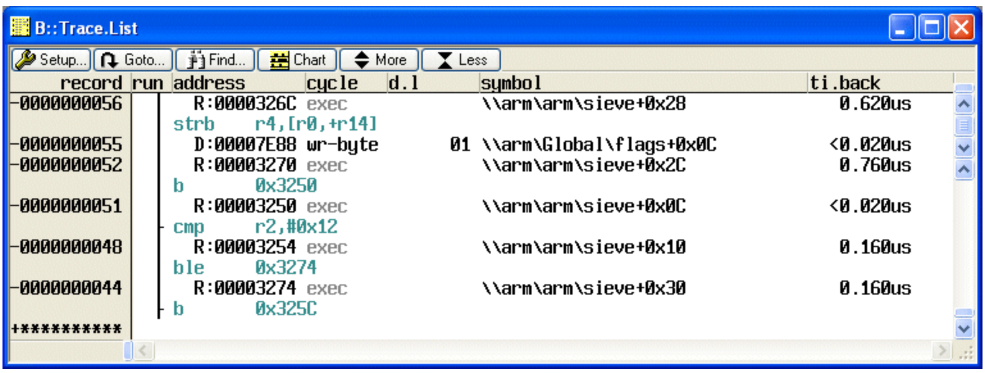
Example: Advise the ETM to generate trace information for the complete instruction flow and for the write accesses where 1 is written as a byte to the variable flags.

1. Set a Write breakpoint to the HLL variable flags, define DATA 1 and select the action TraceData.



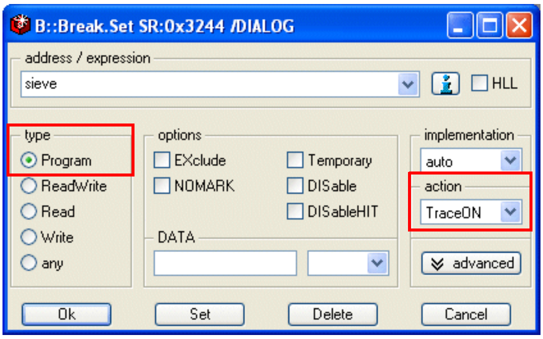
2. Start the program execution and stop it.

3. Display the result.

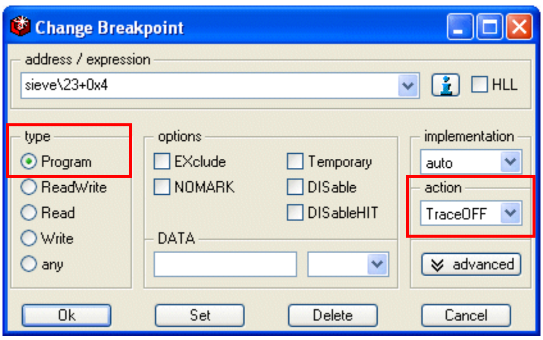


Example: Sample only the function sieve.

1. Set a Program breakpoint to the entry of the function sieve and select the action TraceON.

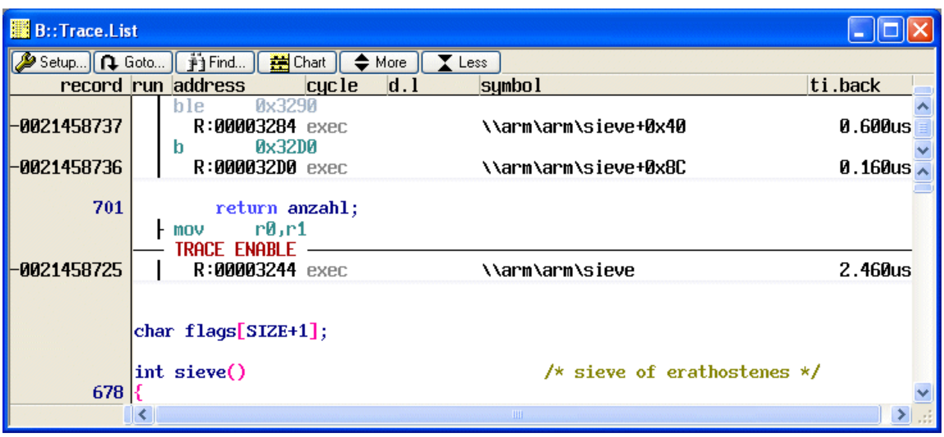


2. Set a Program breakpoint to the exit of the function sieve and select the action TraceOFF.



3. Start the program execution and stop it.

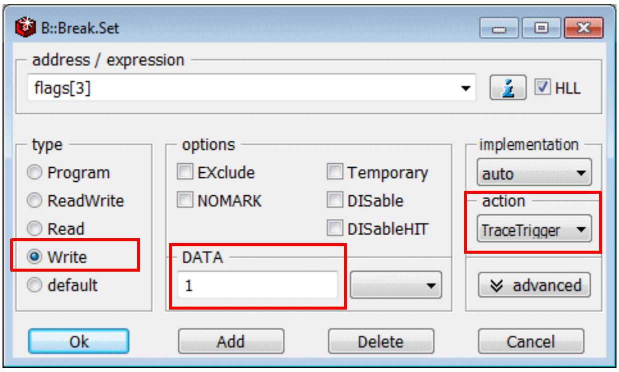
4. Display the result.



The ETM stops the generation for the trace information before trace information is generated for the event the causes the stop.

Example: Stop the recording to the trace buffer after 1 was written to flags[3].

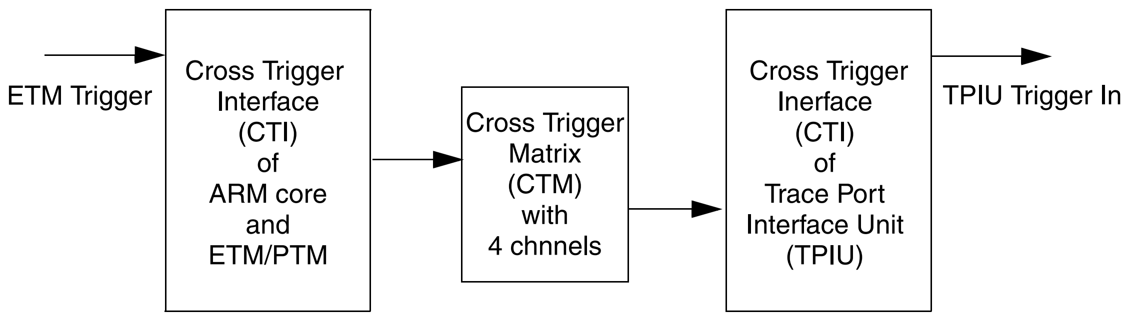
1. Set a write breakpoint to flags[3], define DATA 1 and select the action TraceTrigger.



2. Establish the connection between the ETM Trigger and the TPIU via the Cross Trigger Interface (CTI) if required.

   This is only required for chips with ETMv3 or PTM and CoreSight debug infrastructure (Cortex-R4/R5 and Cortex-A5 to A17). You can skip this step for Cortex-M and cores with ETMv4 (with ETMv4 the trigger is propagated via the trace bus (ATB).)

   You have to set up both the Cross Trigger Interface (CTI) of you trigger-source - here the ETM/PTM - and the sink for the trace trigger - here the TPIU (or ETF or ETR).




```
; core specific example

; configure ETM trigger to channel 3

; enable Core/ETM CTI
Data.Set APB:0x80001000 %Long 1
; map CTITRIGIN[6] (ETM Trigger Out) to channel 3
Data.Set APB:0x80001038 %Long Data.Long(APB:0x80001038)|0x4

; configure channel 3 to TPIU Trigger In

; enable System/TPIU CTI
Data.Set APB:0x80002000 %Long 1
; Map channel 3 to CTITRIGOUT[3] (TPIU Trigger In)
Data.Set APB:0x800020AC %Long Data.Long(APB:0x800020AC)|0x4
```

The peripheral file “percti.per” in your TRACE32 system folder can help you to configure the CTIs:

```
DO "~~/percti.per" <cti_base_address>
```

Check your chips data sheet, for the base addresses of the CTI interface for both ETM and TPIU (or ETF or ETR).

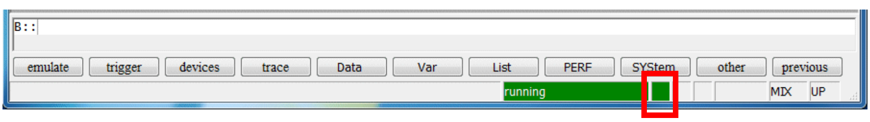
The following CTI inputs and outputs are use by the different ARM cores:

ARM core	ETM Trigger Output to CTM Channel	Core Break Input from CTM Channel	Core Halt Output to CTM Channel
ARM9 / ARM11	CTITRIGIN[6]	CTITRIGOUT[0]	CTITRIGIN[0]
Cortex-R4	CTITRIGIN[6]	CTITRIGOUT[0]	CTITRIGIN[0]
Cortex-A9	CTITRIGIN[6]	CTITRIGOUT[0]	CTITRIGIN[0]
Cortex-M3	CTITRIGIN[7]	CTITRIGOUT[0]	CTITRIGIN[0]
Cortex-R7	CTITRIGIN[2]	CTITRIGOUT[0]	CTITRIGIN[0]
ARMv8-A	CTITRIGIN[4]	CTITRIGOUT[0]	CTITRIGIN[0]

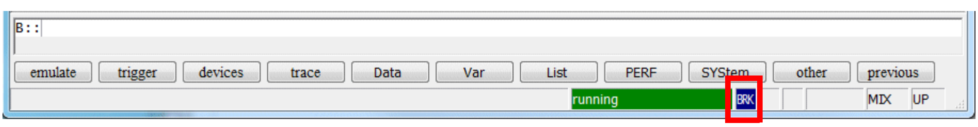
Trace Sink	Trigger Input from CTM Channel
TPIU	CTITRIGOUT[3]
ETF	CTITRIGOUT[7]
ETR	CTITRIGOUT[1]

3. Configure an optional delay with **Trace.TDelay** *<time>* | *<ETM cycles>* | *<percent of trace-buffer>* When the trace trigger occurs, the recording will stop after the specified delay.

4. Start the program execution.



The Trace State field in the command line is **green**, as long a the trace recording is active



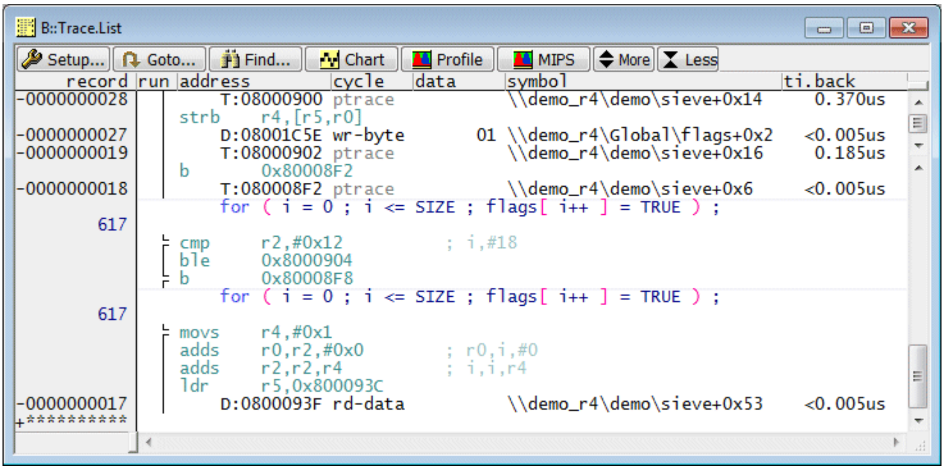
The Trace State field in the command line becomes **blue** after the trace recording was stopped by the Trigger (BRK)

5. Display the result.

Displaying the result requires access to the target memory in order to read the code information. To display the result:

- Stop the program execution or

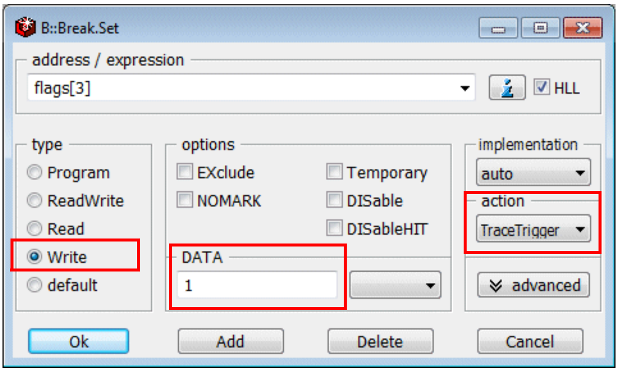
- Load the code to the TRACE32 Virtual Memory before you use the TraceTrigger action (`Data.LOAD <file> /VM`).



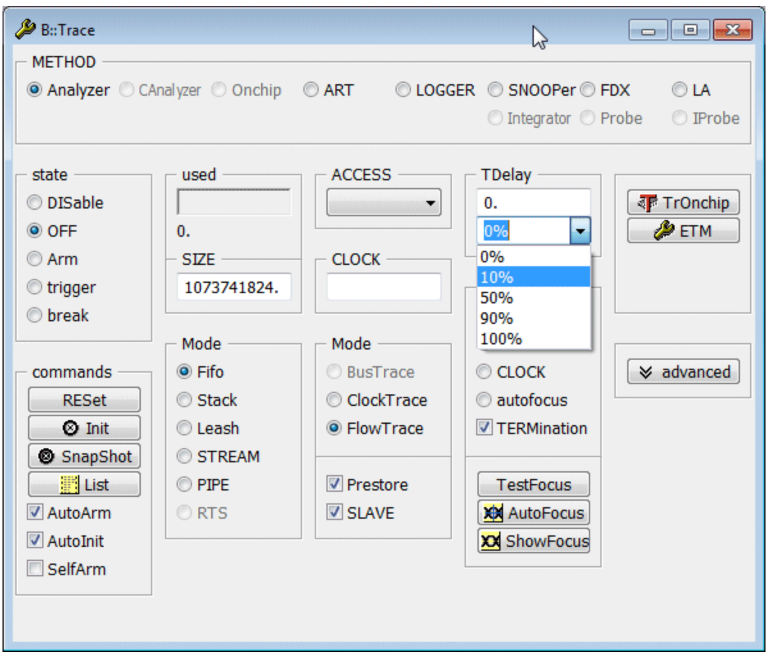
The trace recording is stopped before the event that caused the triggered is exported by the ETM.

Example: Stop the sampling to the trace buffer after 1 was written to flags[3] and another 10% of the trace buffer is filled.

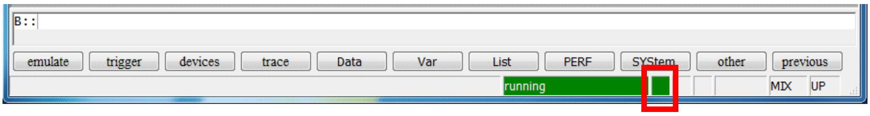
1. Set a write breakpoint to flags[3], define DATA 1 and select the action TraceTrigger.



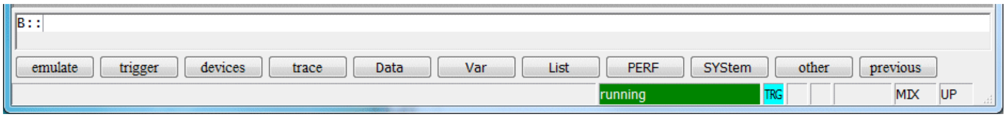
2. Define the trigger delay (TDelay) in the Trace Configuration Window.



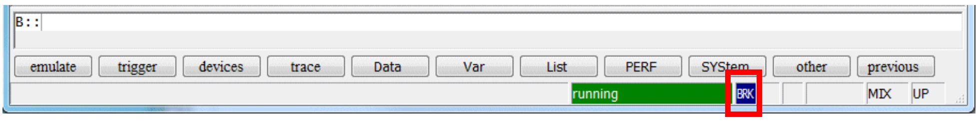
3. Start the program execution.



The Trace State field in the command line is **green**, as long a the trace recording is active

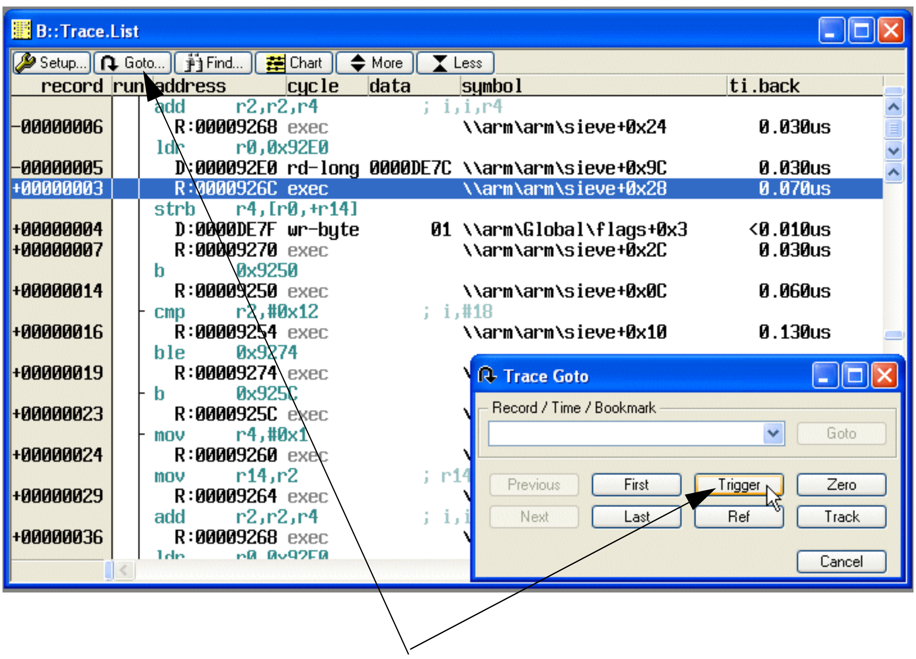


The Trace State field in the command line becomes **cyan** after the Trigger occured (TRG) and the trigger delay (TDelay) starts



The Trace State field in the command line becomes **blue** after the trace recording was stopped because the trigger delay ran down (BRK)

4. Display the result.

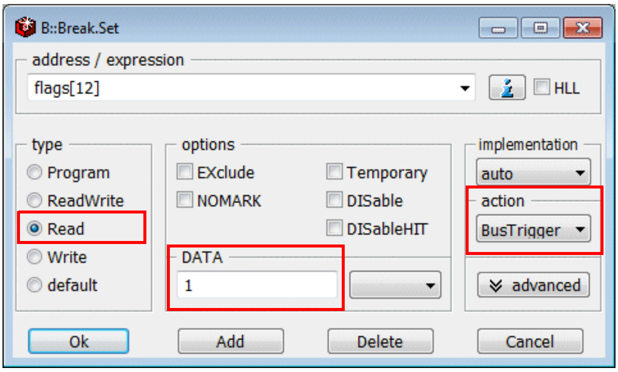


Push the ***Trigger*** button in the ***Trace Goto*** window to find the record, where TraceTrigger was accepted by the trace. Here the sign of the record numbers has changed.

Example: Indicate with a pulse on ETM External Output 1 that 0 was read from flags[12].

In order to measure this pulse on ETM External Output 1 you need to check where the ETM External Output 1 can be measured on your core.

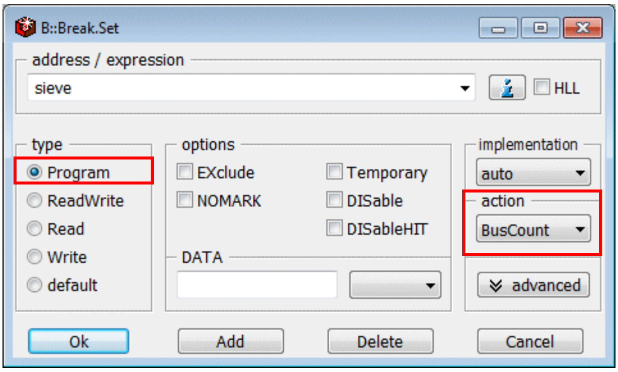
1. Set a read breakpoint to flags[12], define DATA 1 and select the action BusTrigger.



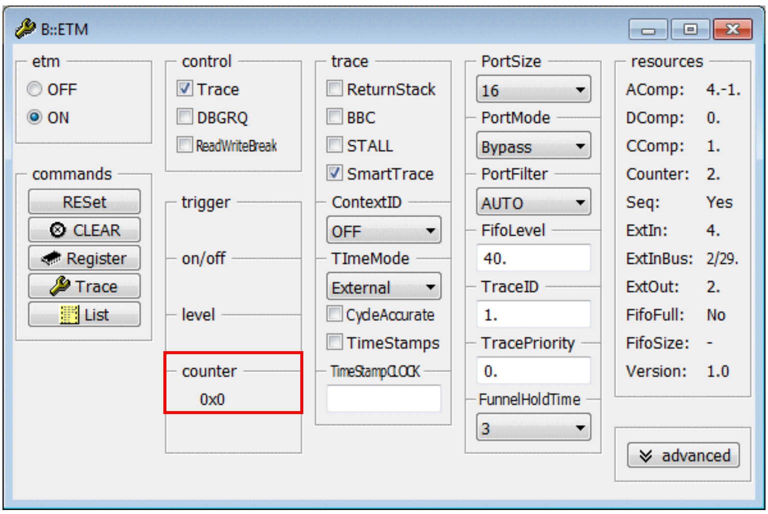
2. Measure the result.

Example: Advise the ETM to decrement its counter at each entry to the function sieve.

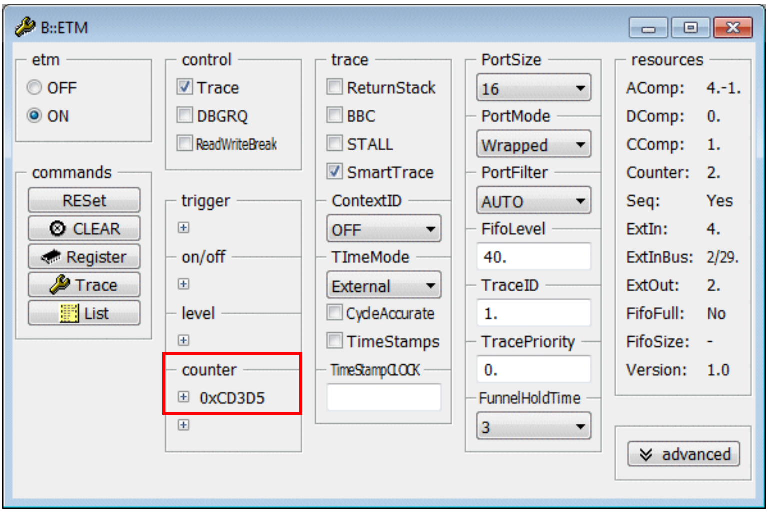
1. Set a program breakpoint to the entry of the function sieve and select the action BusCount.



2. Use the ETM configuration window to observe the counter.



3. Start the program execution



OS-Aware Tracing

OS-aware tracing is relatively simple if you use an operating system that does not use dynamic memory management (e.g. eCos).

The OS-aware tracing for an operating that uses dynamic memory management to handle processes/tasks is more complex. That is why this is explained in a separate chapter.

OS (No Dynamic Memory Management)

Activate the TRACE32 OS Awareness (Supported OS)

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

Lauterbach provides configuration files for most common available OS.

If your kernel is compiled with symbol and debug information, the adaptation to your OS can be activated as follows:

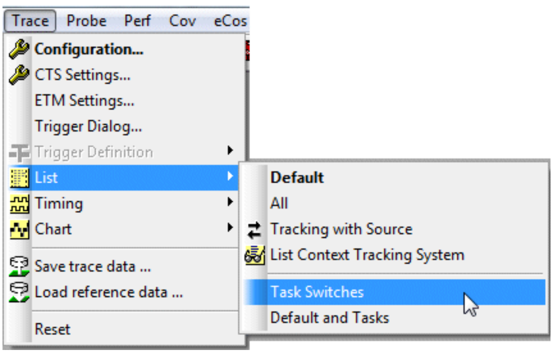
TASK.CONFIG *<file>*	Configures the OS debugger using a configuration file provided by Lauterbach
MENU.ReProgram *<file>*	Program a ready-to-run OS menu
HELP.FILTER.Add *<filter>*	Add the help information for the OS debugger

All necessary files can be found under `~~/demo/arm/kernel`, where ~~ expands to the TRACE32 system directory.

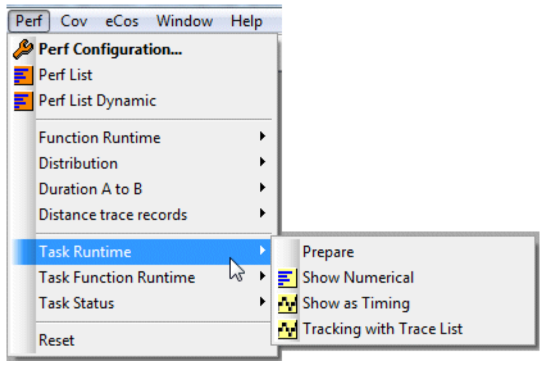
Example for eCos:

```
; enable eCos specific commands and features within TRACE32 PowerView
TASK.CONFIG ~~/demo/arm/kernel/ecos/ecos.t32
```

```
; extend the Trace menu and the Perf menu for OS-aware tracing
MENU.ReProgram ~~/demo/arm/kernel/ecos/ecos.men
```

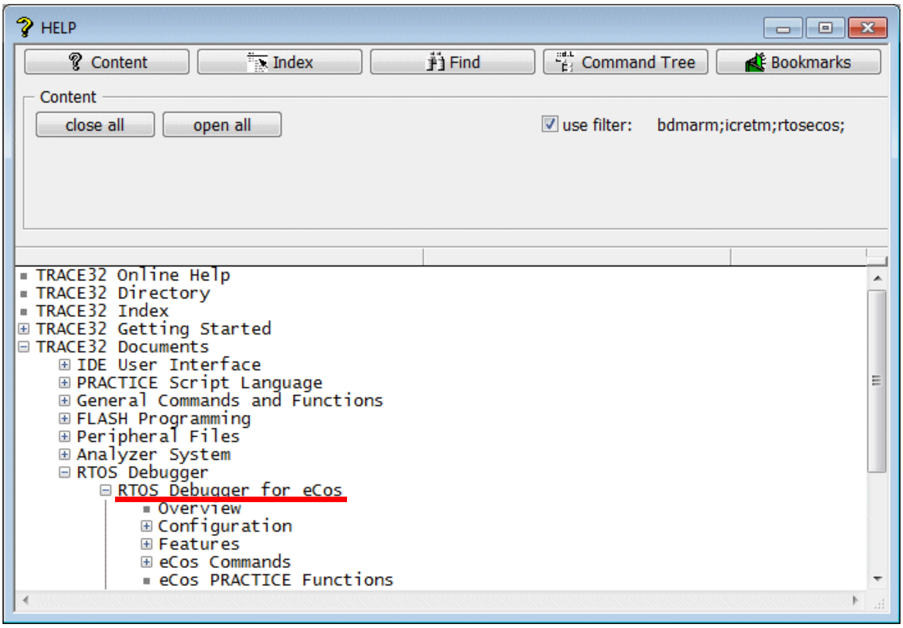


The **Trace** menu is extended by the commands
- Task Switches
- Default and Tasks



The **Perf** menu is extended by the commands
- Task Runtime
- Task Function Runtime
- Task Status

```
; enable eCos-specific help
HELP.FILTER.Add rtosecos
```



There a two methods how task switch information can be generated by the ETM:

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

  This method requires that the ETM can generate Data Address information and Data Value information for write accesses (see table below). If this is possible this method is the preferred one because it **does not require any support from the operating system.**

- **By generating a Context ID packet**

  This method should only be used, if the ETM can not generate Data Address information and Data Value information for write accesses (see table below). The reason is that it requires support from the **operating system**. If the generation of Context ID packets is not supported by your operating system it **has to be patched** in order to provide this information.

Core	ETM Version	Data Address	Data Value	Context ID
ARM7 **ARM9**	ETMv1	■	■	■
ARM9	ETMv3	■	■	■
ARM11	ETMv3	■	■	■
Cortex-M3/M4 **Cortex-M23**	ETMv3	(DWT)	(DWT)	-
Cortex-M7 **Cortex-M33**	ETMv4 Config. 1	(DWT)	(DWT)	-
Cortex-R4 **Cortex-R5**	ETMv3	■	■	■
Cortex-R7/R8 **Cortex-R52**	ETMv4 Config. 3	■	■	■
Cortex-A5 **Cortex-A7**	ETMv3	■	■	■
Cortex-A8	ETMv3	■	-	■
Cortex-A9 **Cortex-A15** **Cortex-A17**	PTM	-	-	■
Cortex-A3x **Cortex-A5x** **Cortex-A7x**	ETMv4 Config. 2	-	-	■

Each operating system has a variable that contains the information which task is currently running. This variable can hold a task ID, a pointer to the task control block or something else that is unique for each task.

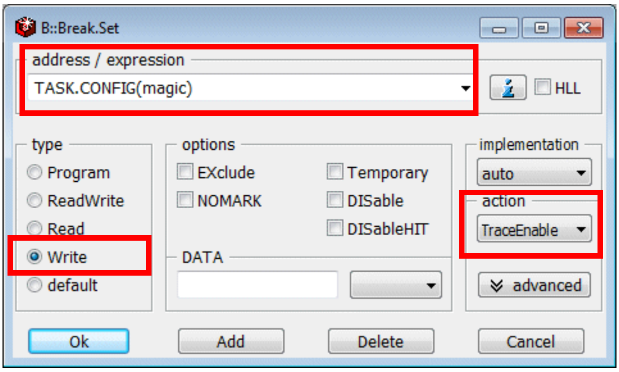
One way to export task switch information is to advise the ETM to generate trace information when a write access to this variable occurs.

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

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

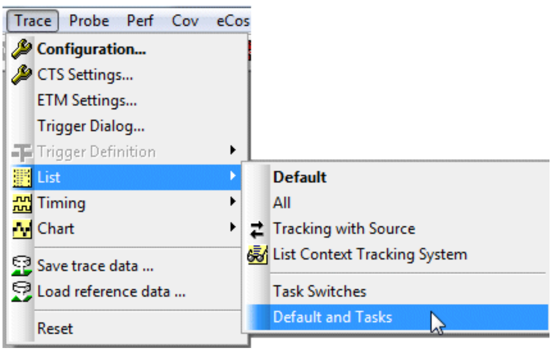
Example: Advise the ETM to generate only trace information on task switches.

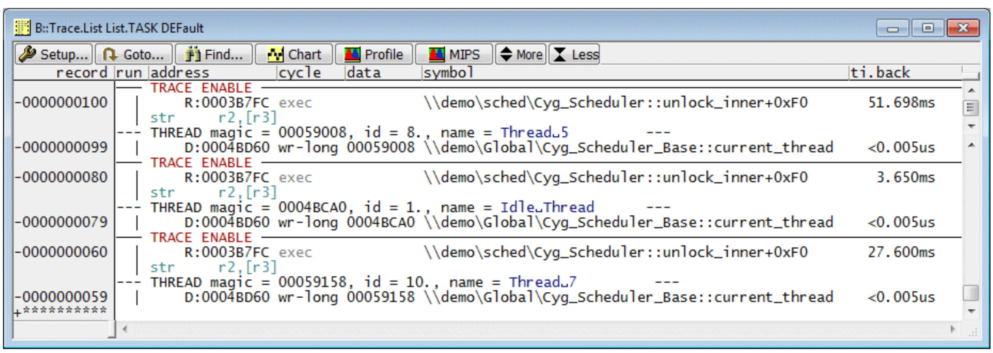
1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic) and select the trace action TraceEnable.



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

3. Display the result.





A Context ID packet is generated when the OS updates the **Context ID Register** (CONTEXTIDR) on a task switch.

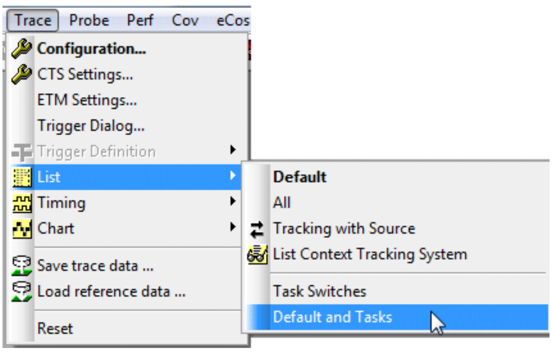
The generation of Context ID packets has to be enabled within TRACE32.

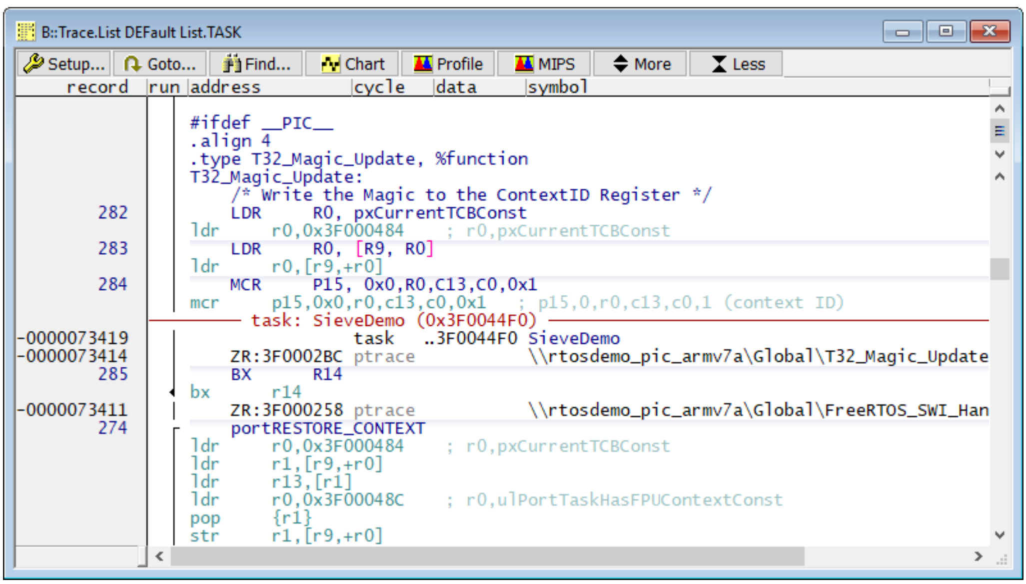
```
ETM.ContextID 32          ; enable the generation of Context ID packets and
                          ; inform TRACE32 that the Context ID is a 32-bit
                          ; value
```

Example:

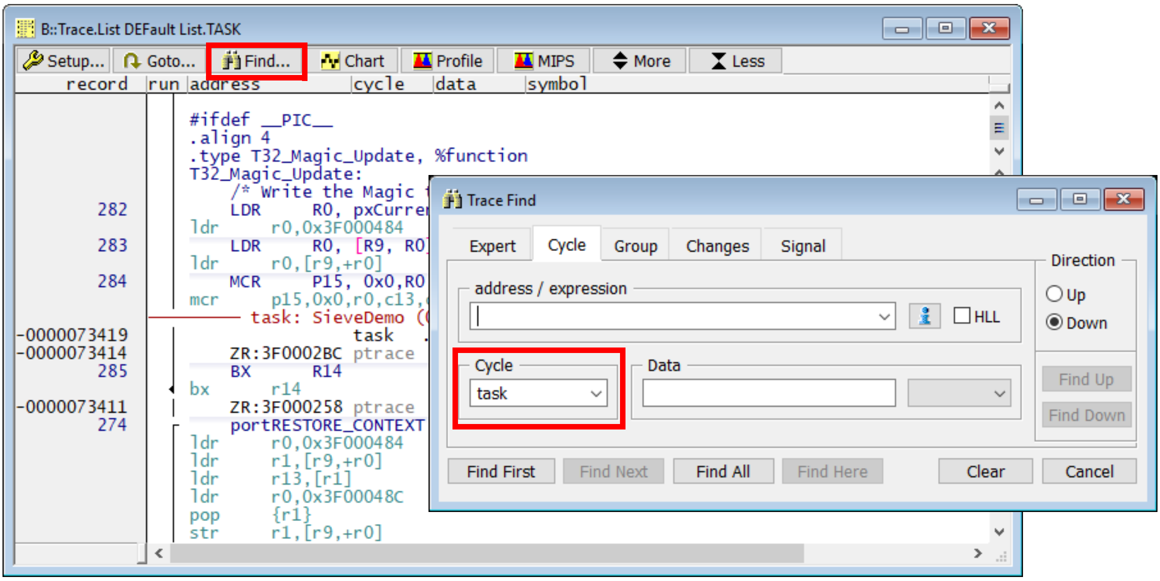
1. Start and stop the program execution to fill the trace buffer.

2. Display the result.





Searching for the Context IDs can be performed as follows:



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 testrtos.ad
```

Trace.SAVE saves the trace row data plus decompressed addresses, data and op-codes, but not the task names.

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

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

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

Trace.LOAD testrtos.ad                  ; load the trace file

Data.LOAD.Elf demo.elf /NoCODE          ; load the symbol and debug
                                        ; information

TASK.CONFIG ecos                        ; activate the TRACE32
                                        ; OS Awareness

TASK.NAME.Set 0x58D68 "Thread 1"        ; assign the task name to the
                                        ; saved task identifier

...                                     ; assign the other task names

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

Enable an OS-aware Tracing (Not-Supported OS)

If you use an OS that is not supported by Lauterbach you can use the “simple” awareness to configure your debugger for OS-aware tracing.

Current information on the “simple” awareness can be found under ~~/demo/kernel/simple/readme.txt.

Each operating system has a variable that contains the information which task is currently running. This variable can hold a task ID, a pointer to the task control block or something else that is unique for each task.

Use the following command to inform TRACE32 about this variable:

```
TASK.CONFIG ~~/demo/kernel/simple/simple.t32 <var> Var.SIZEOF(<var>)
```

If current_thread is the name of your variable the command would be as follows:

```
TASK.CONFIG ~~/demo/kernel/simple/simple current_thread
Var.SIZEOF(current_thread)
```

The OS-aware debugging is easier to perform, if you assign names to your tasks.

TASK.NAME.Set *<task_id> <name>*	Specify a name for your task
TASK.NAME.view	Display all specified names

```
TASK.NAME.Set 0x58D68 "My_Task 1"
```

The “simple” awareness only supports task switches that are exported for the write accesses to the variable that contains the information which task is currently running.

The “simple” awareness does currently not support context ID packets.

OS+MMU (Dynamic Memory Management)

Since Linux is widely used, it is taken as example target OS for this training chapter.

Activate the TRACE32 OS Awareness

Please refer to **“Training Linux Debugging”** (training_rtos_linux.pdf) on how to activate the Linux awareness on your target.

If you use a different OS that uses dynamic memory management to handle processes/tasks refer to the corresponding target **OS Awareness Manual** (rtos_<*os*>.pdf).

There a two methods how process/thread switch information can be generated by the ETM:

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

  This method requires that the ETM can generate Data Address information and Data Value information for write accesses (see table below). If this is possible this method is the preferred one because it does not require any support from the operating system.

- **By generating a Context ID packet.**

  This method should only be used, if the ETM can not generate Data Address information and Data Value information for write accesses (see table below). The reason is that it requires support from the operating system. If the generation of Context ID packets is not supported by your operating system it has to be patched in order to provide this information.

Core	ETM Version	Data Address	Data Value	Context ID
ARM7 **ARM9**	ETMv1	■	■	■
ARM9	ETMv3	■	■	■
ARM11	ETMv3	■	■	■
Cortex-M3/M4 **Cortex-M23**	ETMv3	(DWT)	(DWT)	-
Cortex-M7 **Cortex-M33**	ETMv4 Config. 1	(DWT)	(DWT)	-
Cortex-R4 **Cortex-R5**	ETMv3	■	■	■
Cortex-R7/R8 **Cortex-R52**	ETMv4 Config. 3	■	■	■
Cortex-A5 **Cortex-A7**	ETMv3	■	■	■
Cortex-A8	ETMv3	■	-	■
Cortex-A9 **Cortex-A15** **Cortex-A17**	PTM	-	-	■
Cortex-A3x **Cortex-A5x** **Cortex-A7x**	ETMv4 Config. 2	-	-	■

Each operating system has a variable that contains the information which process/thread is currently running. This variable can hold a process/thread, a pointer to the process control block or something else that is unique for each process/thread.

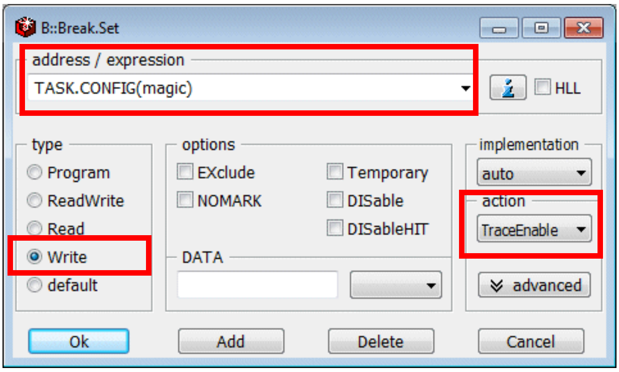
One way to export process/thread switch information is to advise the ETM to generate trace information when a write access to this variable occurs.

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

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

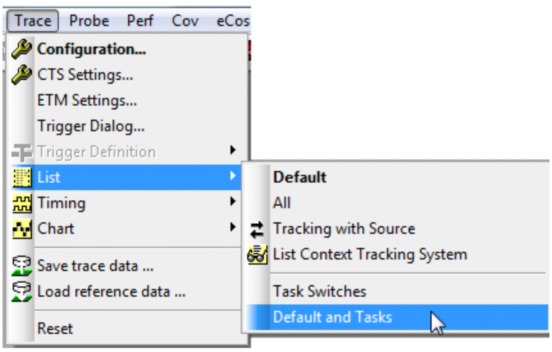
Example: Advise the ETM to generate only trace information on a process/thread switch.

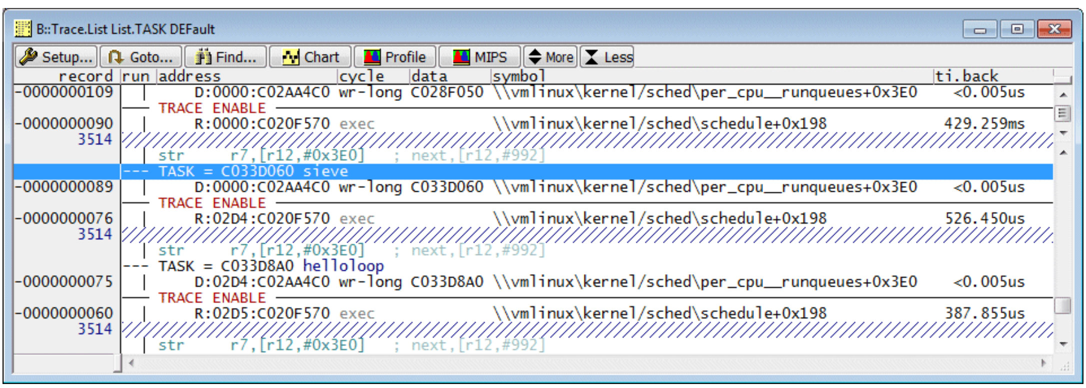
1. Set a Write breakpoint to the address indicated by TASK.CONFIG(magic) and select the trace action TraceEnable.



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

3. Display the result.





A Context ID packet is generated when the OS updates the **Context ID Register** (CONTEXTIDR) on a process/thread switch.

Linux, in most cases, writes only the **Linux Address Space ID** (ASID) to CONTEXTIDR. This allows tracking the program flow of the processes and evaluating of the process switches. But it does not provide performance information on threads. In particular the idle thread can not be detected.

To allow a detailed performance analysis also on Linux threads, the **Linux Address Space ID** and the **Linux PID** (each thread gets its own PID - despite the name) has to be written to CONTEXTIDR. The swapper gets the PID -1.

Lauterbach provide a Linux patch, that takes care that Linux writes the required information to the **Context ID Register**.

The information written via the Lauterbach patch to CONTEXTIDR is called **TRACE32 traceid** in the following.

Linux PID	ASID	**TRACE32 traceid**
31 … 8	7 … 0	

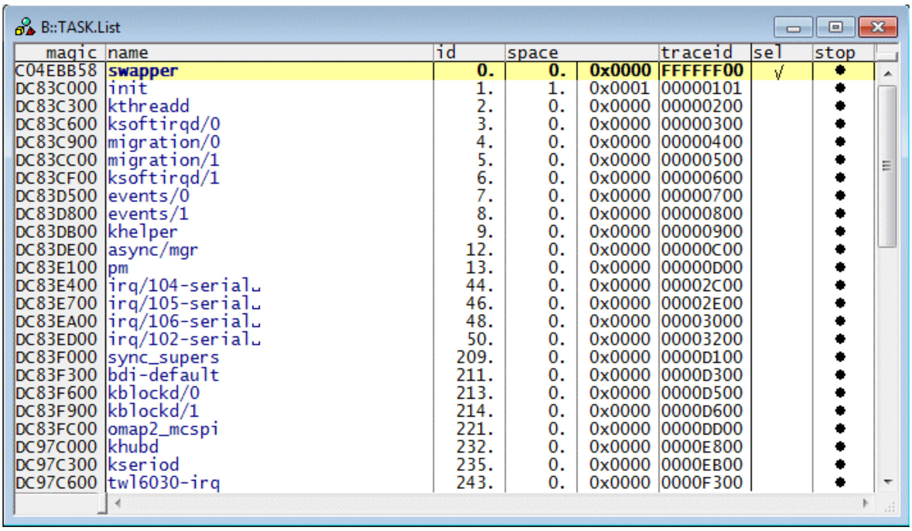
The generation of Context ID packets is disabled after an ETM reset, so it has to be enabled within TRACE32.

```
ETM.ContextID 32          ; enable the generation of Context ID packets and
                          ; inform TRACE32 that the Context ID is a 32-bit
                          ; value
```

If you use the Lauterbach Linux patch, you have to do the following setting within TRACE32:

```
TASK.Option THRCTX ON
```

The TRACE32 **traceid** for the processes/threads can be checked in the **TASK.List** window.



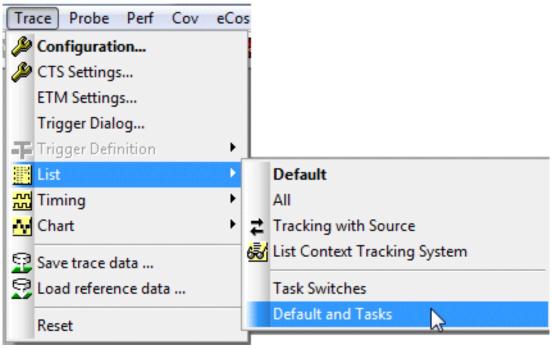
magic	name	id	space		traceid	sel	stop
C04EBB58	swapper	0.	0.	0x0000	FFFFFF00	√	•
DC83C000	init	1.	1.	0x0001	00000101		•
DC83C300	kthreadd	2.	0.	0x0000	00000200		•
DC83C600	ksoftirqd/0	3.	0.	0x0000	00000300		•
DC83C900	migration/0	4.	0.	0x0000	00000400		•
DC83CC00	migration/1	5.	0.	0x0000	00000500		•
DC83CF00	ksoftirqd/1	6.	0.	0x0000	00000600		•
DC83D500	events/0	7.	0.	0x0000	00000700		•
DC83D800	events/1	8.	0.	0x0000	00000800		•
DC83DB00	khelper	9.	0.	0x0000	00000900		•
DC83DE00	async/mgr	12.	0.	0x0000	00000C00		•
DC83E100	pm	13.	0.	0x0000	00000D00		•
DC83E400	irq/104-serial	44.	0.	0x0000	00002C00		•
DC83E700	irq/105-serial	46.	0.	0x0000	00002E00		•
DC83EA00	irq/106-serial	48.	0.	0x0000	00003000		•
DC83ED00	irq/102-serial	50.	0.	0x0000	00003200		•
DC83F000	sync_supers	209.	0.	0x0000	0000D100		•
DC83F300	bdi-default	211.	0.	0x0000	0000D300		•
DC83F600	kblockd/0	213.	0.	0x0000	0000D500		•
DC83F900	kblockd/1	214.	0.	0x0000	0000D600		•
DC83FC00	omap2_mcspi	221.	0.	0x0000	0000DD00		•
DC97C000	khubd	232.	0.	0x0000	0000E800		•
DC97C300	kseriod	235.	0.	0x0000	0000EB00		•
DC97C600	twl6030-irq	243.	0.	0x0000	0000F300		•

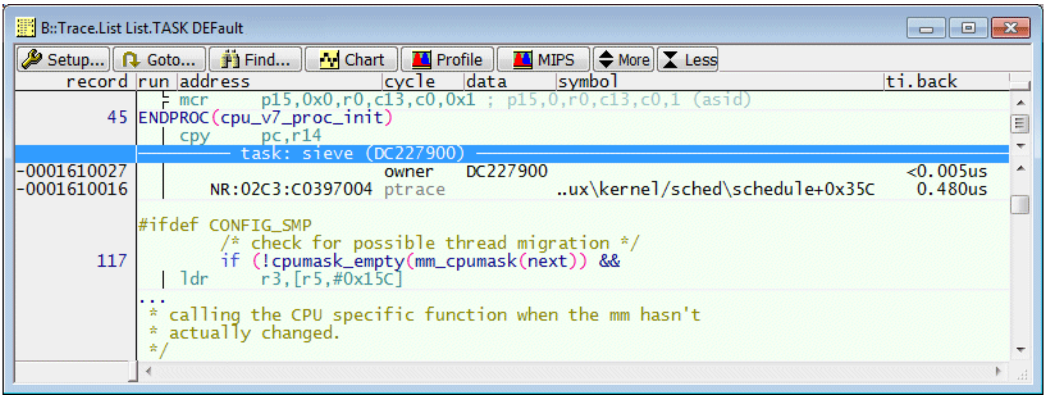
magic	Address of process descriptor *task_struct*
id	Linux Process IDentifier (PID)
space	Identifier for virtual address space (decimal and hex), TRACE32 space ID
traceid	Information exported via Context ID packet

Example:

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

2. Display the result.





The TRACE32 Instruction Set Simulator can be used for a belated Linux-aware trace evaluation.

Example for the PandaBoard

1. Perform the following steps to save the relevant information within TRACE32.

```
; save the trace contents
Trace.SAVE belated_linux.ad

; save the whole Linux address range - code and data area
Data.SAVE.Binary image.bin ASD:0x80000000--0x9fffffff

; generate a script that re-configures the important MMU registers
OPEN #1 MMU_Register.cmm /Create
WRITE #1 "PER.SET C15:0x1 %Long " Data.Long(C15:0x1)
WRITE #1 "PER.SET C15:0x2 %Long " Data.Long(C15:0x2)
WRITE #1 "PER.SET C15:0x102 %Long " Data.Long(C15:0x102)
CLOSE #1
```

2. Set up the TRACE32 Instruction Set Simulator for a belated Linux-aware trace evaluation.

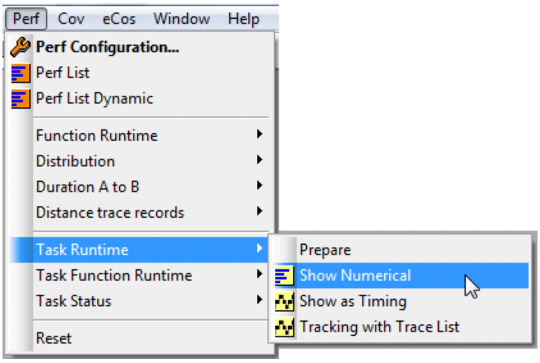
```
RESet

; select the OMAP4430 as target chip
SYStem.CPU OMAP4430

; Extends the address scheme of the debugger to include memory
; spaces
SYStem.Option MMUSPACES ON

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

; set the important MMU register
DO MMU_Register.cmm

; load the binary file you saved before
Data.LOAD.Binary image.bin ASD:0x80000000--0x9fffffff

; load the Linux symbol and debug information
Data.LOAD.Elf vmlinux /NoCODE

; specify MMU table format
MMU.FORMAT linux swapper_pg_dir 0xc0000000--0xdfffffff  0x80000000
TRANSlation.COMMON 0xc000000--0xfffffff
MMU.SCAN ALL
TRANSlation.ON
```

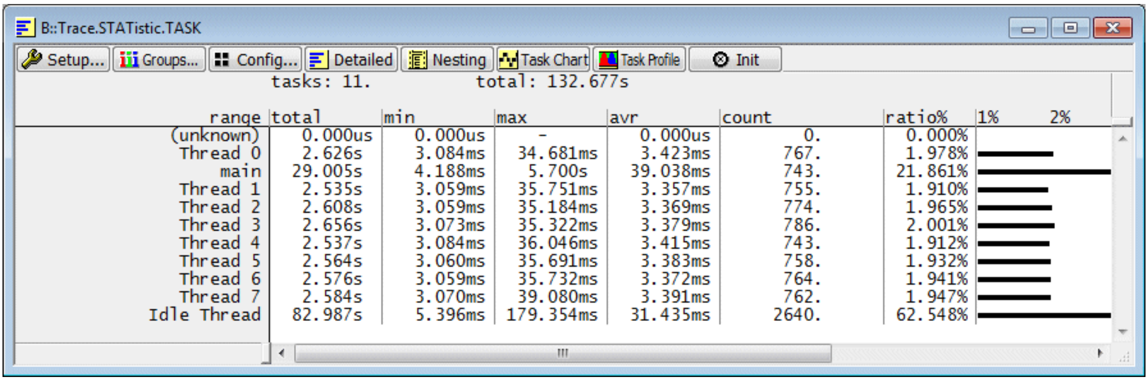
```
; configure the Linux-awareness
TASK.CONFIG ~~/demo/arm/kernel/linux/linux-<version>.x/linux.t32
MENU.ReProgram ~~/demo/arm/kernel/linux/linux-<version>.x/linux.men
HELP.FILTER.Add rtoslinux

; load the saved trace file
Trace.LOAD belated_linux.ad

; load the symbol and debug information for the running processes
&address_space=TASK.PROC.SPACEID("sieve")
Data.LOAD.Elf &address_space:0 /NoCODE /NoClear
…

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

Specific Write Access vs. Context ID Packet

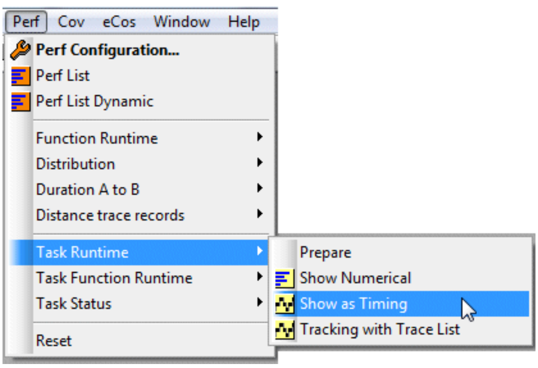
Specific Write Access	Context ID Packet
Requires an ETM that allows to export Data Address and Data Write Value information	Requires an ETM that allows to export Context ID Packets
No support from the OS required	Requires support from OS and/or patch
ETM can be advised to only generate trace information on the specific write access	Context ID information is only exported in conjunction with the instruction trace

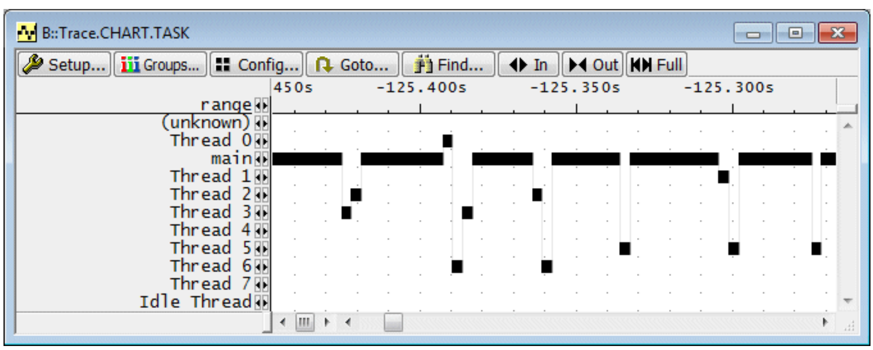
The following two commands perform a statistical analysis of the task/process/thread switches:



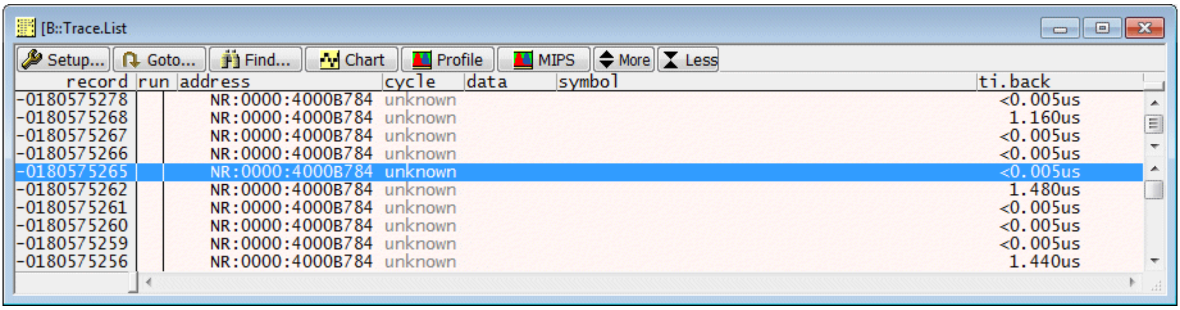


range	total	min	max	avr	count	ratio%
(unknown)	0.000us	0.000us	-	0.000us	0.	0.000%
Thread 0	2.626s	3.084ms	34.681ms	3.423ms	767.	1.978%
main	29.005s	4.188ms	5.700s	39.038ms	743.	21.861%
Thread 1	2.535s	3.059ms	35.751ms	3.357ms	755.	1.910%
Thread 2	2.608s	3.059ms	35.184ms	3.369ms	774.	1.965%
Thread 3	2.656s	3.073ms	35.322ms	3.379ms	786.	2.001%
Thread 4	2.537s	3.084ms	36.046ms	3.415ms	743.	1.912%
Thread 5	2.564s	3.060ms	35.691ms	3.383ms	758.	1.932%
Thread 6	2.576s	3.059ms	35.732ms	3.372ms	764.	1.941%
Thread 7	2.584s	3.070ms	39.080ms	3.391ms	762.	1.947%
Idle Thread	82.987s	5.396ms	179.354ms	31.435ms	2640.	62.548%



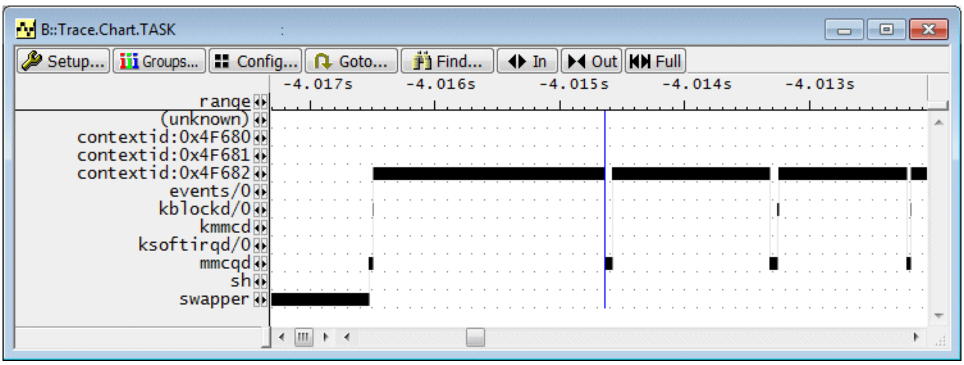


When a process is ended the process information is removed from the process table. Hereby the information on the virtual address space of the process is lost. This is the reason why TRACE32 can not decompress the trace information for ended processes.



Instructions of ended processes are marked as **unknown** in the trace, since TRACE32 has no access to the source code which is required to decompress the trace information.

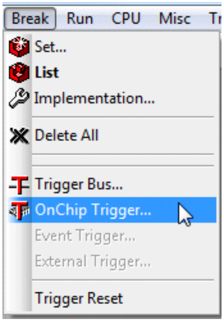
Since the process table no longer contains the name for an ended process, its process/thread ID or its TRACE32 traceid is displayed in the Trace.Chart/Trace.STATistic analyses.



Context ID Comparator

If your target-OS updates the **Context ID Register** (CONTEXTIDR) on a task/process/thread switch and if TRACE32 supports the Context ID for your OS, you can use the ETM Context ID Comparator to set up filter and trigger.

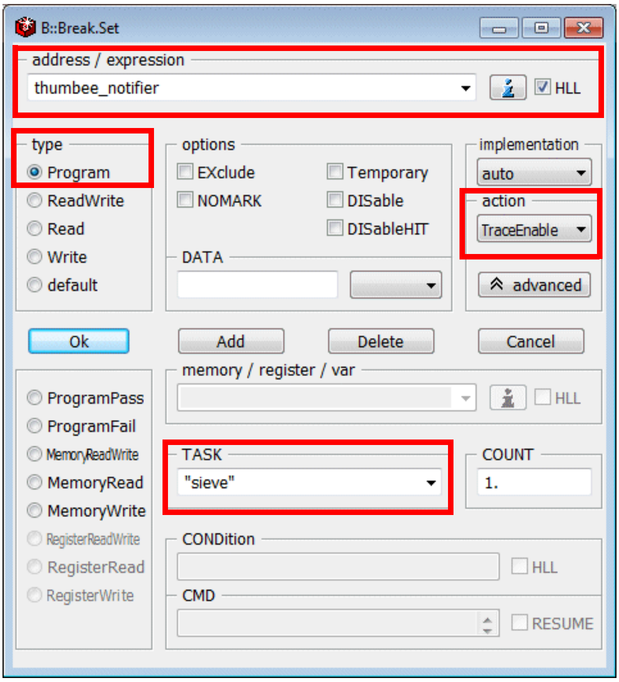
Before you can use the Context ID Comparator, you have to enable the usage of the Context ID within TRACE32.




```
TrOnchip.ContextID ON
```

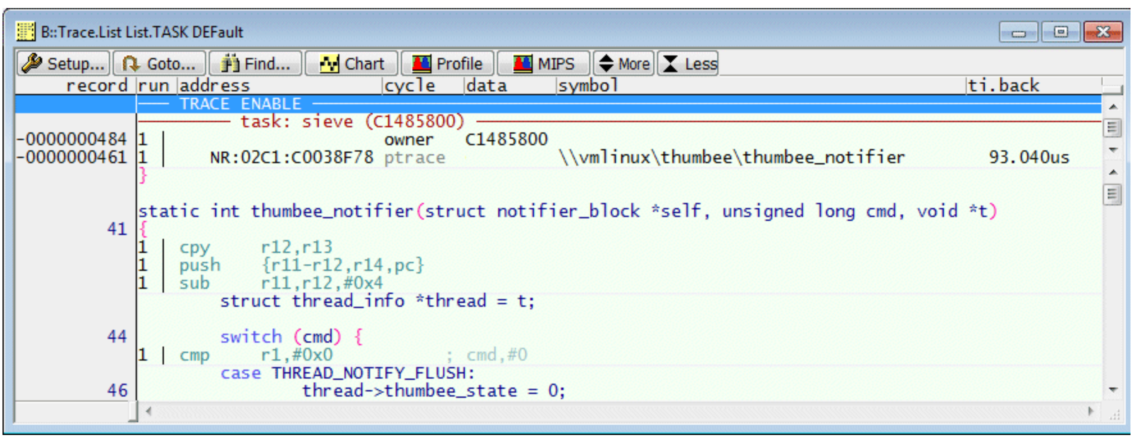
Example: Advise the ETM to only generate trace information if the process sieve executes an instruction within the function thumbee_notifier.

1. Set a Program Breakpoint to the address range of the function thumbee_notifier and select the trace action TraceEnable. Additionally select the process sieve in the TASK field.



2. Start and stop the program execution.

3. Display the result.



Function Run-Times Analysis

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

Software under Analysis (no OS, OS or OS+MMU)

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

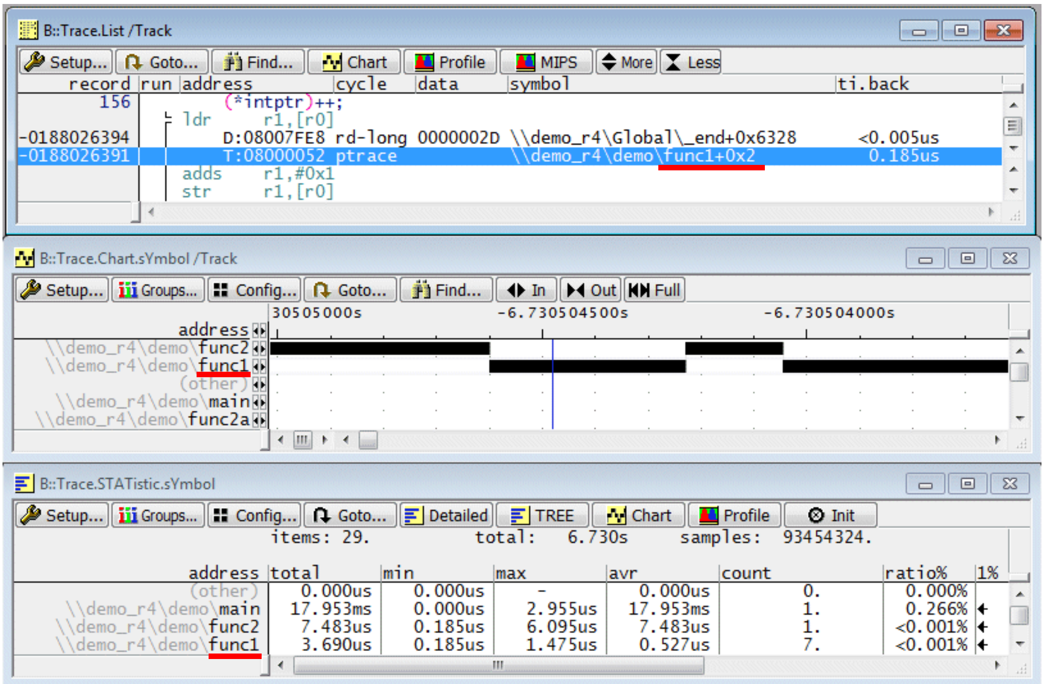
1. Software without operating system (abbreviation: **no OS**)
2. Software with an operating system without dynamic memory management (abbreviation: **OS**)
3. Software with an operating system that uses dynamic memory management to handle processes/tasks (abbreviation: **OS+MMU**). If an OS+MMU is used,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

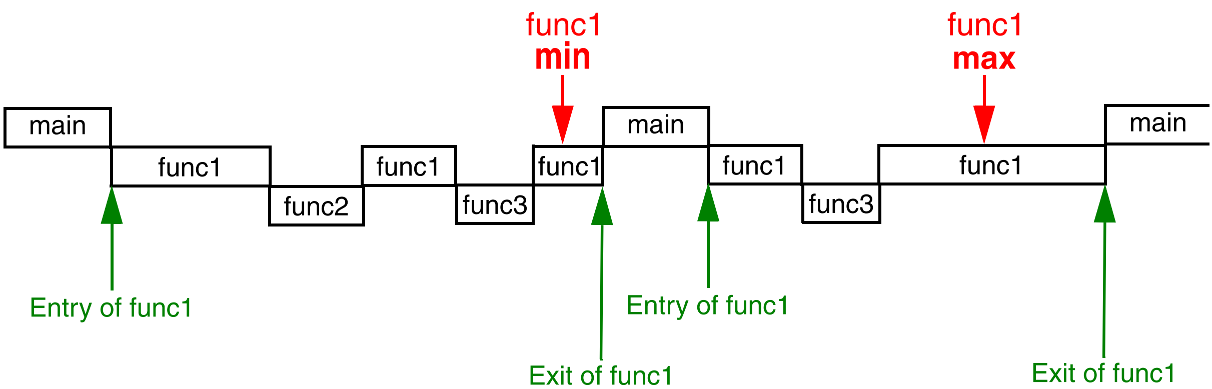
Flat vs. Nesting Analysis

TRACE32 provides two methods to analyze function run-times:

- Flat analysis
- Nesting analysis

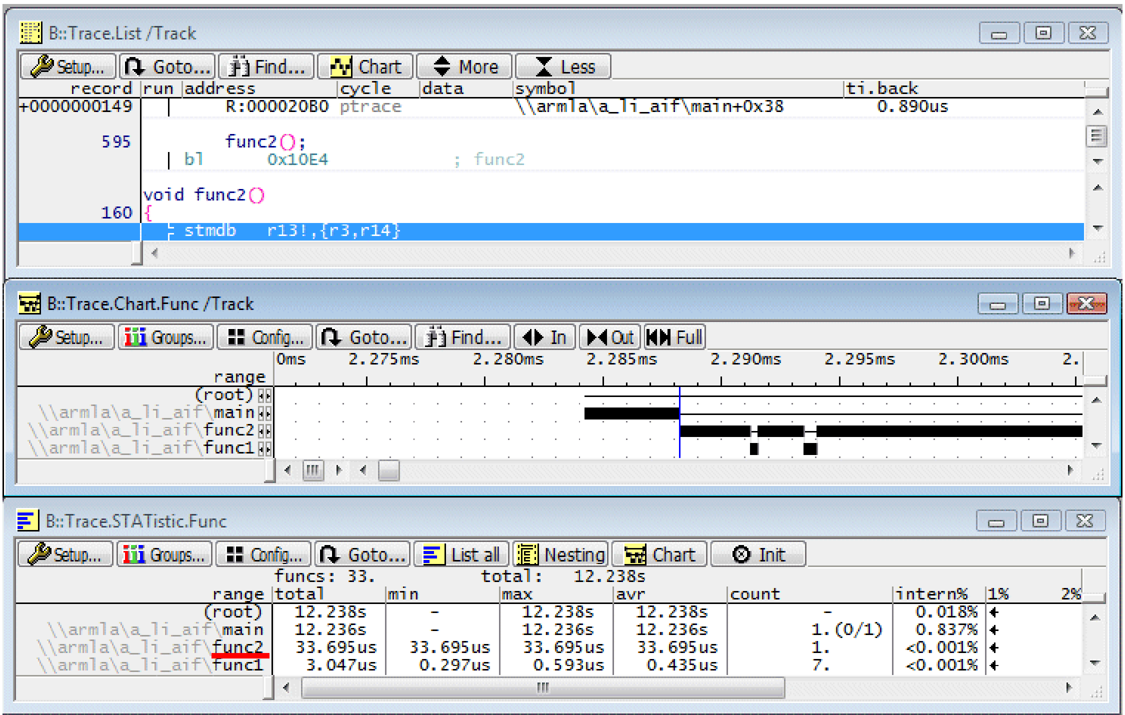
The flat function run-time analysis bases on the symbolic instruction addresses of the trace entries. The time spent by an instruction is assigned to the corresponding function/symbol region.

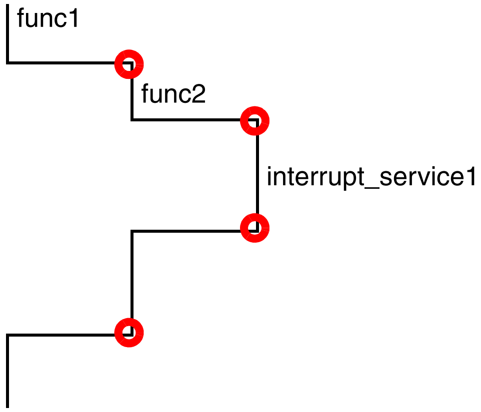




min	shortest time continuously in the address range of the function/symbol region
max	longest time continuously in the address range of the function/symbol region

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





In order to display a nested function run-time analysis TRACE32 analyzes the structure of the program execution by processing the trace information to find:

1. **Function entries**

2. **Function exits**

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

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

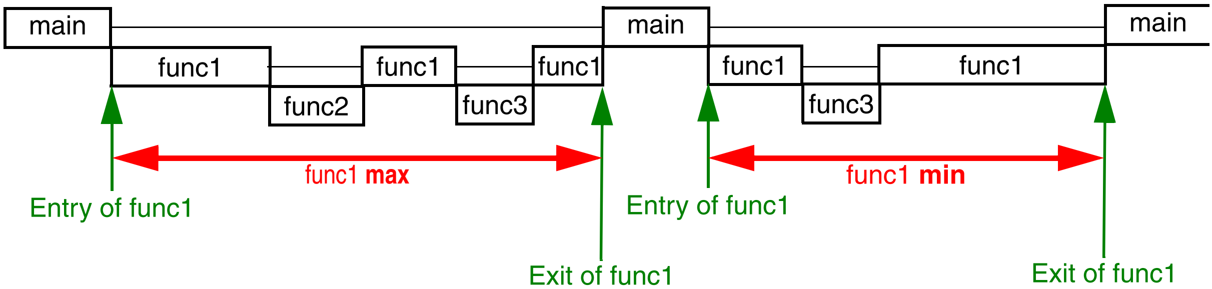
   The HLL function started following the interrupt is regarded as interrupt service routine.

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

4. **Exits of interrupt service routines**

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

6. **Exits of TRAP handlers**



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

Summary

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

Flat Analysis

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

- To avoid an overload of the ETM port.
- To make best use of the available trace memory.
- To get a more accurate timestamp (no-cycle accurate mode).

Optimum ETM Configuration (No OS or OS)

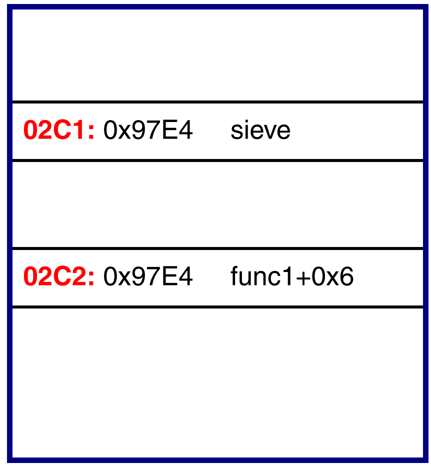
Flat function run-time analysis does **not** require any **data information** if no OS or an OS is used. That's why it is recommended to switch the broadcasting of data information off.

```
ETM.DataTrace off
```

Virtual address: 0x97E4

The virtual address exported by the ETM is not enough to indentify the function/ symbol range.

The Address Space ID is required!



TRACE32 Symbol Database

If an target operating system is used, that uses dynamic memory management to handle processes, the instruction flow plus information on the **Address Space ID** is required in order to perform a flat function run-time analysis.

The standard way to get the Address Space ID is to advise the ETM to export the instruction flow and the process switches. For details refer to the chapter **OS-Aware Tracing** of this training.

Optimum Configuration 1 (process switches are exported in form of a special write access)**:**

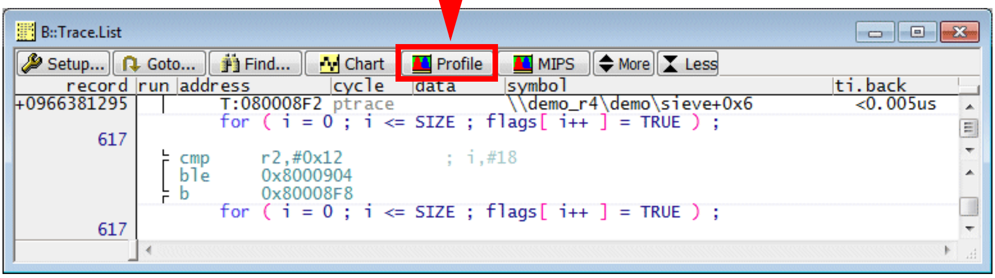
```
Break.Set TASK.CONFIG(magic) /Write /TraceData
```

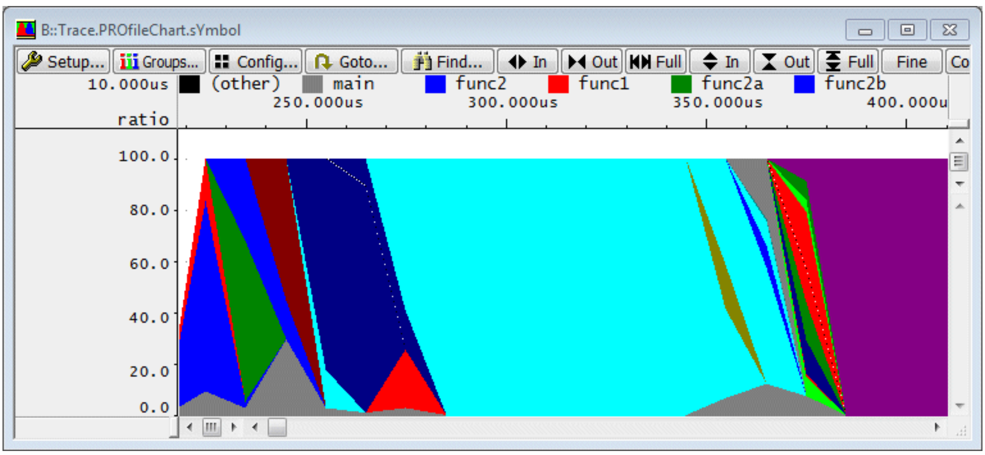
Optimum Configuration 2 (process switches are exported in form of a Context ID packet)**:**

```
ETM.ContextID 32
```

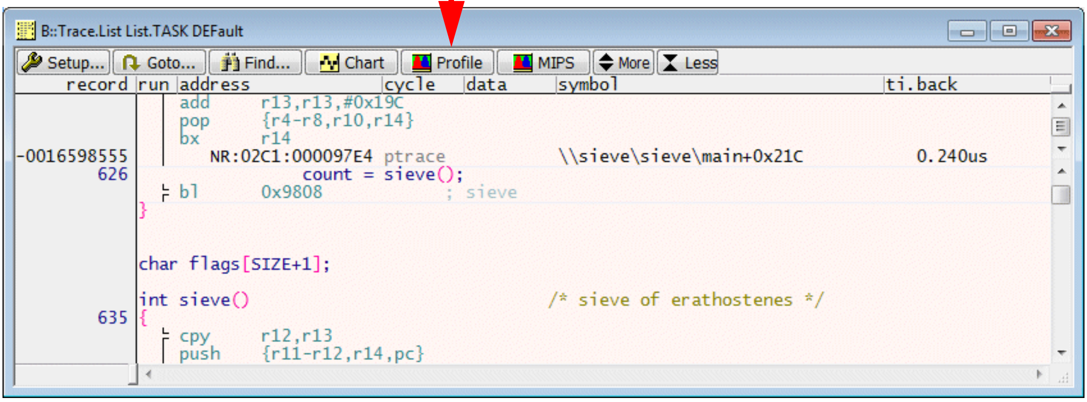
Look and Feel (No OS or OS)

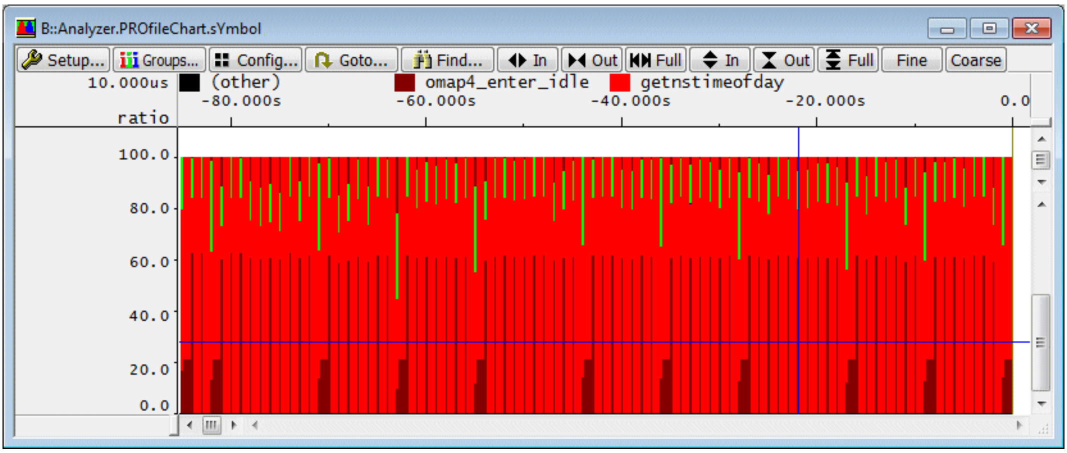
Push the **Profile** button to get information on the dynamic behaviour of the program

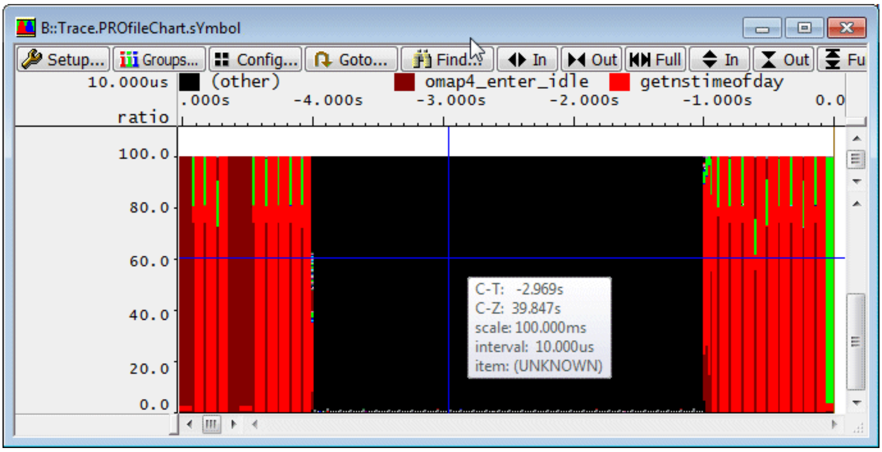




Push the **Profile** button to get information on the dynamic behaviour of the program



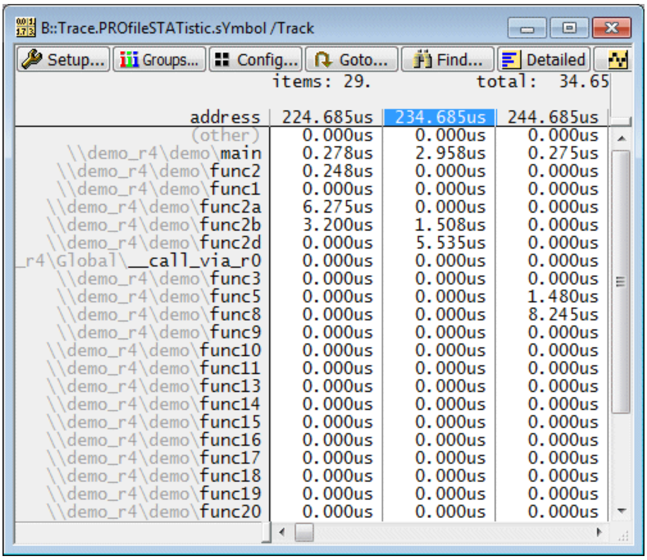




UNKNOW instructions are separately named in the **Trace.PROfileChart.sYmbol** analysis.

To draw the **Trace.PROfileChart.sYmbol** graphic, TRACE32 PowerView partition the recorded instruction flow information into time segments. The default segment size is 10.us.

For each time segment rectangles are draw that represent the time ratio the executed functions consumed within the time segment. For the final display this basic graph is smoothed.

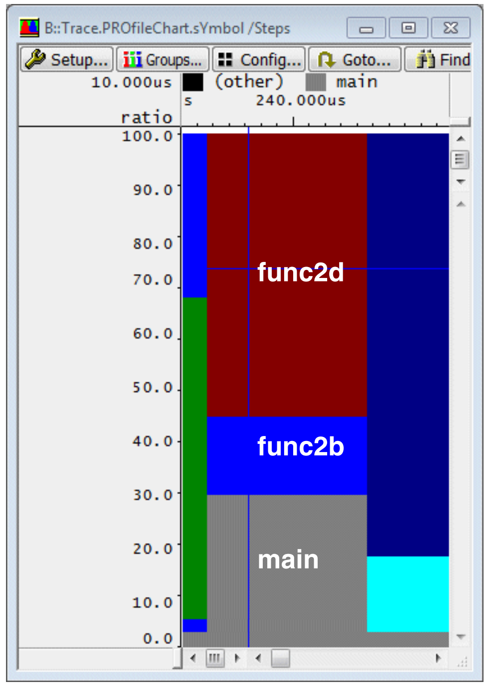


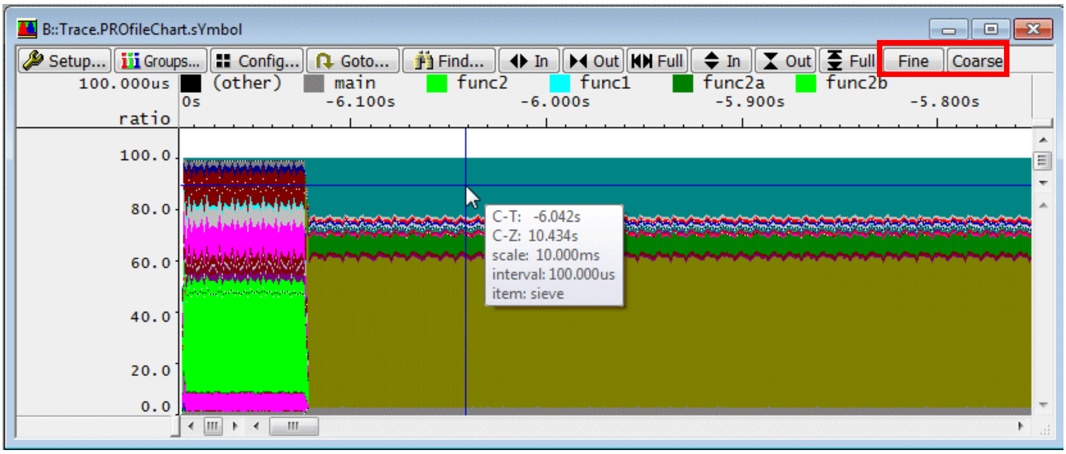
B::Trace.PROfileSTATistic.sYmbol /Track

Setup... Groups... Config... Goto... Find... Detailed

items: 29. total: 34.65

address	224.685us	234.685us	244.685us
(other)	0.000us	0.000us	0.000us
\\demo_r4\demo\main	0.278us	2.958us	0.275us
\\demo_r4\demo\func2	0.248us	0.000us	0.000us
\\demo_r4\demo\func1	0.000us	0.000us	0.000us
\\demo_r4\demo\func2a	6.275us	0.000us	0.000us
\\demo_r4\demo\func2b	3.200us	1.508us	0.000us
\\demo_r4\demo\func2d	0.000us	5.535us	0.000us
_r4\Global__call_via_r0	0.000us	0.000us	0.000us
\\demo_r4\demo\func3	0.000us	0.000us	0.000us
\\demo_r4\demo\func5	0.000us	0.000us	1.480us
\\demo_r4\demo\func8	0.000us	0.000us	8.245us
\\demo_r4\demo\func9	0.000us	0.000us	0.000us
\\demo_r4\demo\func10	0.000us	0.000us	0.000us
\\demo_r4\demo\func11	0.000us	0.000us	0.000us
\\demo_r4\demo\func13	0.000us	0.000us	0.000us
\\demo_r4\demo\func14	0.000us	0.000us	0.000us
\\demo_r4\demo\func15	0.000us	0.000us	0.000us
\\demo_r4\demo\func16	0.000us	0.000us	0.000us
\\demo_r4\demo\func17	0.000us	0.000us	0.000us
\\demo_r4\demo\func18	0.000us	0.000us	0.000us
\\demo_r4\demo\func19	0.000us	0.000us	0.000us
\\demo_r4\demo\func20	0.000us	0.000us	0.000us





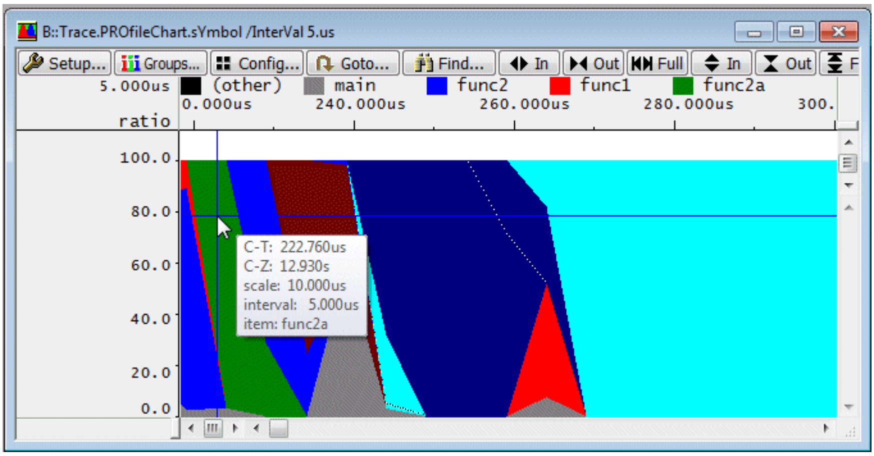
Fine	Decrease the time segment size by the factor 10
Coarse	Increase the time segment size by the factor 10

The time segment size can also be set manually.

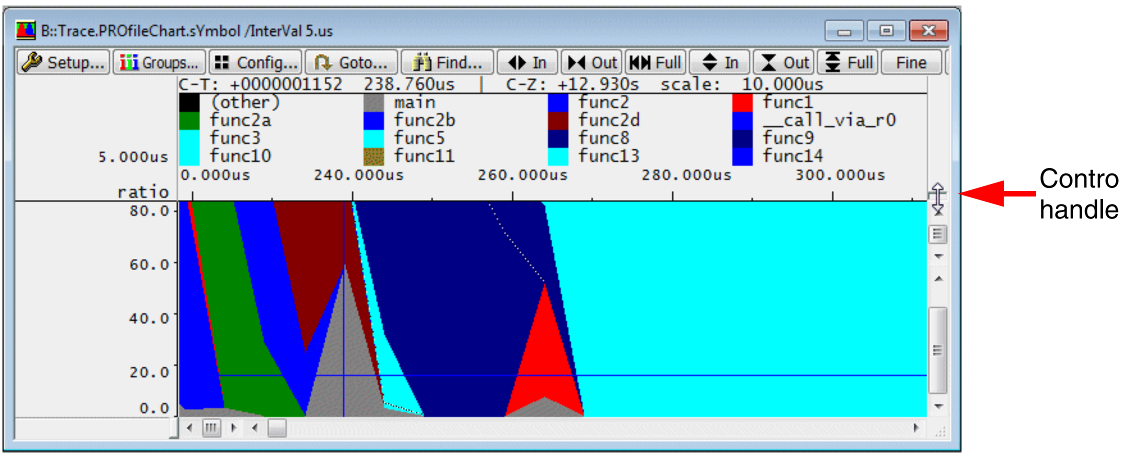
```
Trace.PROfileChart.sYmbol /InterVal 5.ms          ; change the time
                                                  ; segment size to 5.ms
```

Color Assignment - Basics

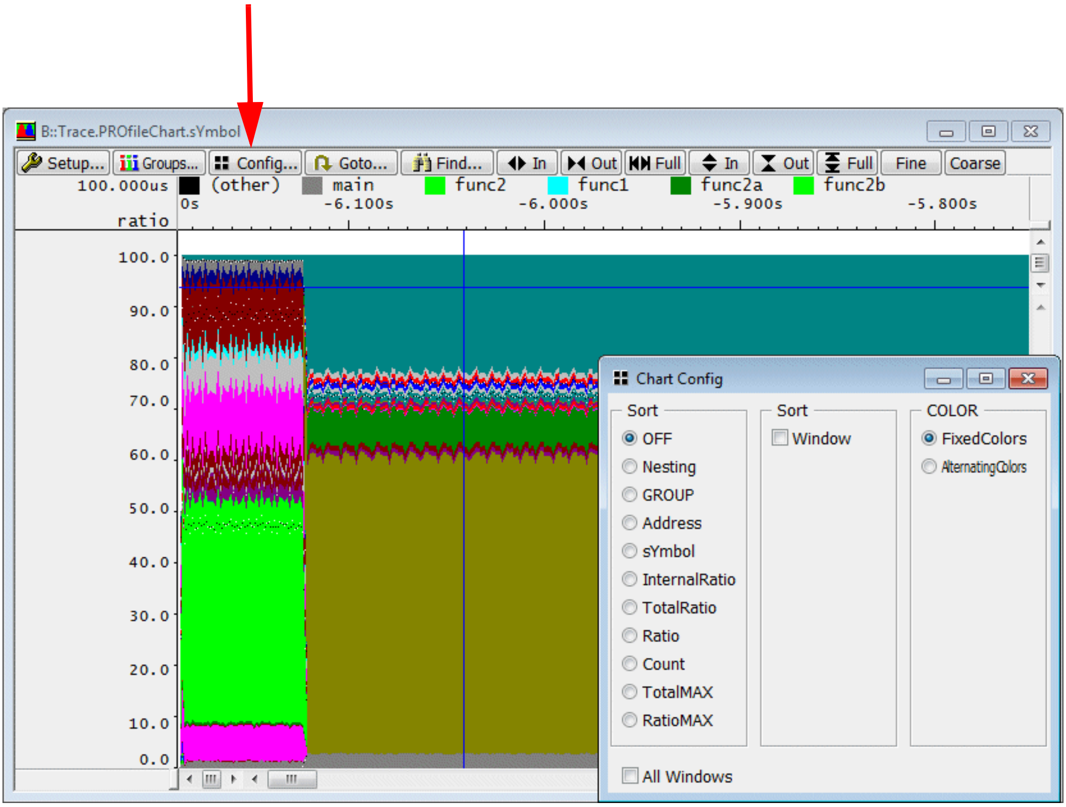
- The tooltip at the cursor position shows the color assignment and the used segment size (InterVal).



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



Color Assignment - Statically or Dynamically



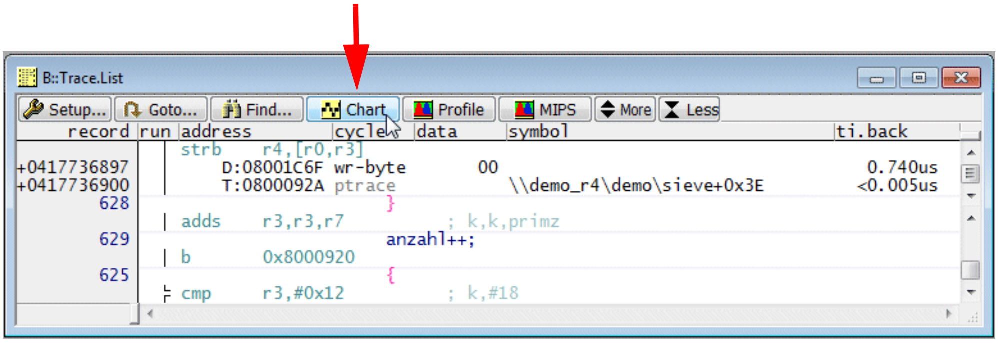
FixedColors	Colors are assigned fixed to functions (default). Fixed color assignment has the risk that two functions with the same color are drawn side by side and thus may convey a wrong impression of the dynamic behavior.
AlternatingColors	Colors are assigned by the recording order of the functions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

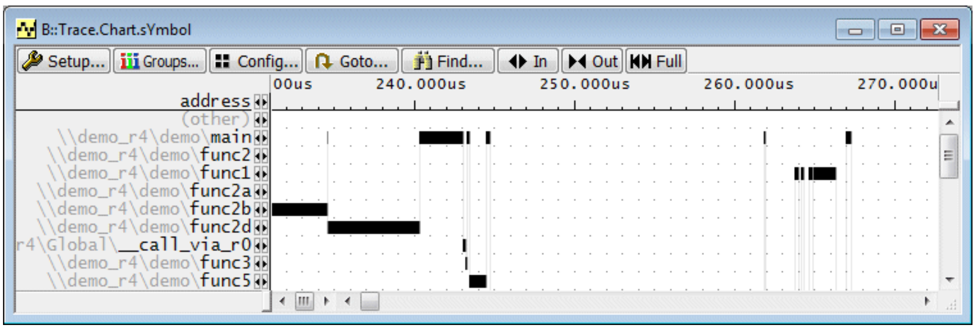
Trace.PROfileChart.sYmbol [/InterVal *<time>*]	Overview on the dynamic behavior of the program - graphical display
Trace.PROfileSTATistic.sYmbol [/InterVal *<time>*]	Overview on the dynamic behavior of the program - numerical display for export as comma-separated values
Trace.STATistic.COLOR FixedColors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

Look and Feel (No OS or OS)

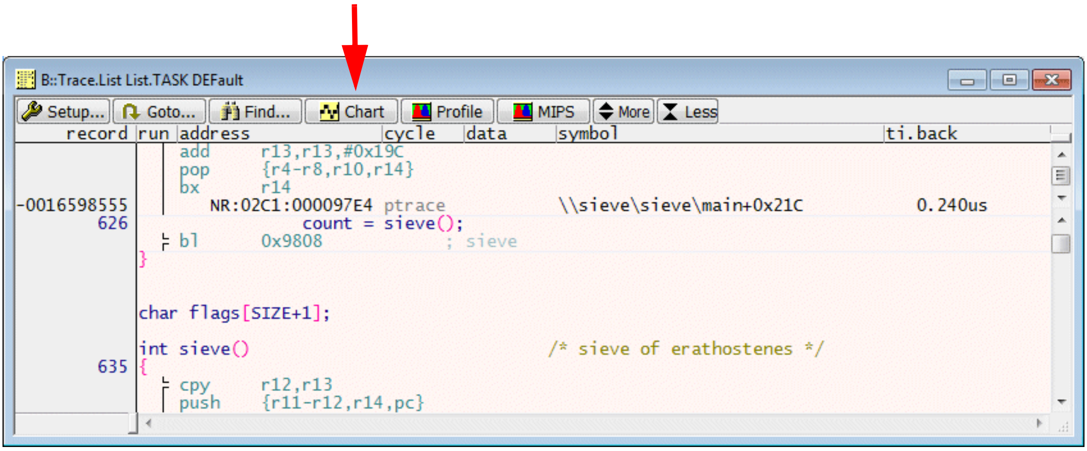
TRACE32 PowerView provides a timing diagram which shows when the program counters was in which function/symbol range.

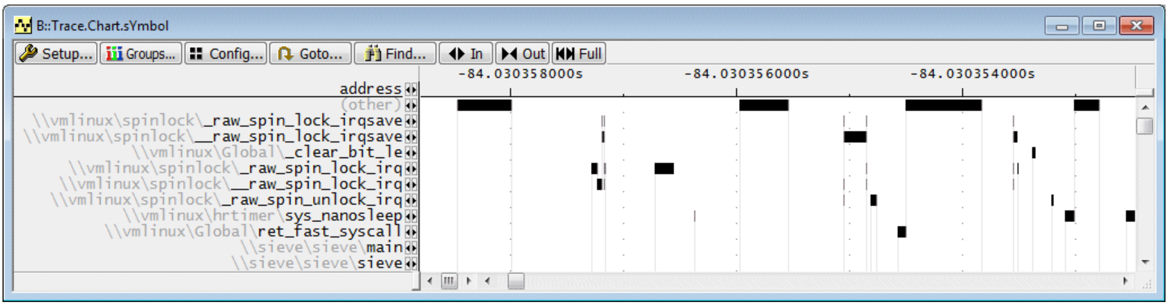
Pushing the **Chart** button in the **Trace.List** window opens a **Trace.Chart.sYmbol** window

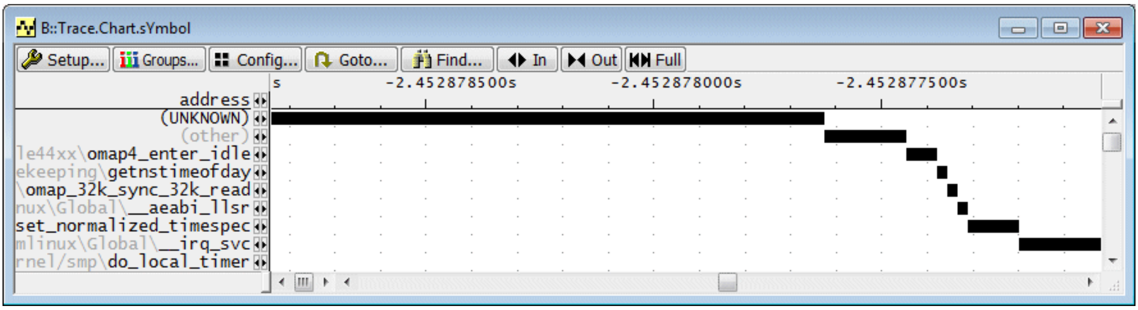


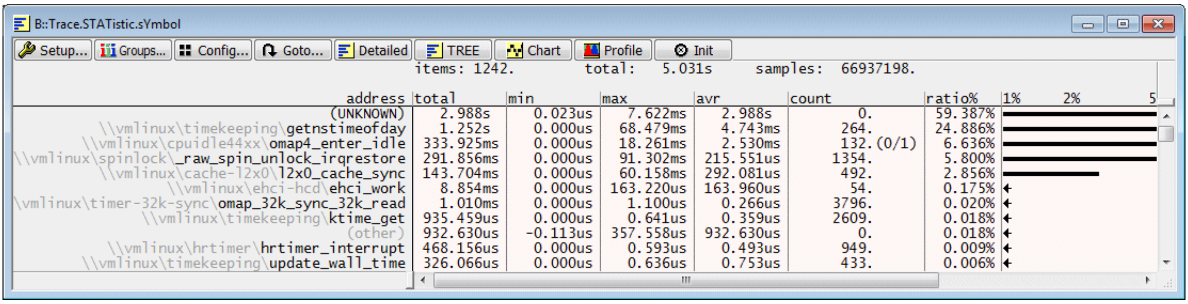


Pushing the **Chart** button in the **Trace.List** window opens a **Trace.Chart.sYmbol** window



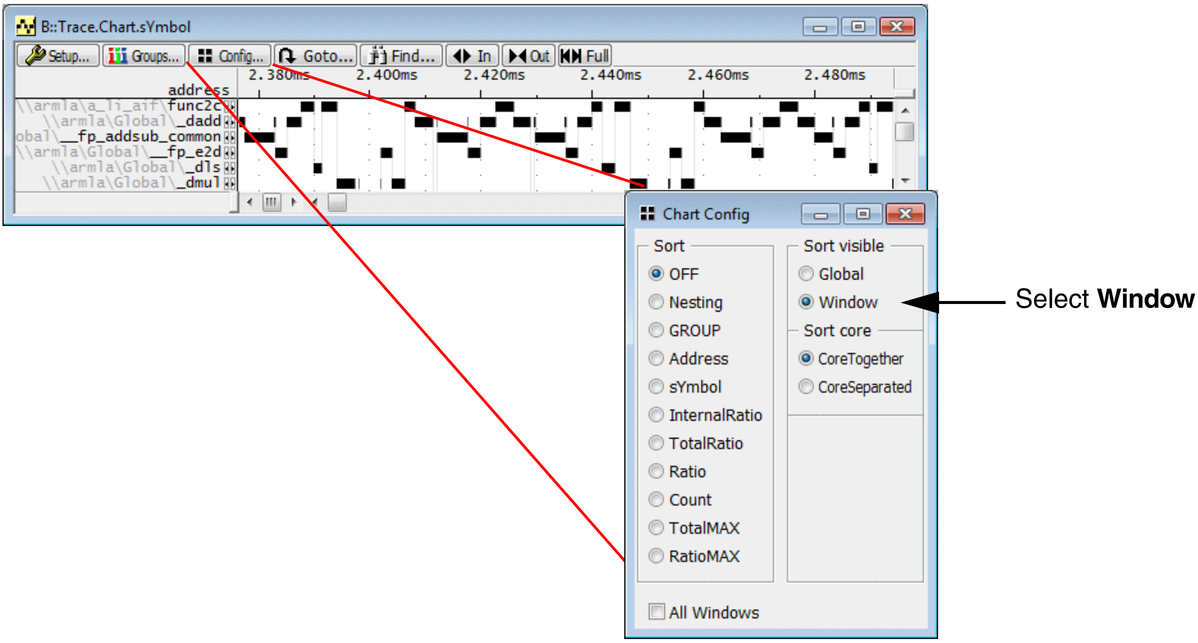






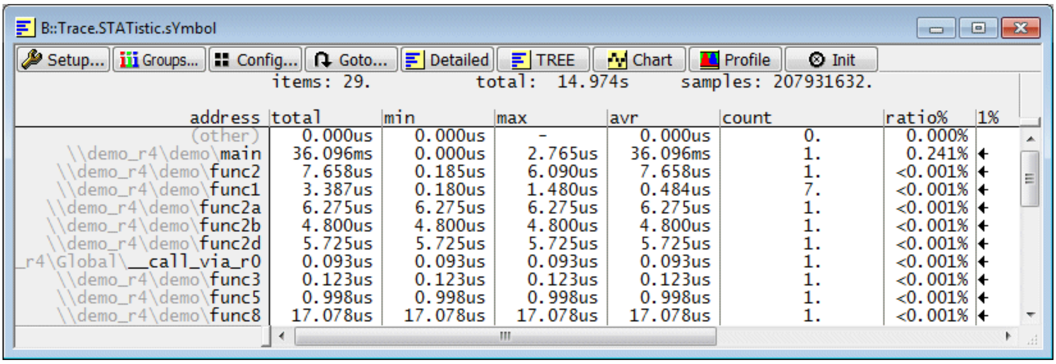
address	total	min	max	avr	count	ratio%
(UNKNOWN)	2.988s	0.023us	7.622ms	2.988s	0.	59.387%
\\vmlinux\timekeeping\getnstimeofday	1.252s	0.000us	68.479ms	4.743ms	264.	24.886%
\\vmlinux\cpuidle44xx\omap4_enter_idle	333.925ms	0.000us	18.261ms	2.530ms	132. (0/1)	6.636%
\\vmlinux\spinlock_raw_spin_unlock_irqrestore	291.856ms	0.000us	91.302ms	215.551us	1354.	5.800%
\\vmlinux\cache-l2x0\l2x0_cache_sync	143.704ms	0.000us	60.158ms	292.081us	492.	2.856%
\\vmlinux\ehci-hcd\ehci_work	8.854ms	0.000us	163.220us	163.960us	54.	0.175%
\vmlinux\timer-32k-sync\omap_32k_sync_32k_read	1.010ms	0.000us	1.100us	0.266us	3796.	0.020%
\\vmlinux\timekeeping\ktime_get	935.459us	0.000us	0.641us	0.359us	2609.	0.018%
(other)	932.630us	-0.113us	357.558us	932.630us	0.	0.018%
\\vmlinux\hrtimer\hrtimer_interrupt	468.156us	0.000us	0.593us	0.493us	949.	0.009%
\\vmlinux\timekeeping\update_wall_time	326.066us	0.000us	0.636us	0.753us	433.	0.006%

(UNKNOWN) instructions are separatly listed in the analysis.



If Sort **visible/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.

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



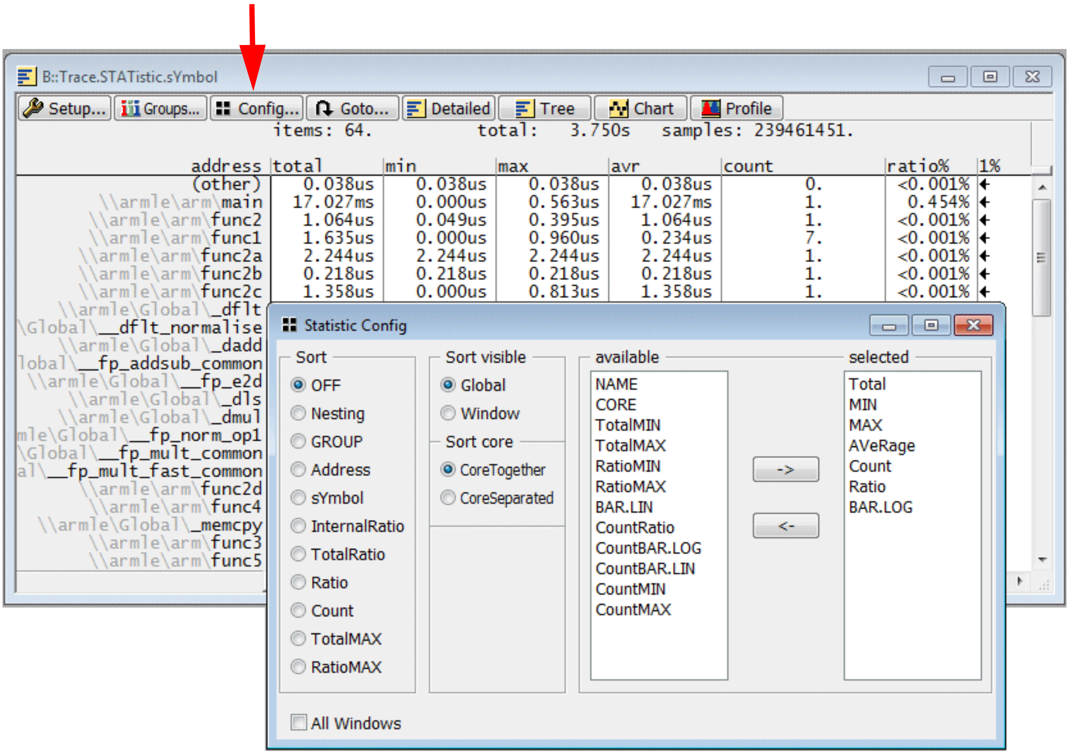
address	total	min	max	avr	count	ratio%
(other)	0.000us	0.000us	-	0.000us	0.	0.000%
\\demo_r4\demo\main	36.096ms	0.000us	2.765us	36.096ms	1.	0.241%
\\demo_r4\demo\func2	7.658us	0.185us	6.090us	7.658us	1.	<0.001%
\\demo_r4\demo\func1	3.387us	0.180us	1.480us	0.484us	7.	<0.001%
\\demo_r4\demo\func2a	6.275us	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)

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

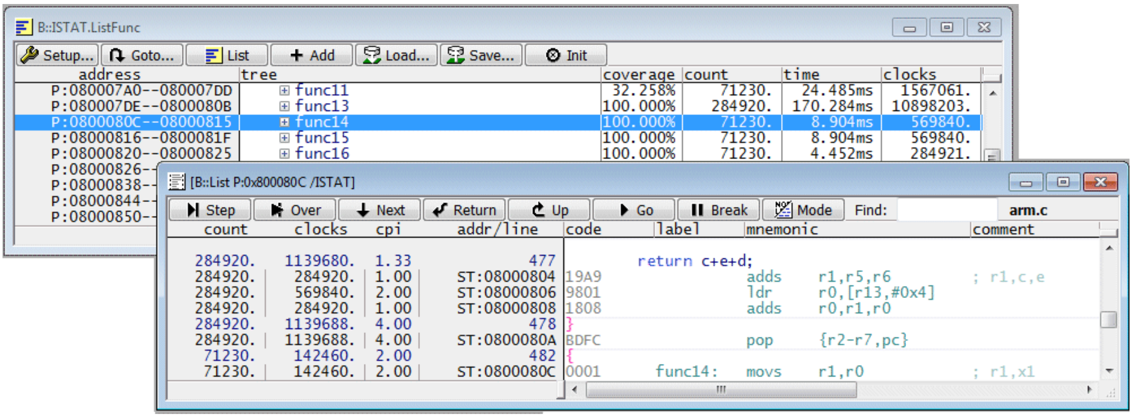
count	number of new entries (start address executed) into the address range of the function/symbol region
ratio	ratio of time in the function/symbol region with regards to the total time period recorded

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 - numerical display
Trace.Chart.sYmbol	Flat function run-time analysis - graphical display

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



Preparation

Constant clock while recording

```
ETM.TImeMode CycleAccurate              ; select cycle-accurate tracing

Trace.CLOCK 600.MHz                     ; inform TRACE32 about your
                                        ; CPU/core frequency
```

Changing clock while recording

```
; combine cycle accurate tracing and TRACE32 external timestamp
ETM.TImeMode CycleAccurate+ExternalTrack
```

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

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

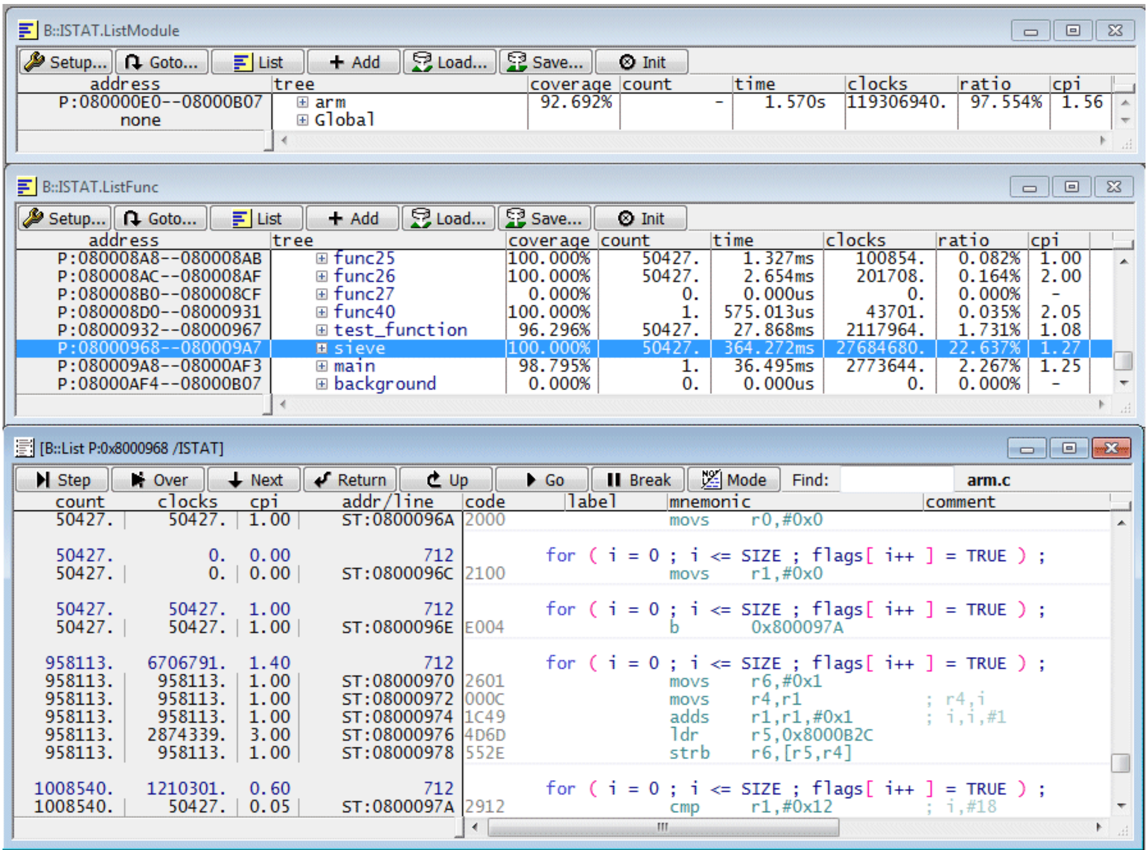
1. Enable cycle-accurate tracing.
2. Specify the core/CPU clock.
3. Clear the database.
4. Fill the trace memory.
5. Transfer the contents of the trace memory to the database.
6. Display the result.
7. (Repeat step 4-6 if required)

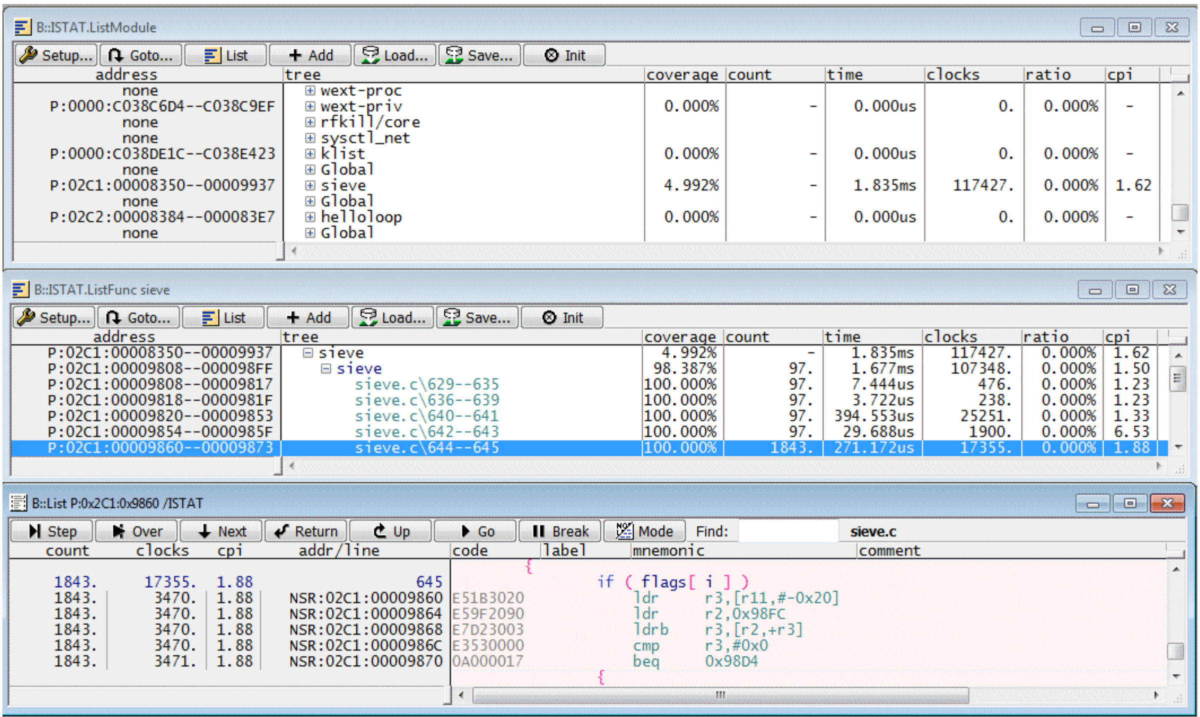
The following commands are available:

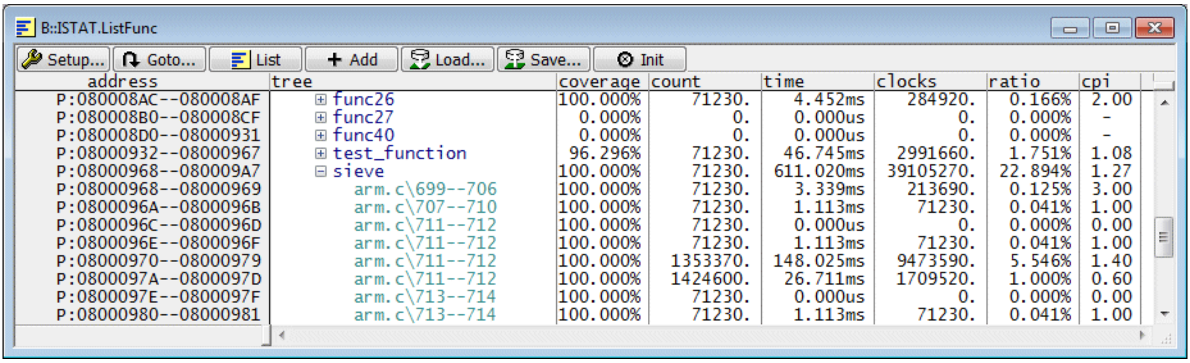
Trace.CLOCK *<clock>*	Specify the core/CPU clock for the trace evaluation.
ISTATistic.RESet	Clear the instruction statistic database.
ISTATistic.add	Add the contents of the trace memory to the instruction statistic database.
ISTATistic.ListFunc	List function run-time analysis based on the contents of the instruction statistic database.
List *<address>* **/ISTAT**	List run-time analysis for the single instructions.

A detailed function run-time analysis can be performed as follows (ARM11 with ETMv3 as example):

```
                                          ; ETM setup

ETM.CycleAccurate ON                      ; switch cycle accurate tracing on

...                                       ; general procedure

Trace.CLOCK 176.MHz                       ; inform TRACE32 about your CPU
                                          ; frequency

ISTATistic.RESet                          ; reset instruction statistic data
                                          ; base

Trace.Mode Leash                          ; switch trace to Leash mode

Go                                        ; start program execution

;WAIT !RUN()                              ; wait until program stops

Trace.FOWPROCESS                          ; upload the trace information to
                                          ; the host and merge source code

IF Analyzer.FLOW.FIFOFULL()>6000.
     PRINT "Warning: Please control the FIFOFULLS"

ISTATistic.ADD                            ; add trace information to
                                          ; instruction statistic data
                                          ; base

ISTATistic.ListFunc                       ; list hot-spot analysis
```


B::ISTAT.ListModule
Setup... Goto... List Add Load... Save... Init
address tree coverage count time clocks ratio cpi
P:080000E0--08000B07 arm 92.692% - 1.570s 119306940. 97.554% 1.56
none Global
B::ISTAT.ListFunc
Setup... Goto... List Add Load... Save... Init
address tree coverage count time clocks ratio cpi
P:080008A8--080008AB func25 100.000% 50427. 1.327ms 100854. 0.082% 1.00
P:080008AC--080008AF func26 100.000% 50427. 2.654ms 201708. 0.164% 2.00
P:080008B0--080008CF func27 0.000% 0. 0.000us 0. 0.000% -
P:080008D0--08000931 func40 100.000% 1. 575.013us 43701. 0.035% 2.05
P:08000932--08000967 test_function 96.296% 50427. 27.868ms 2117964. 1.731% 1.08
P:08000968--080009A7 sieve 100.000% 50427. 364.272ms 27684680. 22.637% 1.27
P:080009A8--08000AF3 main 98.795% 1. 36.495ms 2773644. 2.267% 1.25
P:08000AF4--08000B07 background 0.000% 0. 0.000us 0. 0.000% -
[B::List P:0x8000968 /ISTAT]
Step Over Next Return Up Go Break Mode Find: arm.c
count clocks cpi addr/line code label mnemonic comment
50427. 50427. 1.00 ST:0800096A 2000 movs r0,#0x0
50427. 0. 0.00 712 for (i = 0 ; i <= SIZE ; flags[i++] = TRUE) ;
50427. 0. 0.00 ST:0800096C 2100 movs r1,#0x0
50427. 50427. 1.00 712 for (i = 0 ; i <= SIZE ; flags[i++] = TRUE) ;
50427. 50427. 1.00 ST:0800096E E004 b 0x800097A
958113. 6706791. 1.40 712 for (i = 0 ; i <= SIZE ; flags[i++] = TRUE) ;
958113. 958113. 1.00 ST:08000970 2601 movs r6,#0x1
958113. 958113. 1.00 ST:08000972 000C movs r4,r1 ; r4,i
958113. 958113. 1.00 ST:08000974 1C49 adds r1,r1,#0x1 ; i,i,#1
958113. 2874339. 3.00 ST:08000976 4D6D ldr r5,0x8000B2C
958113. 958113. 1.00 ST:08000978 552E strb r6,[r5,r4]
1008540. 1210301. 0.60 712 for (i = 0 ; i <= SIZE ; flags[i++] = TRUE) ;
1008540. 50427. 0.05 ST:0800097A 2912 cmp r1,#0x12 ; i,#18

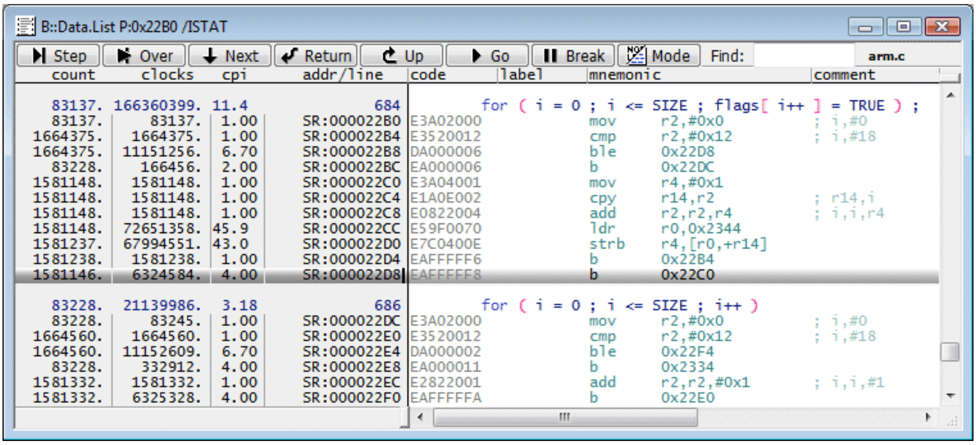

B::ISTAT.ListModule
Setup... Goto... List Add Load... Save... Init
address tree coverage count time clocks ratio cpi
none wext-proc
P:0000:C038C6D4--C038C9EF wext-priv 0.000% - 0.000us 0. 0.000% -
none rfkill/core
none sysctl_net
P:0000:C038DE1C--C038E423 klist 0.000% - 0.000us 0. 0.000% -
none Global
P:02C1:00008350--00009937 sieve 4.992% - 1.835ms 117427. 0.000% 1.62
none Global
P:02C2:00008384--000083E7 helloloop 0.000% - 0.000us 0. 0.000% -
none Global
B::ISTAT.ListFunc sieve
Setup... Goto... List Add Load... Save... Init
address tree coverage count time clocks ratio cpi
P:02C1:00008350--00009937 sieve 4.992% - 1.835ms 117427. 0.000% 1.62
P:02C1:00009808--000098FF sieve 98.387% 97. 1.677ms 107348. 0.000% 1.50
P:02C1:00009808--00009817 sieve.c\629--635 100.000% 97. 7.444us 476. 0.000% 1.23
P:02C1:00009818--0000981F sieve.c\636--639 100.000% 97. 3.722us 238. 0.000% 1.23
P:02C1:00009820--00009853 sieve.c\640--641 100.000% 97. 394.553us 25251. 0.000% 1.33
P:02C1:00009854--0000985F sieve.c\642--643 100.000% 97. 29.688us 1900. 0.000% 6.53
P:02C1:00009860--00009873 sieve.c\644--645 100.000% 1843. 271.172us 17355. 0.000% 1.88
B::List P:0x2C1:0x9860 /ISTAT
Step Over Next Return Up Go Break Mode Find: sieve.c
count clocks cpi addr/line code label mnemonic comment
{
1843. 17355. 1.88 645 if (flags[i])
1843. 3470. 1.88 NSR:02C1:00009860 E51B3020 ldr r3,[r11,#-0x20]
1843. 3470. 1.88 NSR:02C1:00009864 E59F2090 ldr r2,0x98FC
1843. 3470. 1.88 NSR:02C1:00009868 E7D23003 ldrb r3,[r2,+r3]
1843. 3470. 1.88 NSR:02C1:0000986C E3530000 cmp r3,#0x0
1843. 3471. 1.88 NSR:02C1:00009870 0A000017 beq 0x98D4
{



address	tree	coverage	count	time	clocks	ratio	cpi
P:080008AC--080008AF	⊞ func26	100.000%	71230.	4.452ms	284920.	0.166%	2.00
P:080008B0--080008CF	⊞ func27	0.000%	0.	0.000us	0.	0.000%	-
P:080008D0--08000931	⊞ func40	0.000%	0.	0.000us	0.	0.000%	-
P:08000932--08000967	⊞ test_function	96.296%	71230.	46.745ms	2991660.	1.751%	1.08
P:08000968--080009A7	⊟ sieve	100.000%	71230.	611.020ms	39105270.	22.894%	1.27
P:08000968--08000969	arm.c\699--706	100.000%	71230.	3.339ms	213690.	0.125%	3.00
P:0800096A--0800096B	arm.c\707--710	100.000%	71230.	1.113ms	71230.	0.041%	1.00
P:0800096C--0800096D	arm.c\711--712	100.000%	71230.	0.000us	0.	0.000%	0.00
P:0800096E--0800096F	arm.c\711--712	100.000%	71230.	1.113ms	71230.	0.041%	1.00
P:08000970--08000979	arm.c\711--712	100.000%	1353370.	148.025ms	9473590.	5.546%	1.40
P:0800097A--0800097D	arm.c\711--712	100.000%	1424600.	26.711ms	1709520.	1.000%	0.60
P:0800097E--0800097F	arm.c\713--714	100.000%	71230.	0.000us	0.	0.000%	0.00
P:08000980--08000981	arm.c\713--714	100.000%	71230.	1.113ms	71230.	0.041%	1.00

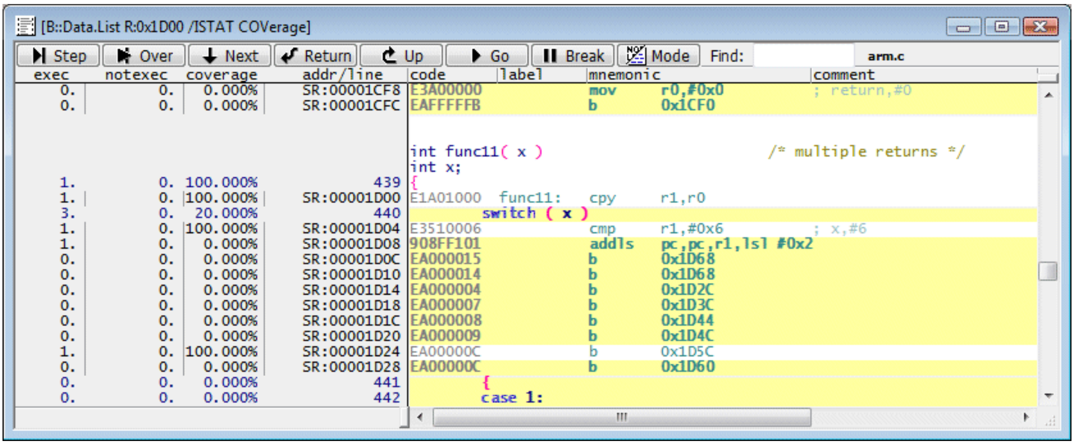
address	address range of the module, function or HLL line
tree	flat module/function/HLL line tree
coverage	code coverage of the module, function or HLL line
count	number of function/HLL line executions
time	total time spent by the module, function or HLL line
clocks	total number of clocks spent by the module, function or HLL line
ratio	Percentage of the total measurement time spent in the module, function or HLL line
cpi	average clocks per instruction for the function or the HLL line

```
List /ISTAT                                    ; list instruction run-time
                                               ; statistic
```



count	total number of instruction executions
clocks	total number of clocks for the instruction
cpi	average clocks per instruction

```
List /ISTAT COVerage                              ; list instruction coverage
```



exec	**conditional instructions:** number of times the instruction was executed because the condition was true. **other instructions:** number of times the instruction was executed
notexec	**conditional instructions:** number of times the instruction wasn't executed because the condition was false.
coverage	instruction coverage

Instructions with a condition are bold-printed on a yellow background if exec or notexec is 0 (or both).
Instructions without a condition are bold-printed on a yellow background if exec is 0.

Nesting Analysis

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.**

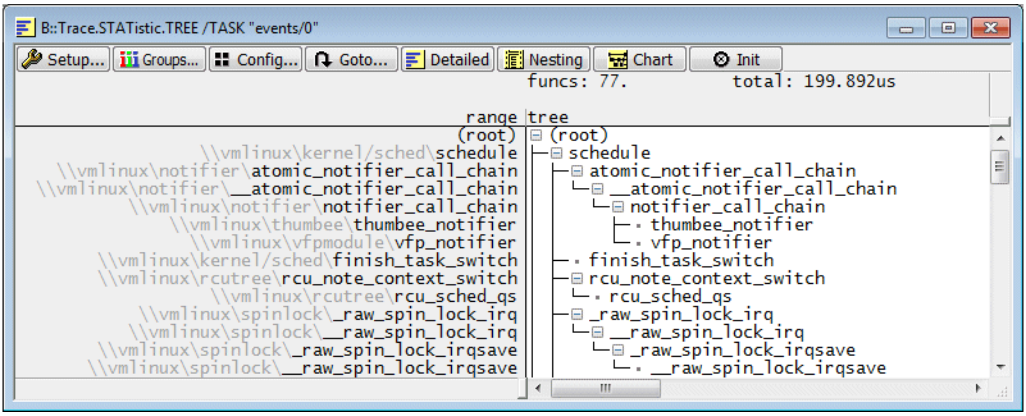
Optimum ETM Configuration (No OS)

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.

```
ETM.DataTrace off                          ; ARM-ETM
```

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

```
Trace.STATistic.TREE /TASK "events/0"
```



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

The standard way to get information on the current task/process/thread is to advise the ETM to export the instruction flow and task/process/thread switches. For details refer to the chapter **OS-Aware Tracing** of this training.

Optimum Configuration 1 (process switches are exported in form of a special write access)**:**

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

Optimum Configuration 2 (process switches are exported in form of a Context ID packet)**:**

```
ETM.ContextID 32
```

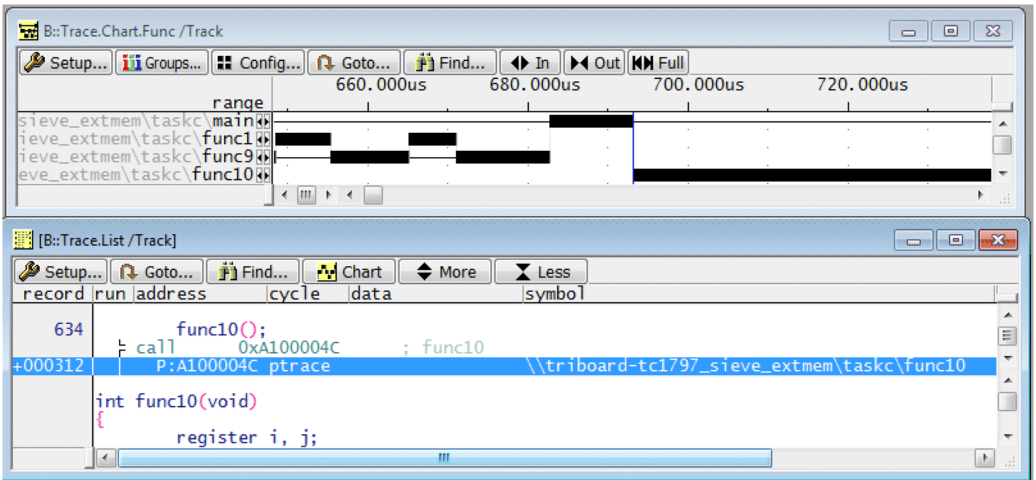
In order to prepare the results for the nesting analysis TRACE32 postprocesses the instruction flow to find:

- **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.

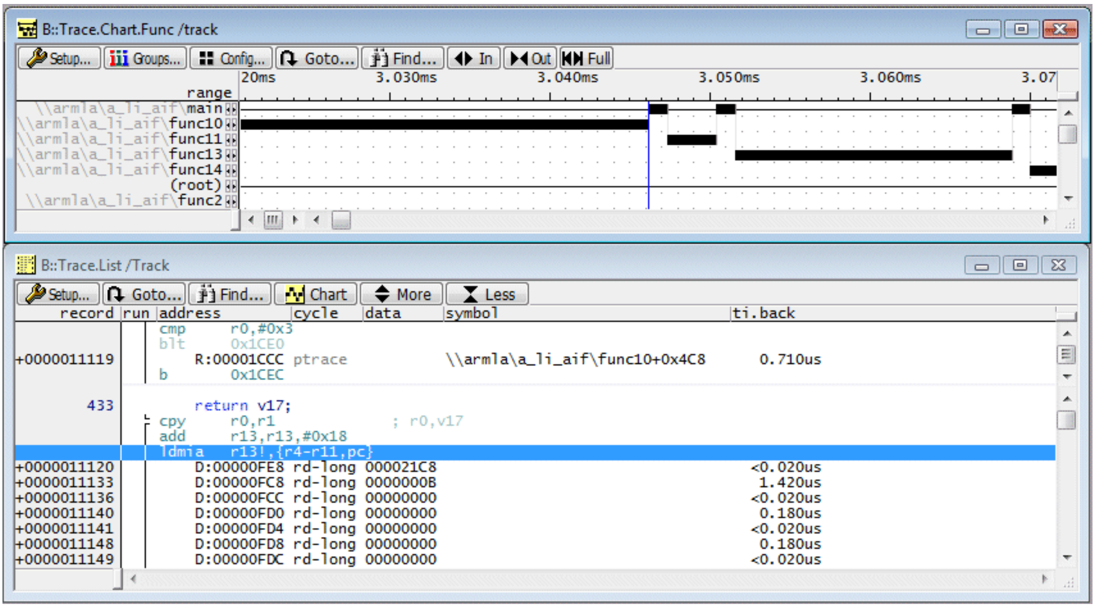
```
Trace.Chart.Func                          ; function func10 as
                                          ; example

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)**

  Interrupts are identified if an entry to the vector table is detected and the vector address indicates an asynchronous/hardware interrupt

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

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

- **Exits of interrupt service routines**

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

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

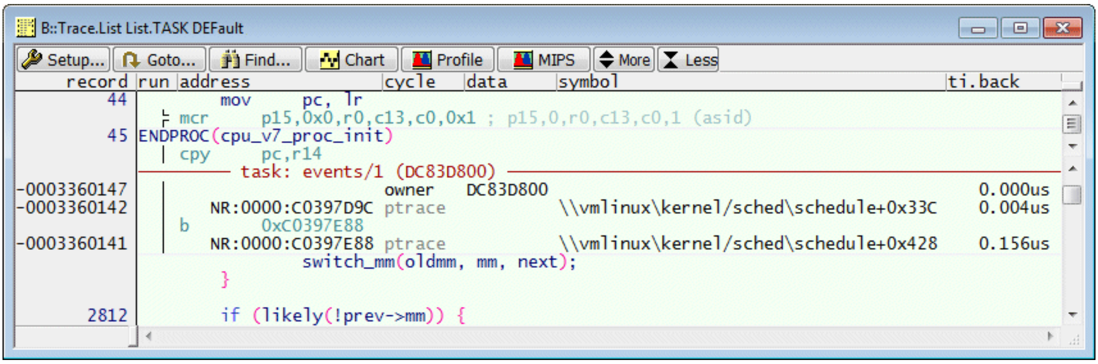
  If an entry to the vector table was identified and if the vector address indicates a synchronous interrupt/trap the following entry to an HLL function is regarded as entry to the trap handler.

- **Exits of TRAP handlers**

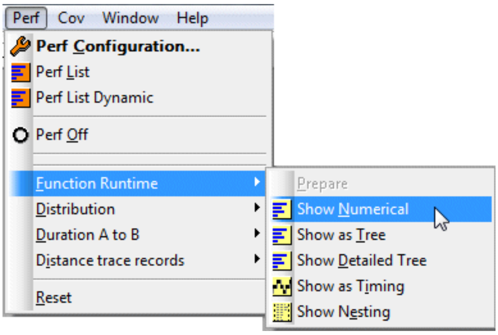
  A RETURN / RETURN FROM INTERRUPT within the HLL trap handler is regarded as exit of the TRAP handler.

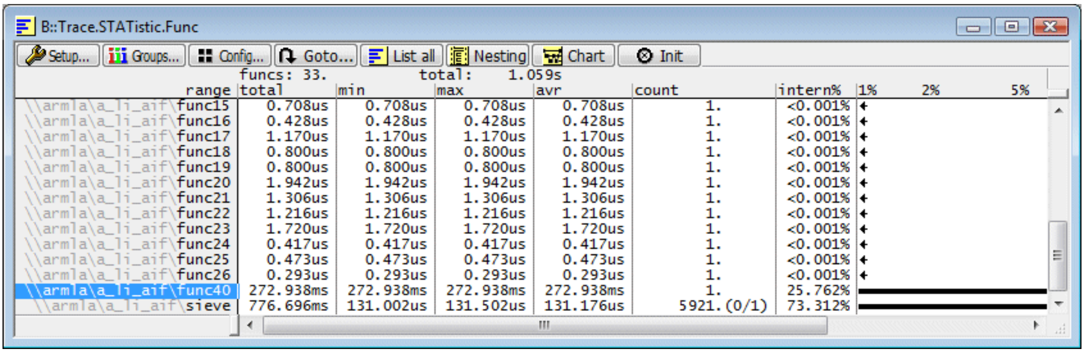
- **Task/process/thread switches**

  Task/process/thread 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






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

survey	
func	number of functions in the trace
total	total measurement time
intr	total time in interrupt service routines

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

- **HLL function**

- **(root)**

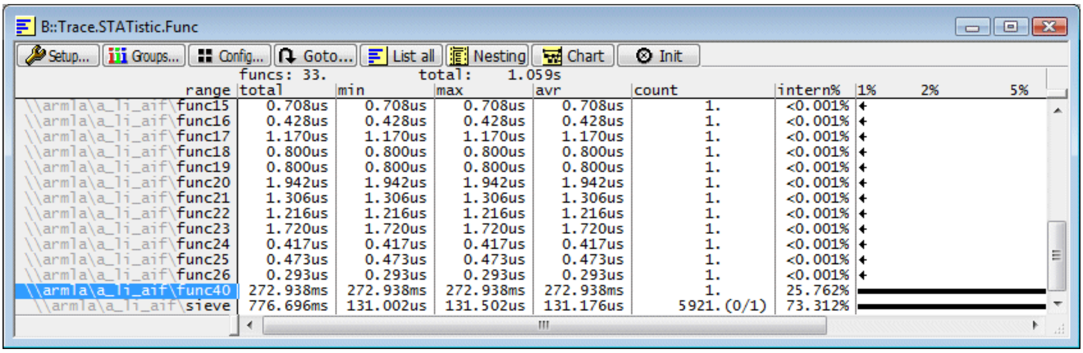
  The function nesting is regarded as tree, root is the root of the function nesting.

- **HLL interrupt service routine**

- **HLL trap handler**

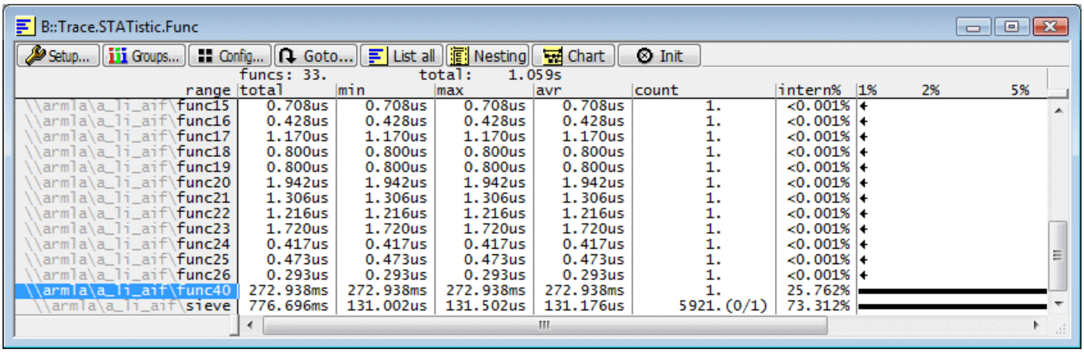

B::Trace.STATistic.Func

Setup... | Groups... | Config... | Goto... | List all | Nesting | Chart | Init

funcs: 33. total: 1.059s

range	total	min	max	avr	count	intern%	1% 2% 5%
\\armla\a_li_aif\func15	0.708us	0.708us	0.708us	0.708us	1.	<0.001%	
\\armla\a_li_aif\func16	0.428us	0.428us	0.428us	0.428us	1.	<0.001%	
\\armla\a_li_aif\func17	1.170us	1.170us	1.170us	1.170us	1.	<0.001%	
\\armla\a_li_aif\func18	0.800us	0.800us	0.800us	0.800us	1.	<0.001%	
\\armla\a_li_aif\func19	0.800us	0.800us	0.800us	0.800us	1.	<0.001%	
\\armla\a_li_aif\func20	1.942us	1.942us	1.942us	1.942us	1.	<0.001%	
\\armla\a_li_aif\func21	1.306us	1.306us	1.306us	1.306us	1.	<0.001%	
\\armla\a_li_aif\func22	1.216us	1.216us	1.216us	1.216us	1.	<0.001%	
\\armla\a_li_aif\func23	1.720us	1.720us	1.720us	1.720us	1.	<0.001%	
\\armla\a_li_aif\func24	0.417us	0.417us	0.417us	0.417us	1.	<0.001%	
\\armla\a_li_aif\func25	0.473us	0.473us	0.473us	0.473us	1.	<0.001%	
\\armla\a_li_aif\func26	0.293us	0.293us	0.293us	0.293us	1.	<0.001%	
\\armla\a_li_aif\func40	272.938ms	272.938ms	272.938ms	272.938ms	1.	25.762%	
\\armla\a_li_aif\sieve	776.696ms	131.002us	131.502us	131.176us	5921.(0/1)	73.312%	

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



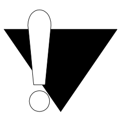
columns (cont.)	
count	number of times within the function

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:

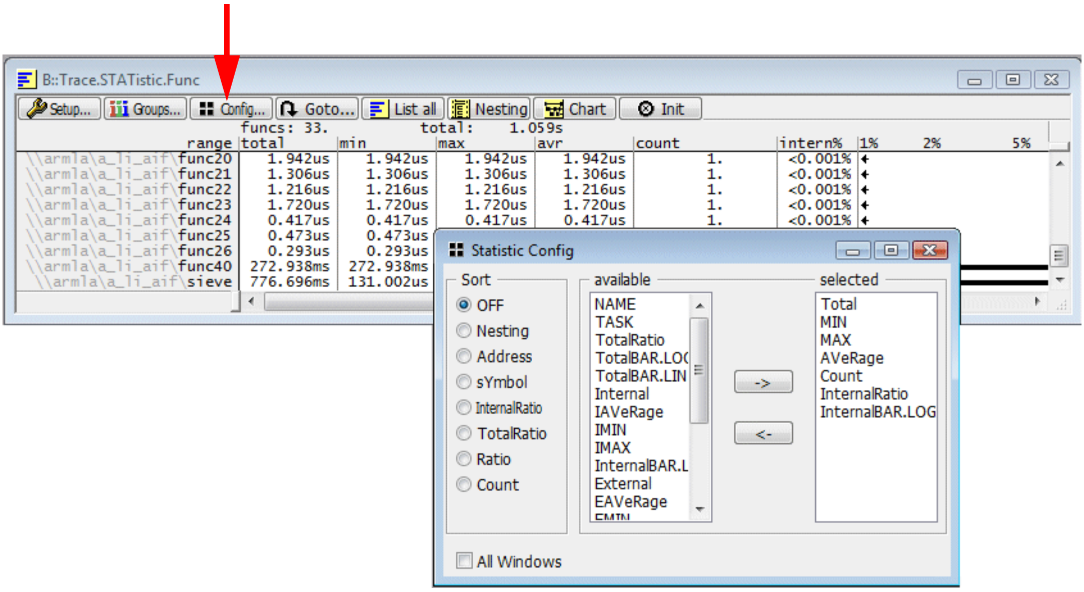
1. 2. (2/0): 2 times within the function, 2 function entries missing
2. 4. (0/3): 4 times within the function, 3 function exits missing
3. 11. (1/1): 11 times within the function, 1 function entry and 1 function exit is missing.



If the number of missing function entries or exits is greater than 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

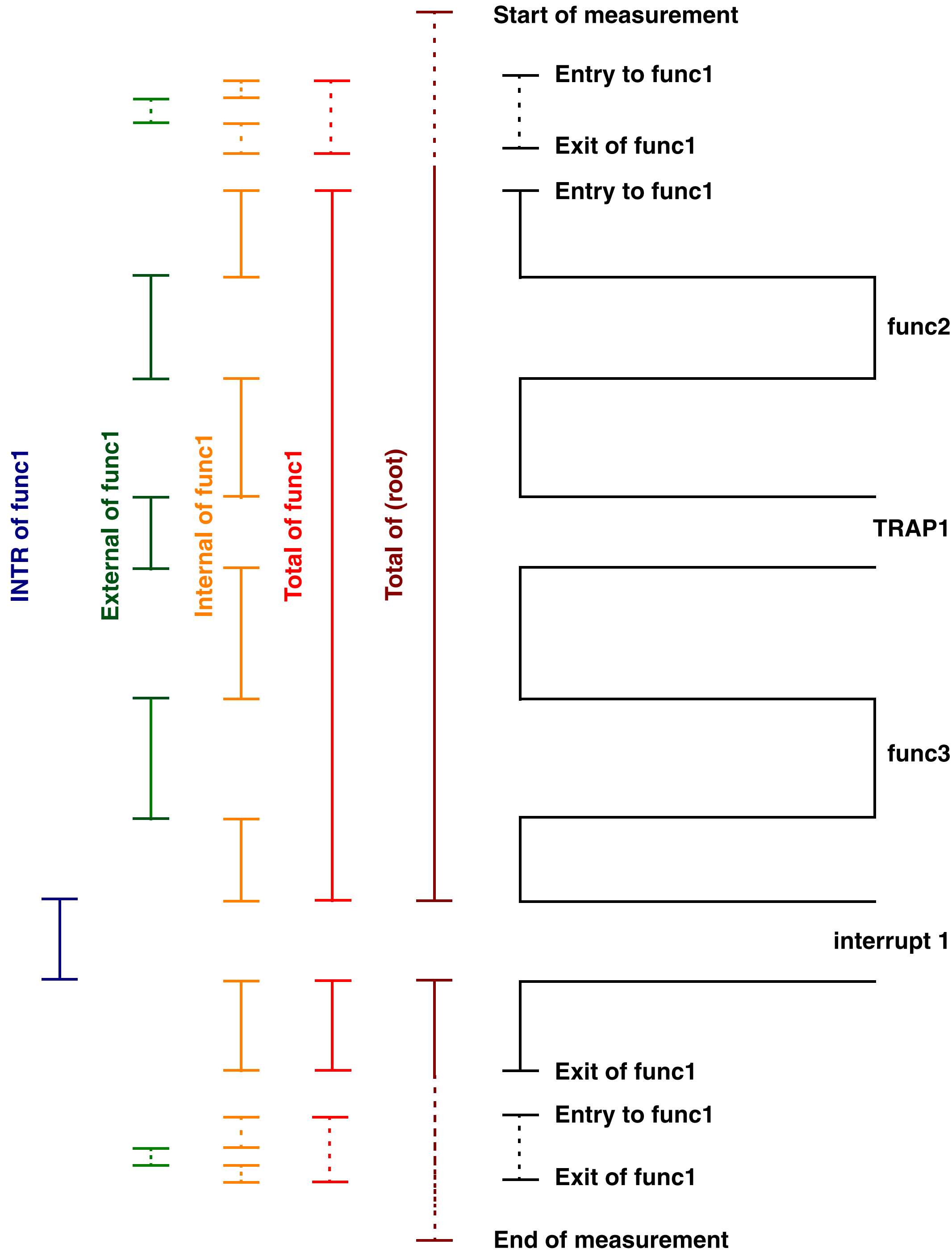


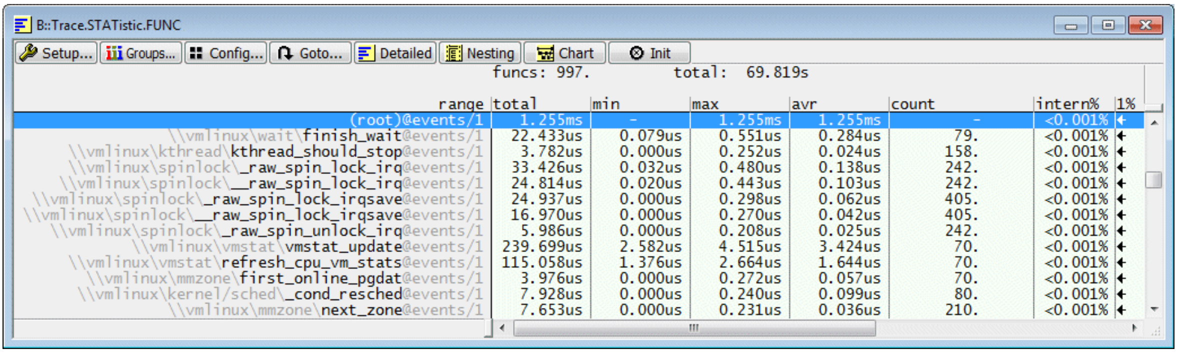
columns (cont.) - times only in function	
Internal	total time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IAVeRage	average time between function entry and exit without called sub-functions, TRAP handlers, interrupt service routines
IMIN	shortest 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:





B::Trace.STATistic.FUNC

Setup... Groups... Config... Goto... Detailed Nesting Chart Init

funcs: 997. total: 69.819s

range	total	min	max	avr	count	intern%
(root)@events/1	1.255ms	-	1.255ms	1.255ms	-	<0.001%
\\vmlinux\wait\finish_wait@events/1	22.433us	0.079us	0.551us	0.284us	79.	<0.001%
\\vmlinux\kthread\kthread_should_stop@events/1	3.782us	0.000us	0.252us	0.024us	158.	<0.001%
\\vmlinux\spinlock_raw_spin_lock_irq@events/1	33.426us	0.032us	0.480us	0.138us	242.	<0.001%
\\vmlinux\spinlock__raw_spin_lock_irq@events/1	24.814us	0.020us	0.443us	0.103us	242.	<0.001%
\\vmlinux\spinlock_raw_spin_lock_irqsave@events/1	24.937us	0.000us	0.298us	0.062us	405.	<0.001%
\\vmlinux\spinlock__raw_spin_lock_irqsave@events/1	16.970us	0.000us	0.270us	0.042us	405.	<0.001%
\\vmlinux\spinlock_raw_spin_unlock_irq@events/1	5.986us	0.000us	0.208us	0.025us	242.	<0.001%
\\vmlinux\vmstat\vmstat_update@events/1	239.699us	2.582us	4.515us	3.424us	70.	<0.001%
\\vmlinux\vmstat\refresh_cpu_vm_stats@events/1	115.058us	1.376us	2.664us	1.644us	70.	<0.001%
\\vmlinux\mmzone\first_online_pgdat@events/1	3.976us	0.000us	0.272us	0.057us	70.	<0.001%
\\vmlinux\kernel/sched_cond_resched@events/1	7.928us	0.000us	0.240us	0.099us	80.	<0.001%
\\vmlinux\mmzone\next_zone@events/1	7.653us	0.000us	0.231us	0.036us	210.	<0.001%

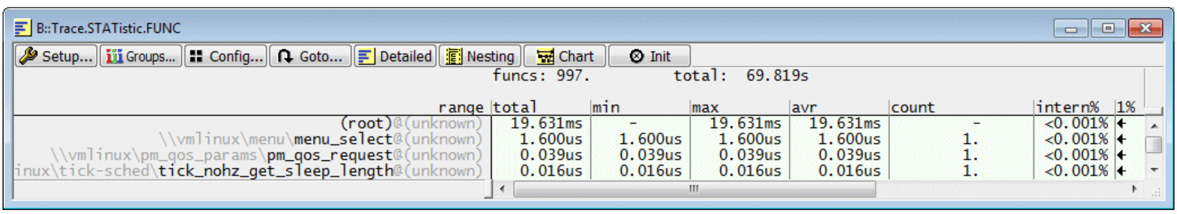
- **HLL function**

  `\\vmlinux\spinlock\_raw_spin_lock_irqsave@events/1`

  HLL function "_raw_spin_lock_irqsave" running in task/process/thread "event/1"

- **Root of call tree for task/process/thread "event/1"**

  `(root)@events/1`



B::Trace.STATistic.FUNC

Setup... Groups... Config... Goto... Detailed Nesting Chart Init

funcs: 997. total: 69.819s

range	total	min	max	avr	count	intern%
(root)@(unknown)	19.631ms	-	19.631ms	19.631ms	-	<0.001%
\\vmlinux\menu\menu_select@(unknown)	1.600us	1.600us	1.600us	1.600us	1.	<0.001%
\\vmlinux\pm_qos_params\pm_qos_request@(unknown)	0.039us	0.039us	0.039us	0.039us	1.	<0.001%
inux\tick-sched\tick_nohz_get_sleep_length@(unknown)	0.016us	0.016us	0.016us	0.016us	1.	<0.001%

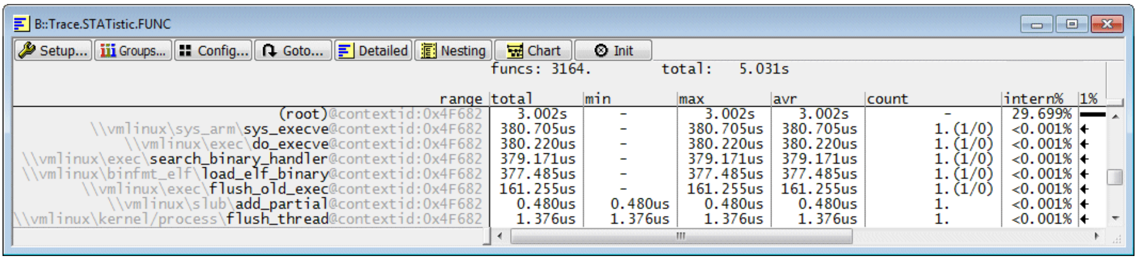
- **Unknow task/process/thread**

  `\\vmlinux\menu\menu_select@(unknown)`

  Before the first task/process/thread switch is found in the trace, the task/process/thread ID is unknown

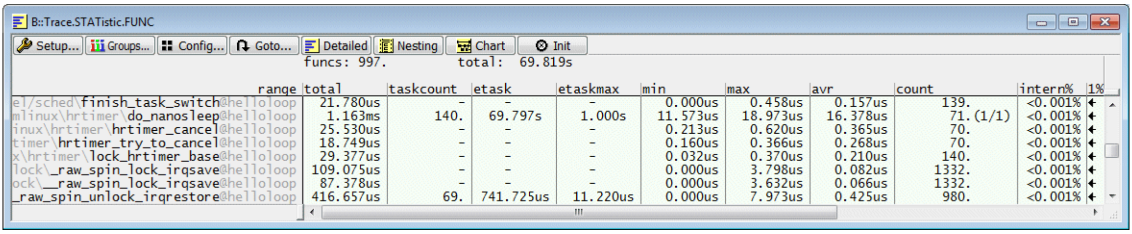
- **Root of unknow task/process/thread**

  `(root)@(unknown)`



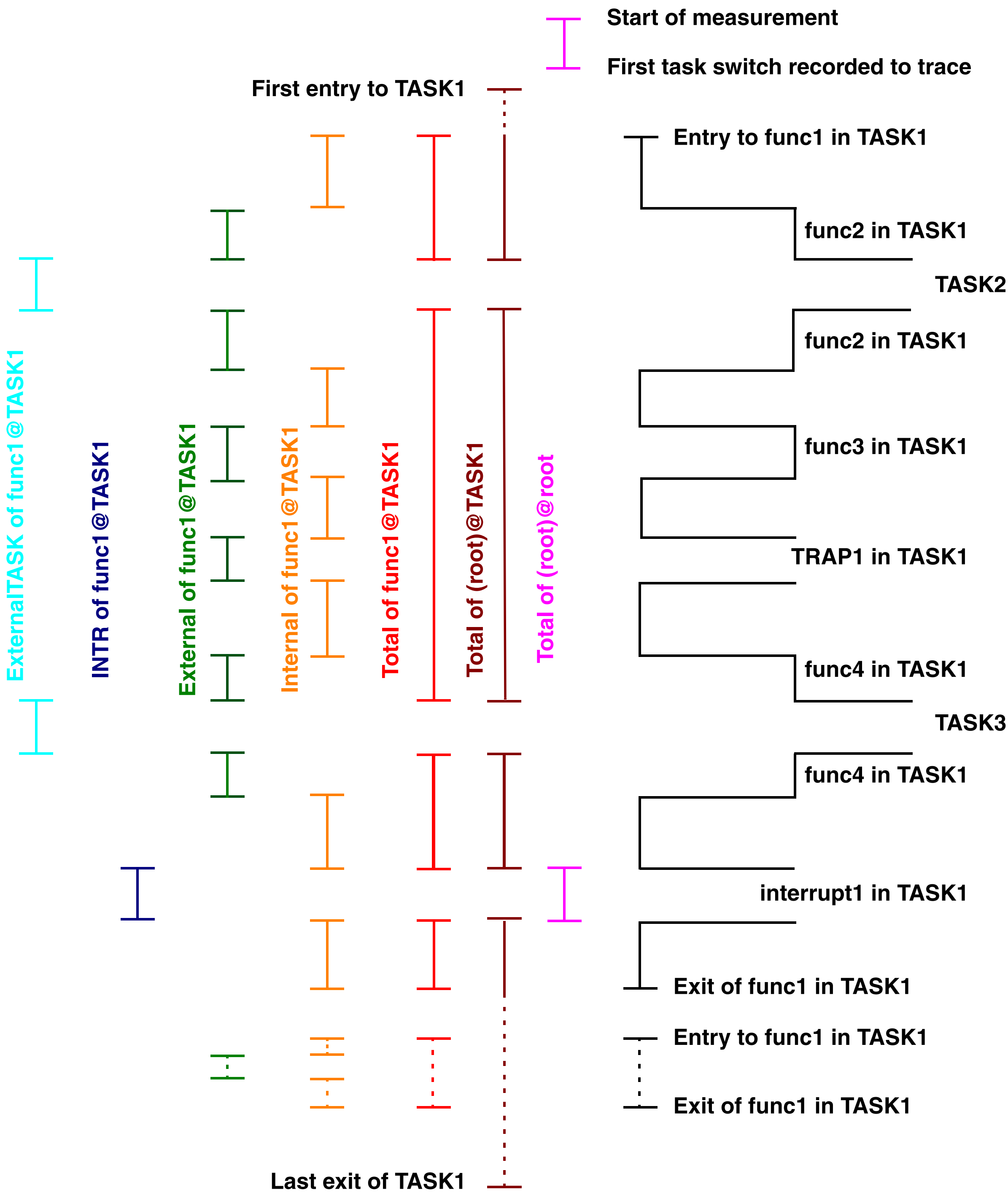
The process/thread ID or the TRACE32 traceid is displayed if a process is already ended. The UNKNOWN cycles are assigned to

(root)@contextid:0x4F682



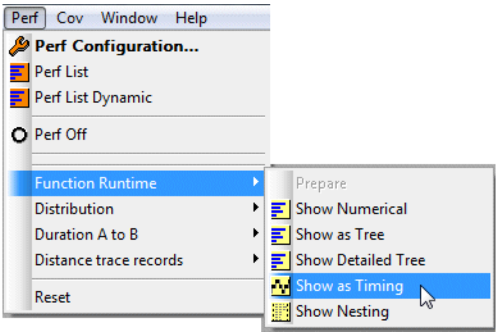
range	total	taskcount	etask	etaskmax	min	max	avr	count	intern%	1%
el/sched\finish_task_switch@helloloop	21.780us	-	-	-	0.000us	0.458us	0.157us	139.	<0.001%	
mlinux\hrtimer\do_nanosleep@helloloop	1.163ms	140.	69.797s	1.000s	11.573us	18.973us	16.378us	71.(1/1)	<0.001%	
inux\hrtimer\hrtimer_cancel@helloloop	25.530us	-	-	-	0.213us	0.620us	0.365us	70.	<0.001%	
timer\hrtimer_try_to_cancel@helloloop	18.749us	-	-	-	0.160us	0.366us	0.268us	70.	<0.001%	
x\hrtimer\lock_hrtimer_base@helloloop	29.377us	-	-	-	0.032us	0.370us	0.210us	140.	<0.001%	
lock_raw_spin_lock_irqsave@helloloop	109.075us	-	-	-	0.000us	3.798us	0.082us	1332.	<0.001%	
ock__raw_spin_lock_irqsave@helloloop	87.378us	-	-	-	0.000us	3.632us	0.066us	1332.	<0.001%	
_raw_spin_unlock_irqrestore@helloloop	416.657us	69.	741.725us	11.220us	0.000us	7.973us	0.425us	980.	<0.001%	

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

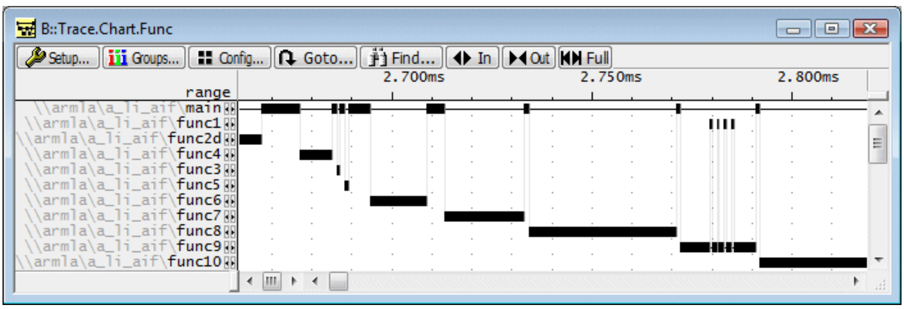


Start of measurement
First task switch recorded to trace
First entry to TASK1
Entry to func1 in TASK1
func2 in TASK1
TASK2
func2 in TASK1
func3 in TASK1
TRAP1 in TASK1
func4 in TASK1
TASK3
func4 in TASK1
interrupt1 in TASK1
Exit of func1 in TASK1
Entry to func1 in TASK1
Exit of func1 in TASK1
Last exit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

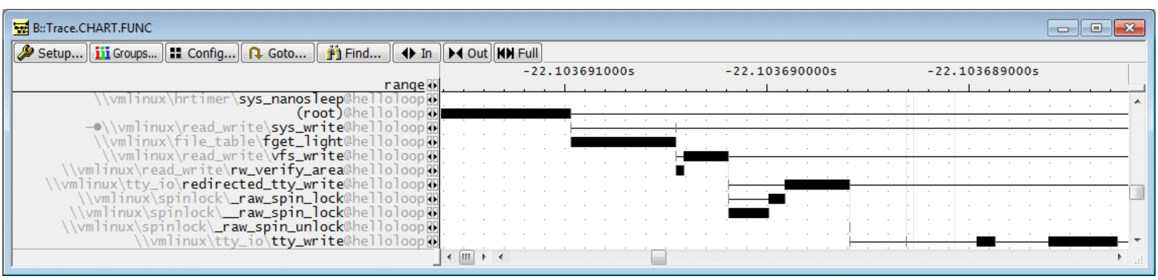
Trace.Chart.Func	Nested function run-time analysis - graphical display



Look and Feel (No OS)

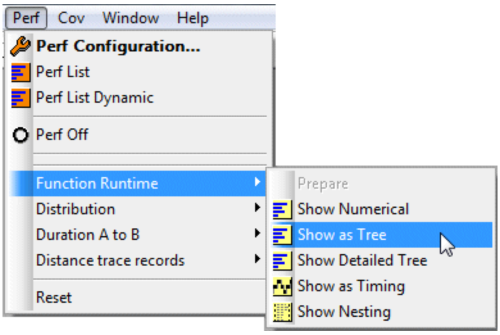


Look and Feel (OS or OS+MMU)

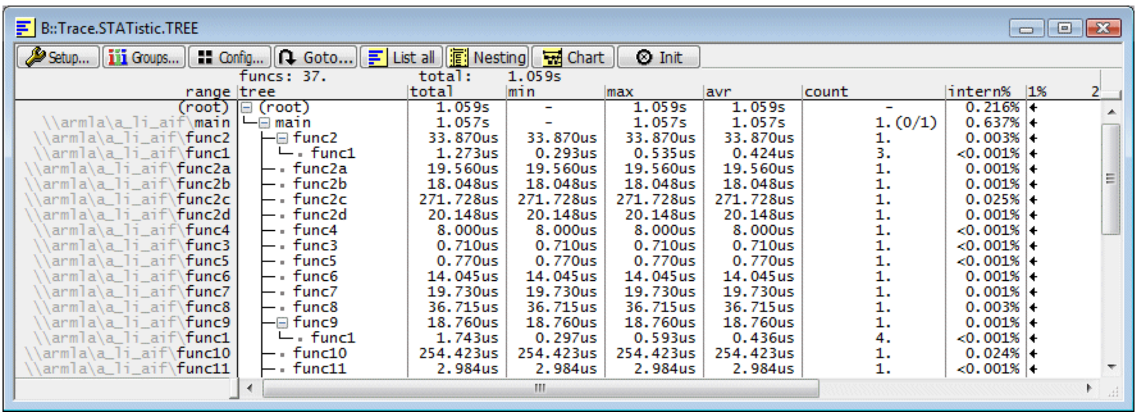


Trace.STATistic.TREE

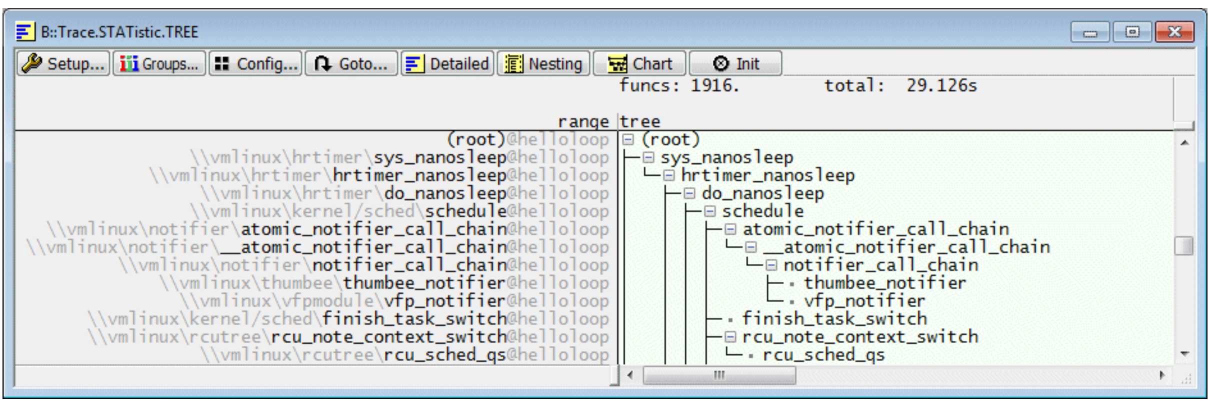
Nested function run-time analysis
- tree display



Look and Feel (No OS)



Look and Feel (OS or OS+MMU)

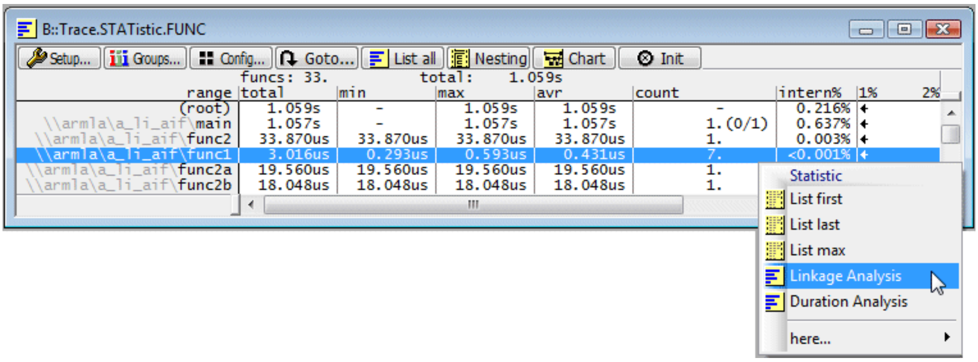


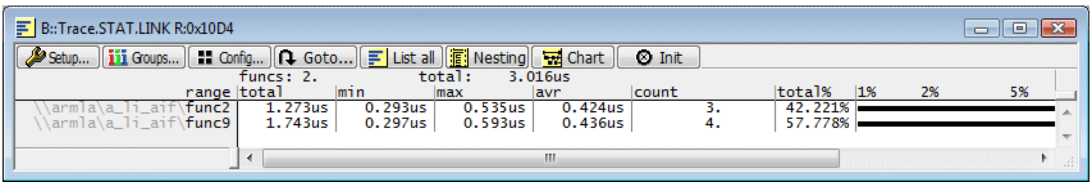

```
Trace.STATistic.TREE /TASK "helloloop"
```

Trace.STATistic.LINKage *<address>*

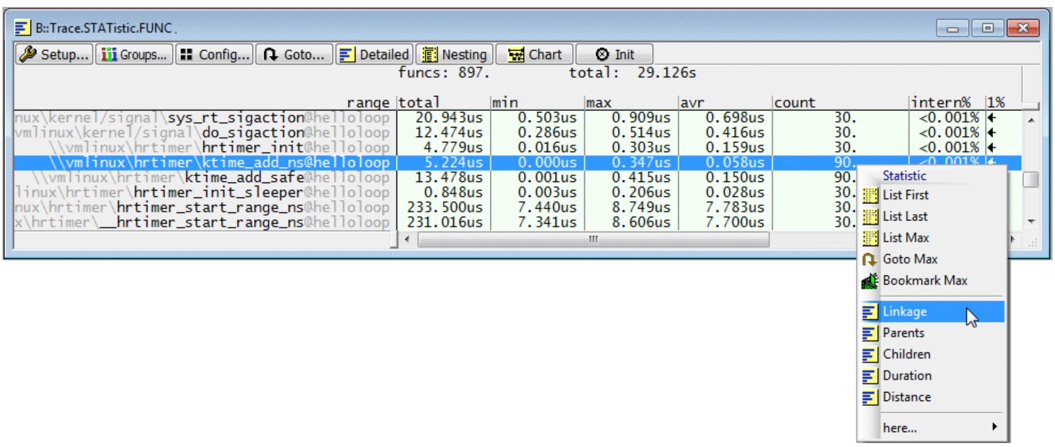
Nested function run-time analysis
- linkage analysis

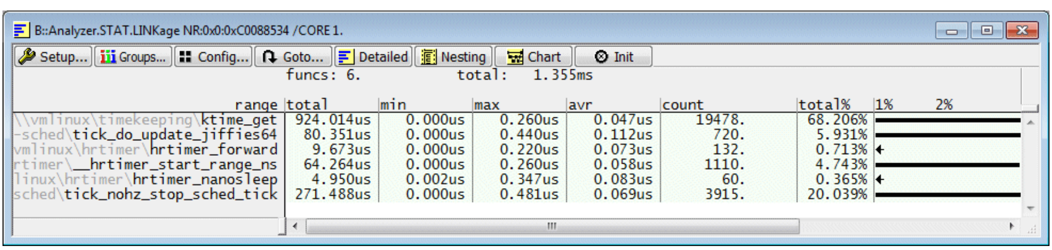
Look and Feel (No OS)





Look and Feel (OS or OS+MMU)





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. However, the manual does not contain details about the architecture-specific setups. Here is an overview of the setups for ARM-ETM.

Optimum ETM Configuration (No OS or OS)

Code coverage does **not** require any **data information** if no OS or an OS is used. That's why it is recommended to switch the broadcasting of data information off.

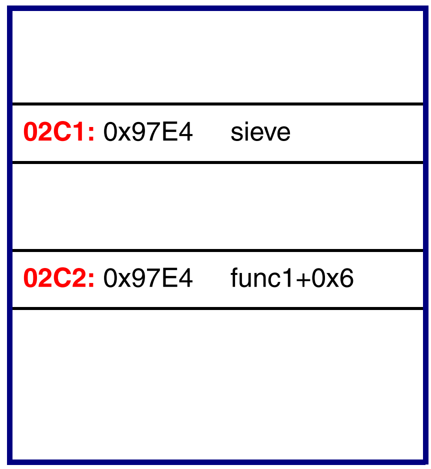
```
ETM.DataTrace OFF
```

Optimum ETM Configuration (OS+MMU)

Virtual address: 0x97E4

The virtual address exported by the ETM is not enough to identify the function/ symbol range.

The Address Space ID is required!



TRACE32 Symbol Database

If an target operating system is used, that uses dynamic memory management to handle processes, the instruction flow plus information on the **Address Space ID** is required in order to perform a code coverage analysis.

The standard way to get the Address Space ID is to advise the ETM to export the instruction flow and the process switches. For details refer to the chapter **OS-Aware Tracing** of this training.

Optimum Configuration 1 (process switches are exported in form of a special write access)**:**

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

Optimum Configuration 2 (process switches are exported in form of a Context ID packet)**:**

```
ETM.ContextID 32
```