Training ICE Analyzer

Version 18-Nov-2020

Introduction

Functional Units of the Analyzer

The analyzer consists of the following parts:

- Trace buffer
- Programmable trigger unit
- Performance analyzer unit

Available as:

- HA120: High Speed State Analyzer

  Analyzer is a separate module of TRACE32-ICE
- HAC:32-bit compact Analyzer

  Analyzer is part of the ECC32 module
- ECC: 8-bit compact Analyzer

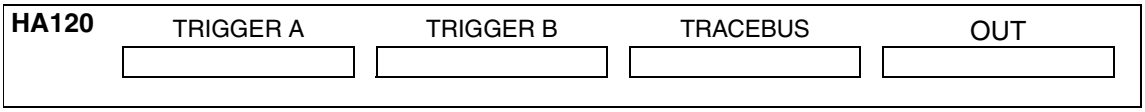
  Analyzer is part of the ECC8 module

	HA120	**HAC**	**ECC8**
Trace buffer size	32 K * 120 bit	32 K * 96 bit	32 K *88 bit
Smallest cycle time	50 ns	50 ns	150 ns
Resolution of the time stamp unit	25 ns 48 bit	100 ns 32 bit	100 ns 40 bit
Trigger Inputs	2 channels / 2 * 8 bit 1 coax input A	1 channels / 1 * 8 bit	1 channel / 1*8 it 1 coax input A
Trigger Outputs	1 coax output A 2 outputs		1 coax output A 2 outputs
Performance Analyzer	64 areas	-	32 areas

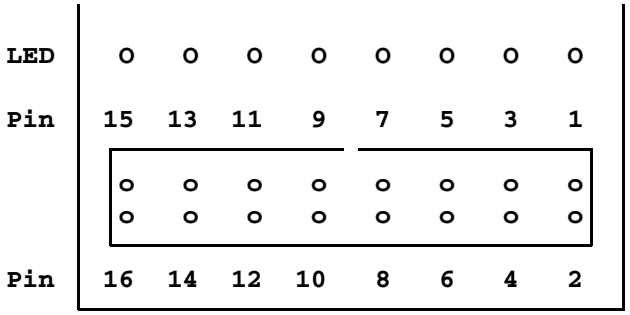
For the main differences for the programmable trigger unit, refer to the section ***Trigger Programming***.

Probe Connectors

HA120

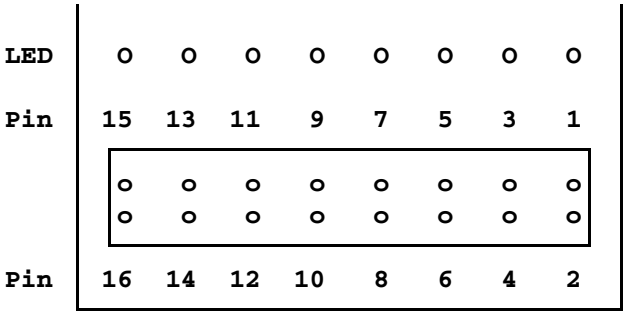


Probe Connectors		
TRIGGER A/TRIGGER B	Input	External trigger inputs
TRACEBUS		Not used
OUT	Output	Trigger outputs C and D



Pin 1	Line 0	Data 0
Pin 3	Line 1	Data 1
Pin 5	Line 2	Data 2
Pin 7	Line 3	Data 3
Pin 9	Line 4	Data 4
Pin 11	Line 5	Data 5
Pin 13	Line 6	Data 6
Pin 15	Line 7	Data 7

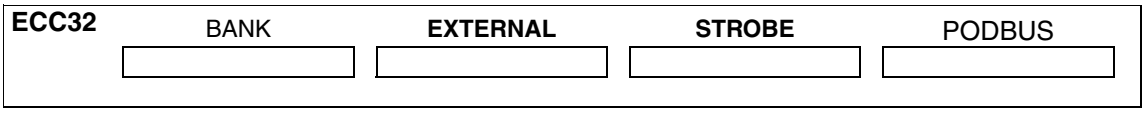
Pin 2, 4, 6, 8, 10, 12, 14, 16 Ground



HA120:

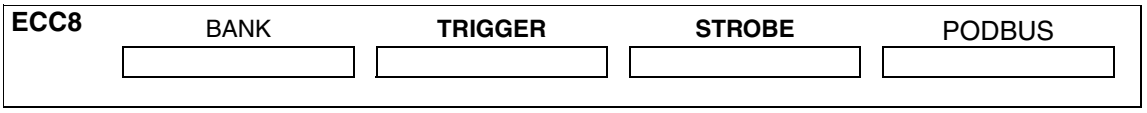
```
Pin   1    Trigger output C high-active
Pin   3    Trigger output D high-active
Pin   5    Not assigned
Pin   7    Not assigned
Pin   9    Not assigned
Pin  11    Not assigned
Pin  13    Cycle signal (synchr. signal to CPU)
Pin  15    Not assigned

Pin  2,4,6,8,10,12,14,16    GND
```

HAC



Probe Connectors		
EXTERNAL	Input	External trigger inputs
STROBE	Output	Not used

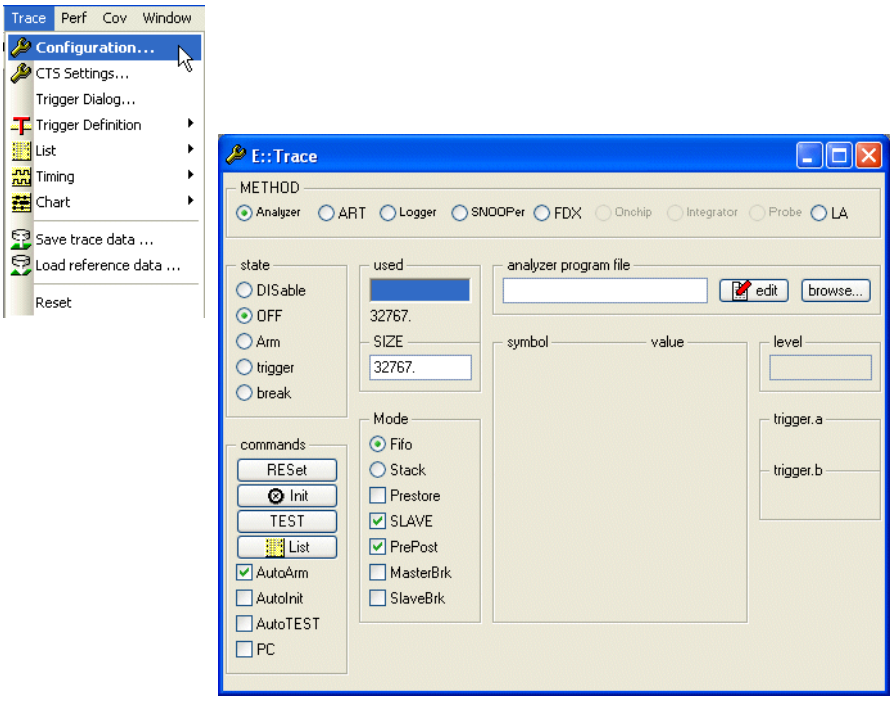
ECC8



Probe Connectors		
TRIGGER	Input	External trigger inputs
STROBE	Output	Trigger outputs C and D

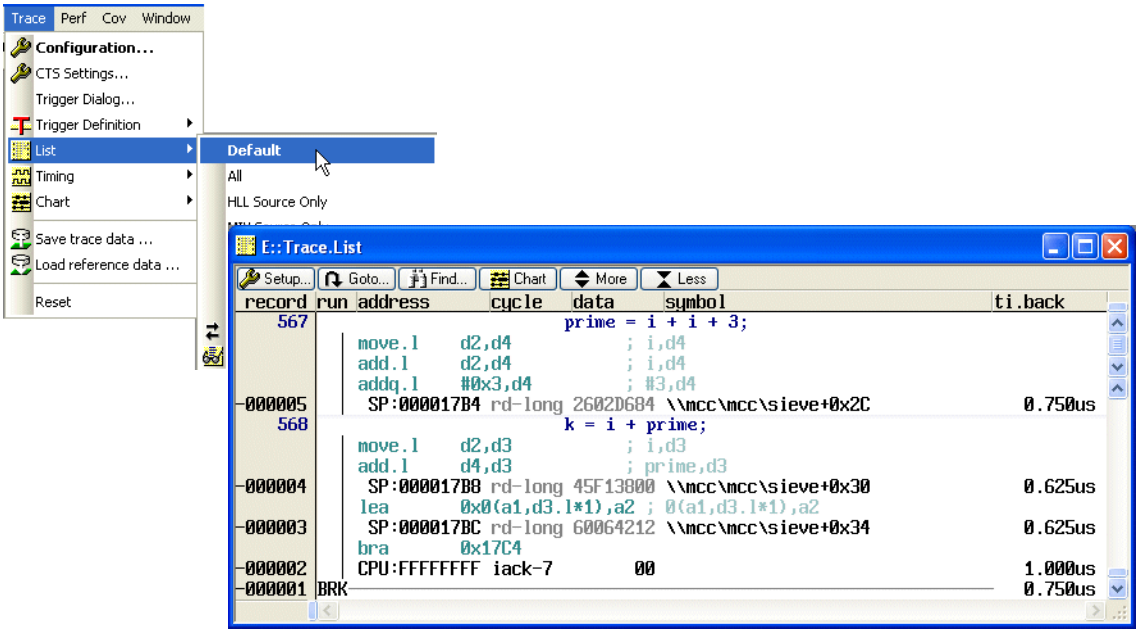
The recording to the trace buffer is controlled

- By the settings of the TRace Configuration Window



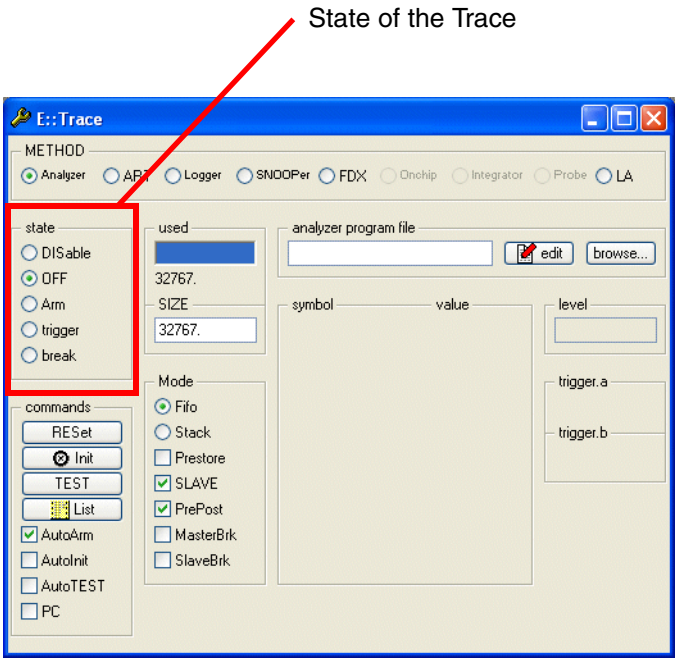
- By the Analyzer Trigger Unit (Trigger/Filter)

  If no trigger program is active, the trace will record all CPU cycles.



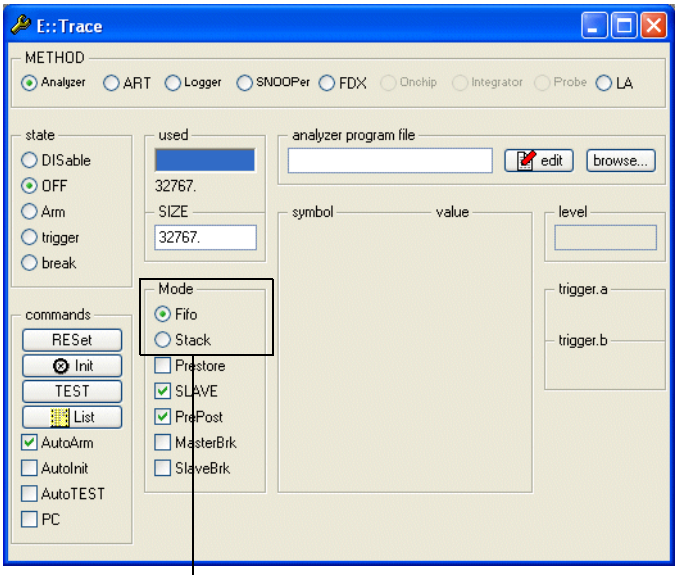
States of the Trace

The trace buffer can either sample or display the information.



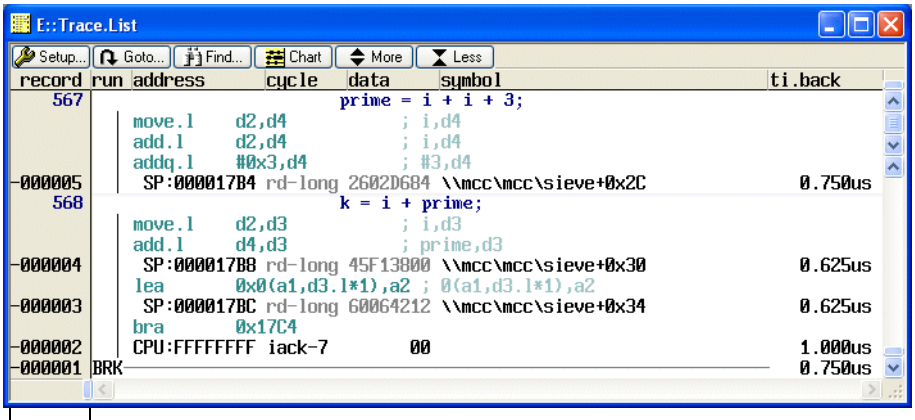
State of the Trace	
DISable	The Trace is not working
OFF	The Trace is not active. Trace buffer can be displayed.
Arm	The Trace is acitve (information is sampled into the trace buffer). Trace buffer can not be displayed.
trigger	Not used for TRACE32-ICE
break	Trace is in break state (BREAK command from a trigger program), trace buffer can be displayed.

Sampling Modes of the Trace



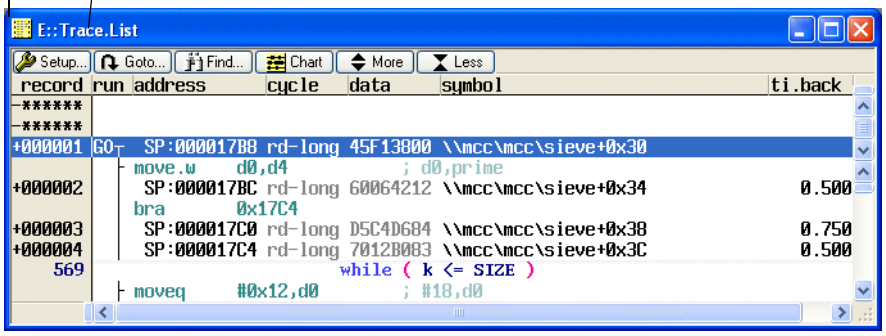
Sampling Modes

Sampling Modes	
Fifo	The trace is working in FIFO mode. When the trace buffer is full, the new records will overwrite the older ones. The trace records always the last cycle before the program execution is stopped.
Stack	The trace is working in STACK mode. If the trace buffer is full, the trace will stop sampling. The trace records always the first cycle after the program start.

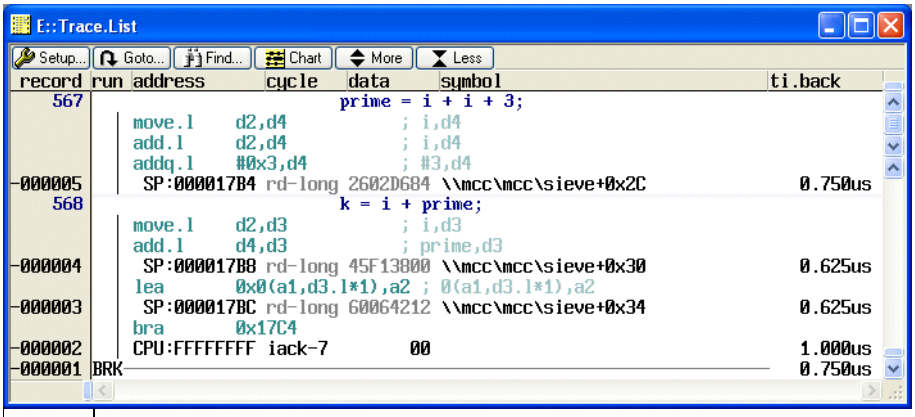


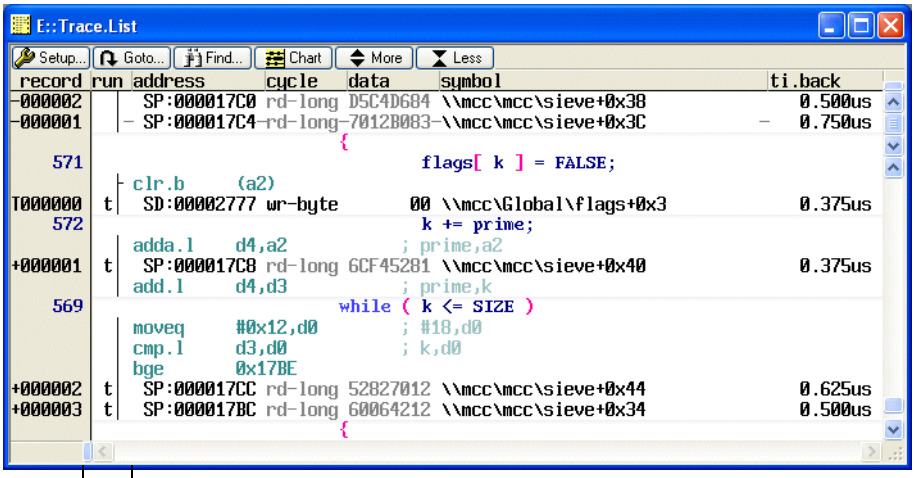
The trace is working in FIFO mode, last record sampled before the stop is record (-1)

Trace is working in STACK mode, the first cycle has record number +1



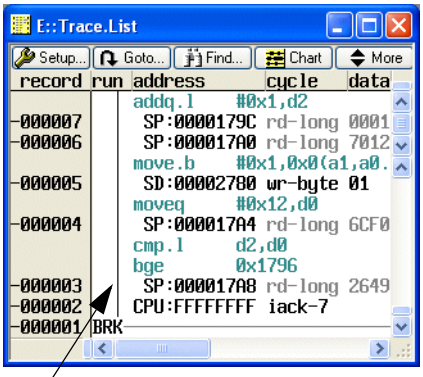
Record Number



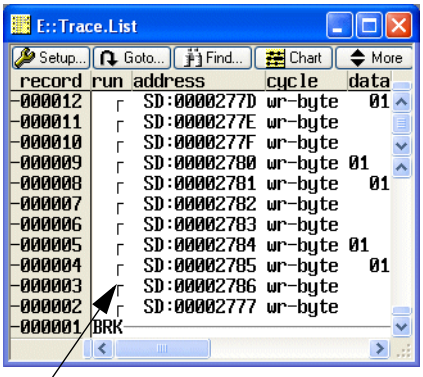


Run condition

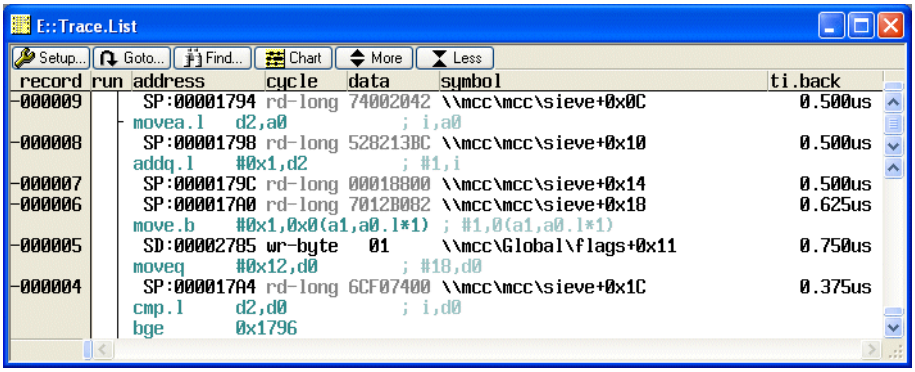
T	Record where an **Emulator Trigger Event** happened (Trigger Point). This record becomes record 0.
t	Record was sampled after an Emulator Trigger Event happened



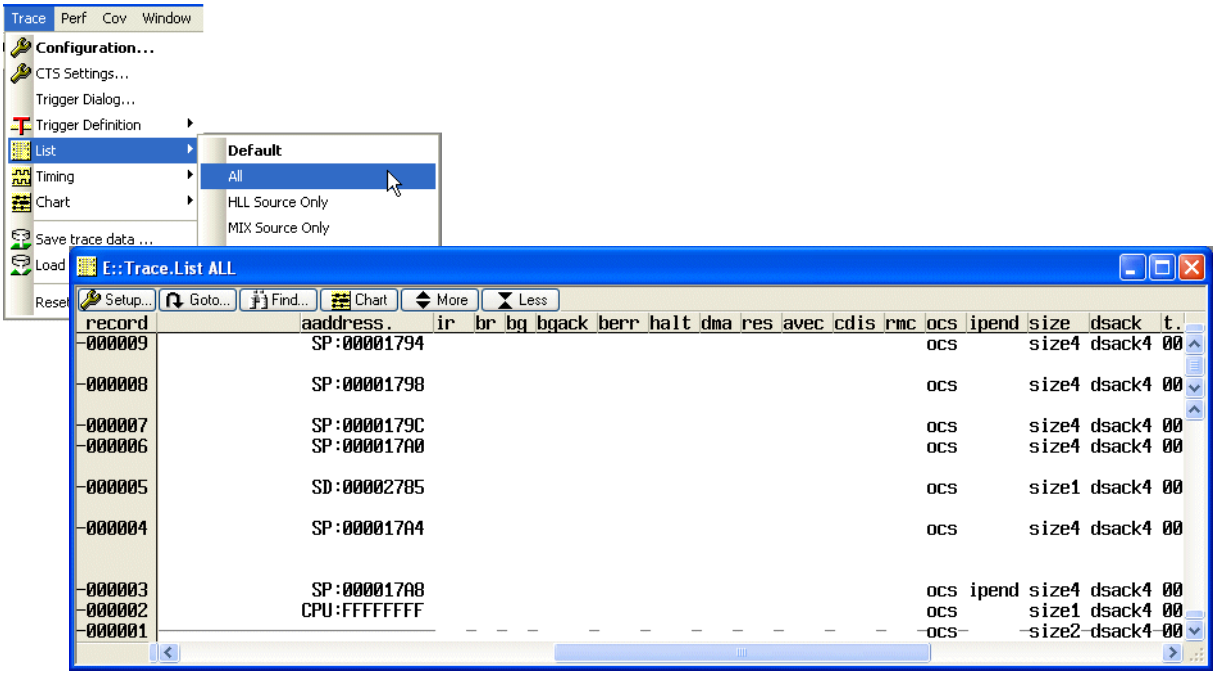
A straight line indicates that all CPU cycles were sampled continuously.



A broken line indicates that between two sampled records there were not sampled CPU cycles.



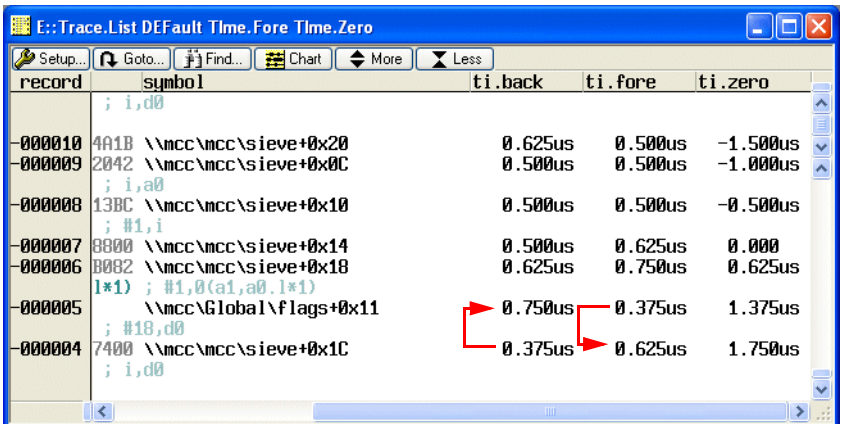
Address	CPU address bus
CYcle	CPU bus cycle type
Data.B/Data.W/Data.L	CPU data bus
sYmbol	Symbolic address with path and offset



AAdress	Absolute (physical) address

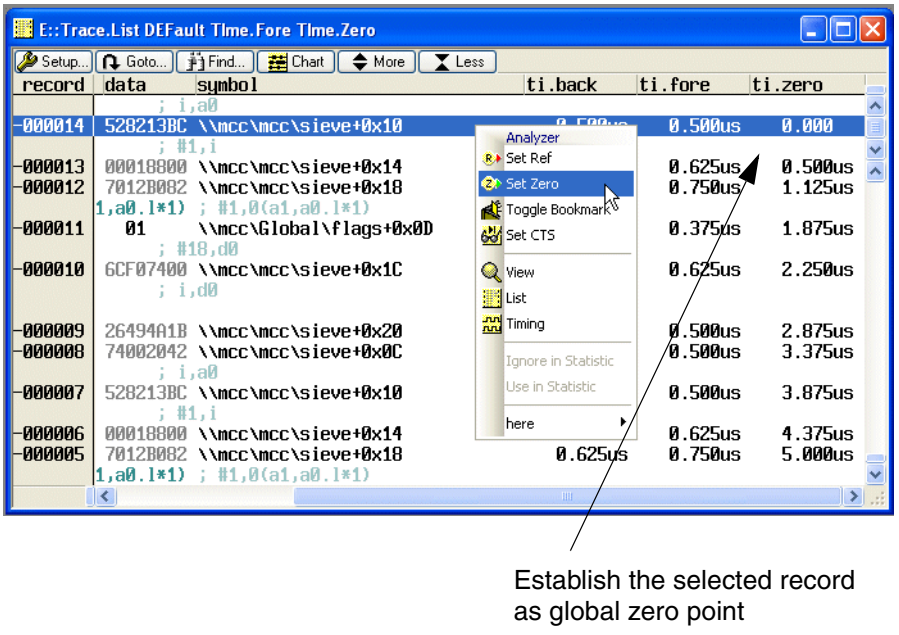
For a list and a description of the CPU specific signals that are sampled in the trace buffer refer to ***ICE Target Guide***, section ***State Analyzer*** chapter ***Keyword for Display*** (Menu: **Help→ICE Target Guide)**.

Additional CPU lines can be sampled using the Port Analyzer (64 lines) or the external trigger inputs (16 lines).



TIme.Back	Time relative to the previous record.
TIme.Fore	Time relative to the next record.
TIme.Zero	Time relative to the global zero point.

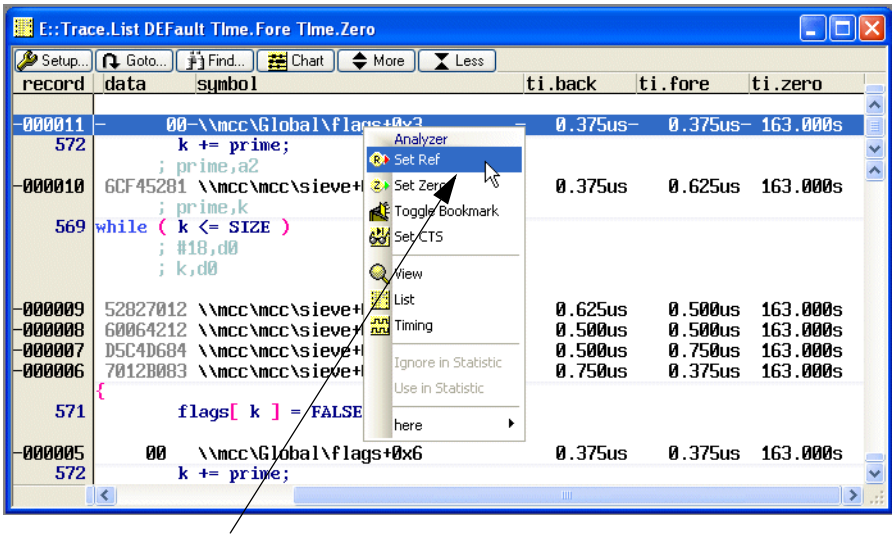
Set the Global Zero Point



Establish the selected record as global zero point

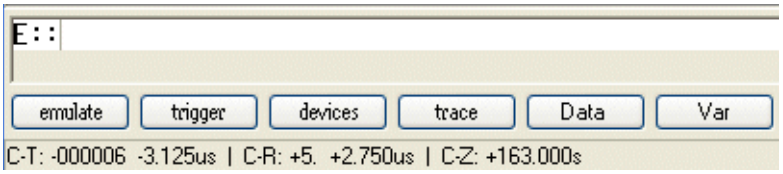
Set the Reference Point

By default, the reference point is always the last record in the trace buffer.



Establish the selected record as reference point

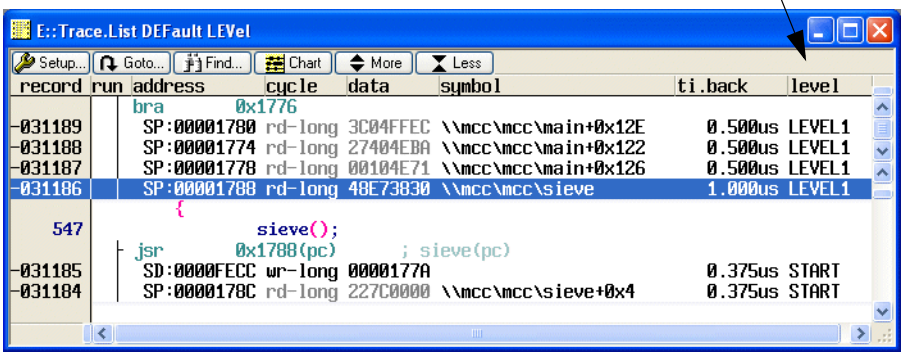
If you select a record in the trace buffer, you get the following time information in the state line:



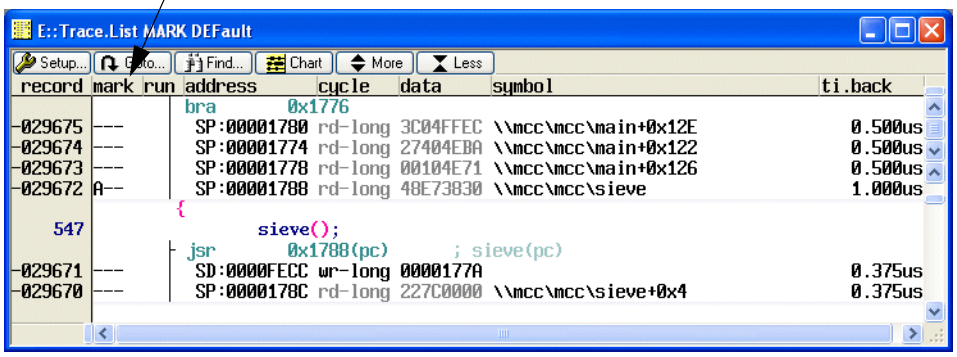
C-T **Cursor-Trigger**	Number of records that were sample from the record at the cursor position to the trigger point.	Time distance from the record at the cursor position to the trigger point.
C-R **Cursor-Reference**	Number of records that were sample from the record at the cursor position to the reference point.	Time distance from the record at the cursor position to the reference point.
C-Z **Cursor-Zero**		Time distance from the record at the cursor position to the global zero point.

Information about the Trigger Level

If the recording to the trace buffer is controlled by a trigger program with several trigger levels, the trigger level, that was active, when the record was sampled is displayed here.

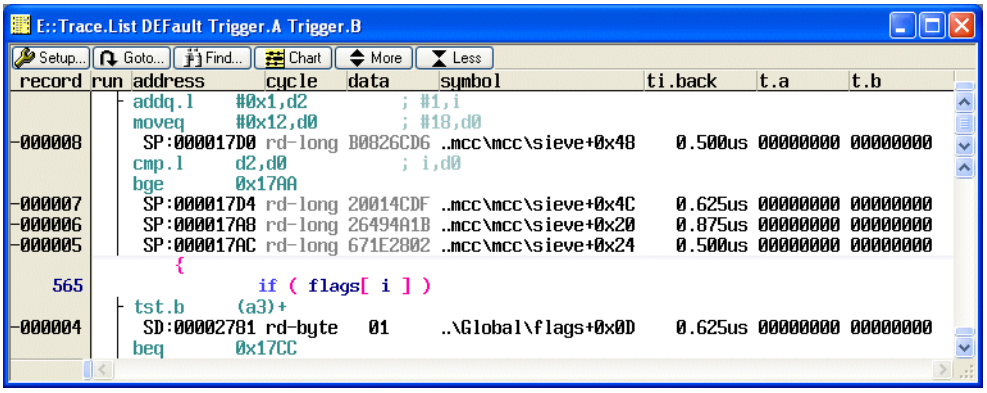


Specific record in the trace buffer can be marked, to make it easier to find and display specific events



Available markers:

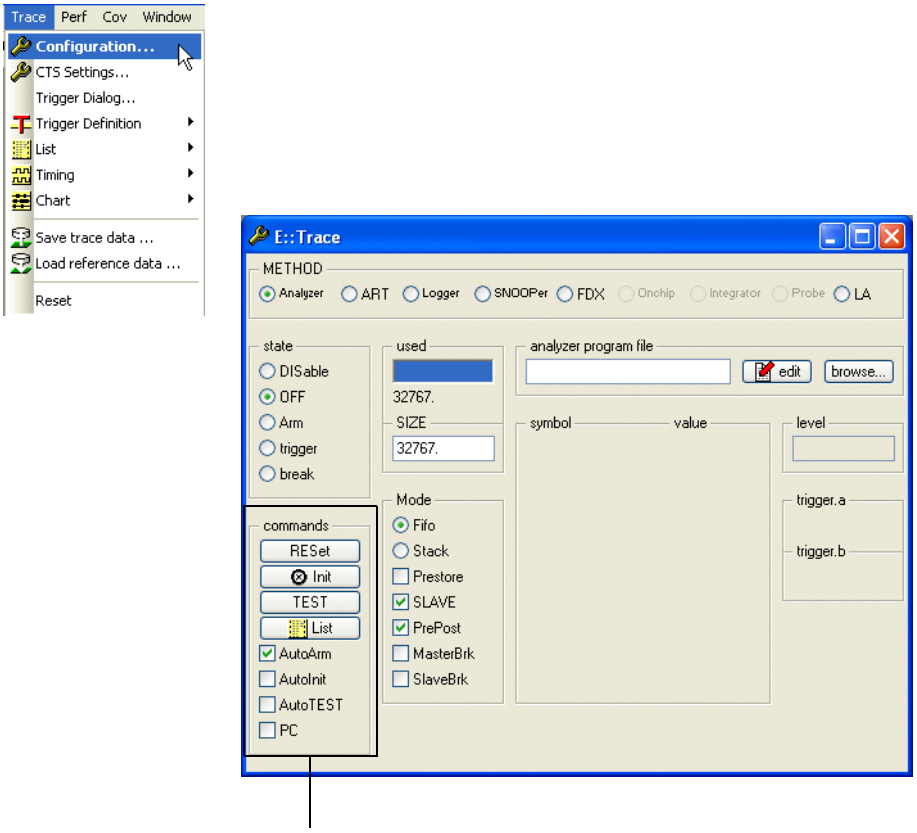
HA120	A, B, C
ECC8	A, B
HAC	A, B, C



Available External Trigger Inputs:

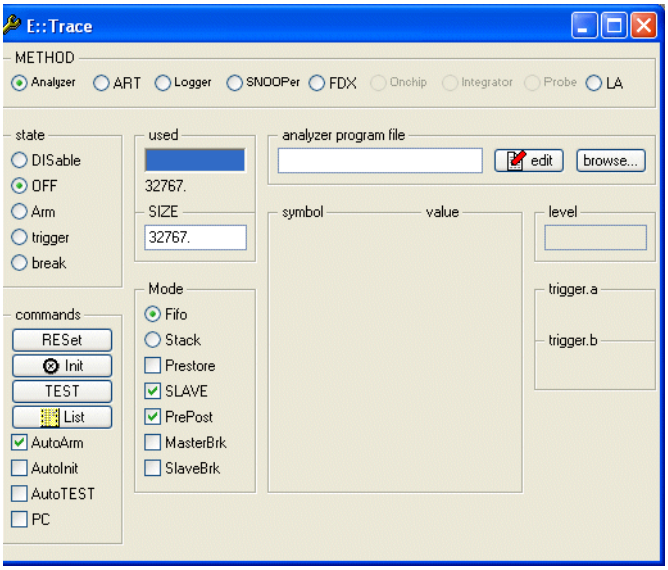
HA120	TRIGGER A, TRIGGER B	16 trigger lines
ECC8	TRIGGER	8 trigger lines
HAC	EXTERNAL	8 trigger lines

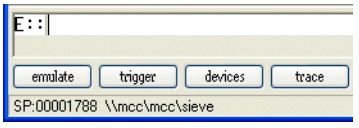
Trace Configuration Window

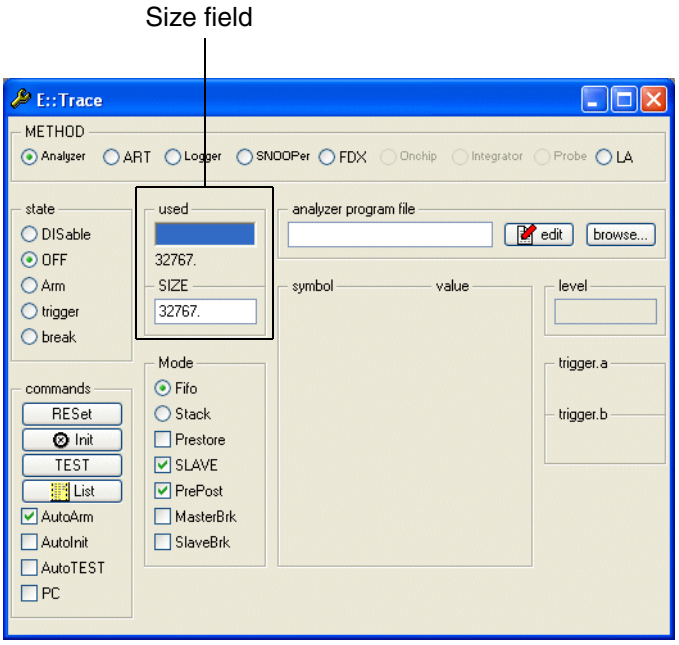


Trace Configuration Commands

Trace.state	Display Trace configuration window

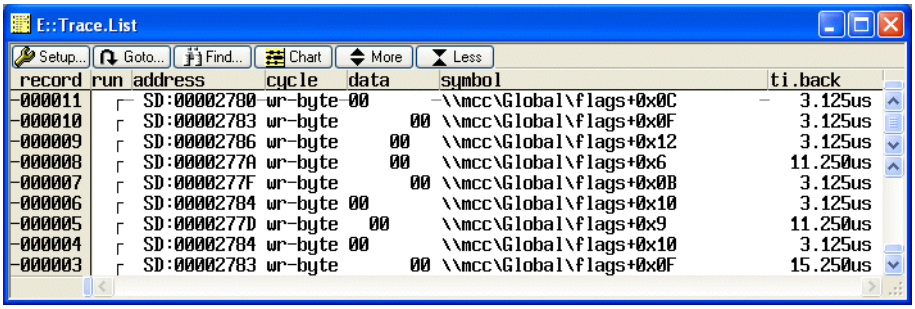


RESet	Reset the Trace Configuration Window to its default settings.
Init	Clears the trace buffer and initializes the programmable trigger unit.
TEST	Rarely used.
AutoArm	Trace is automatically armed, when user program is started. Trace is automatically switched off, when user program stops.
AutoTEST	Rarely used.
AutoInit	By default, the trace buffer is not cleared when the user program is restarted by Go or Step. Autoinit automatically clears the trace buffer and initializes the programmable trigger unit at every program start.
PC	ON: Displays the PC in the state line, while the CPU is running in real time. 

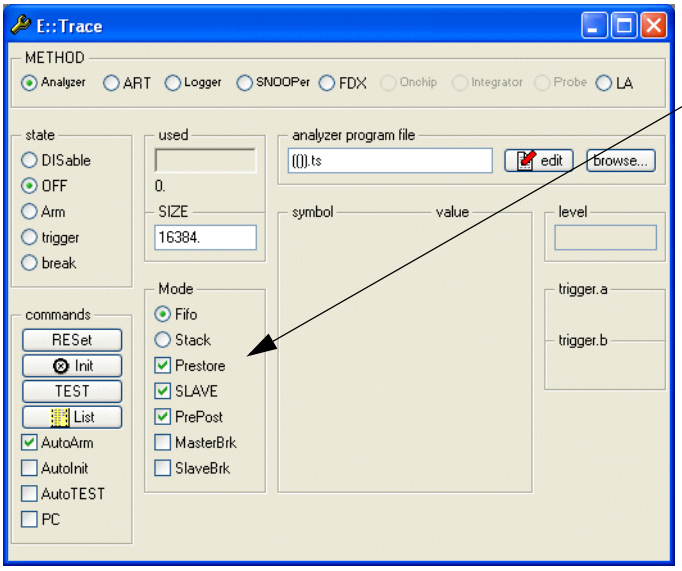


used	Number of sampled records in the trace buffer If the number of used records is gray, this indicates a trace buffer overflow.
SIZE	Size can be decreased for faster search or save operations.

Prestore Mode

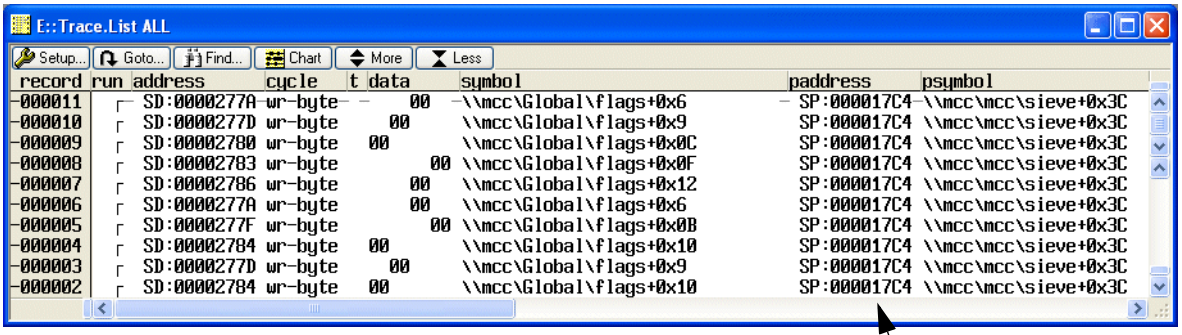


At a selective trace on data accesses it might be interesting to see, which function made the data access.

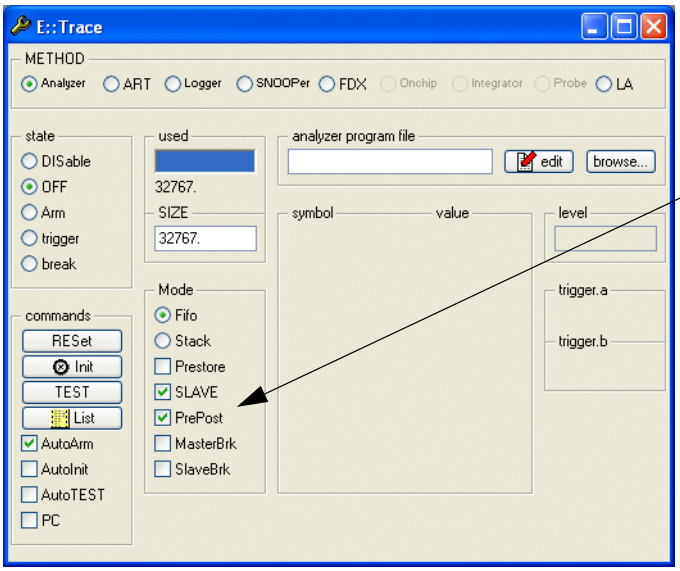


Switch the Prestore mode ON, to sample also the last opfetch before the selective data access.

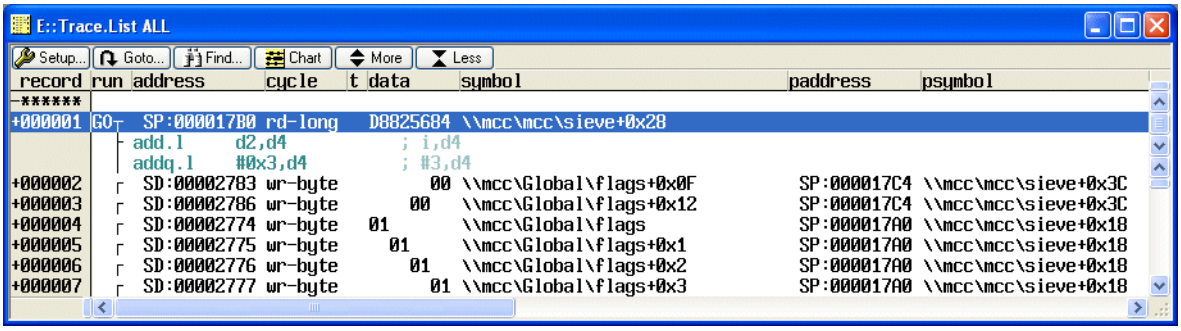
The Prestore mode halves the size of the trace buffer.



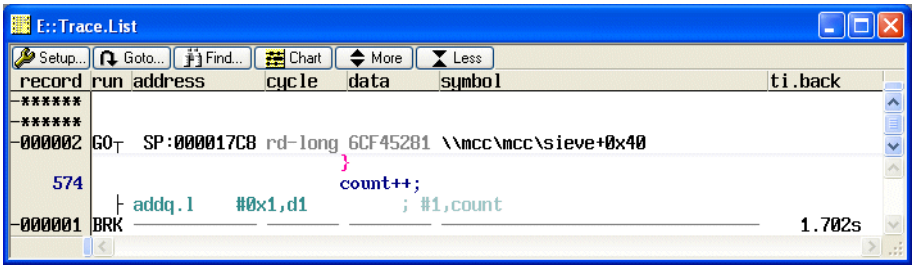
Opfetch before the data access



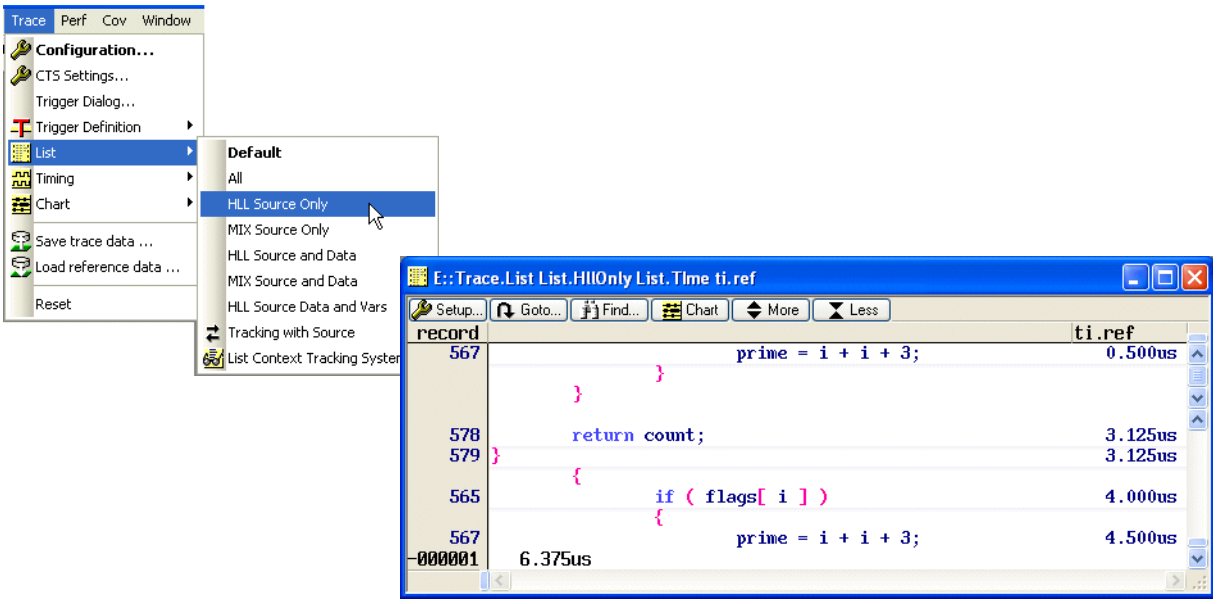
The PrePost Mode forces a recording of the opfetch cycle at the beginning of the program execution (***GO***) and at the end of the program execution (***BRK***).

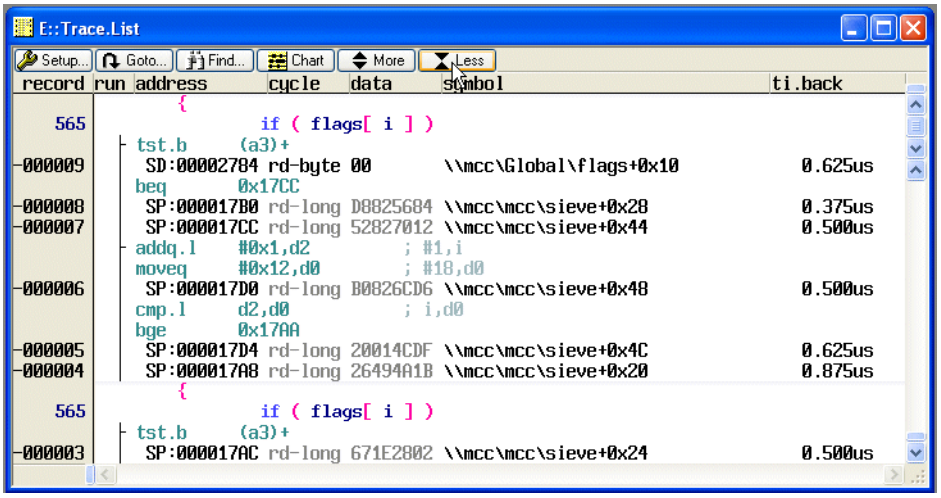


The PrePost option is useful, if you write a trigger program for a selective trace, but nothing is sampled in the trace buffer. The GO and BRK record in the trace buffer show you at least, the user program was executed from GO to BRK and how much time was spent.



Format the Trace Buffer





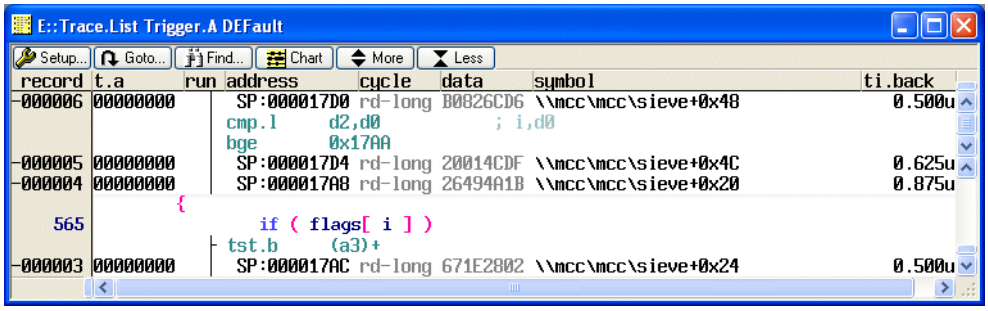
1. time Less	Suppress the display of dummy cycles
2. time Less	Suppress the display of opfetch cycles
3. time Less	Display only HLL sources and data fetches
4. time Less	Display only HLL sources

The ***More*** button works viseversa.

Trace.List	Display Trace contents

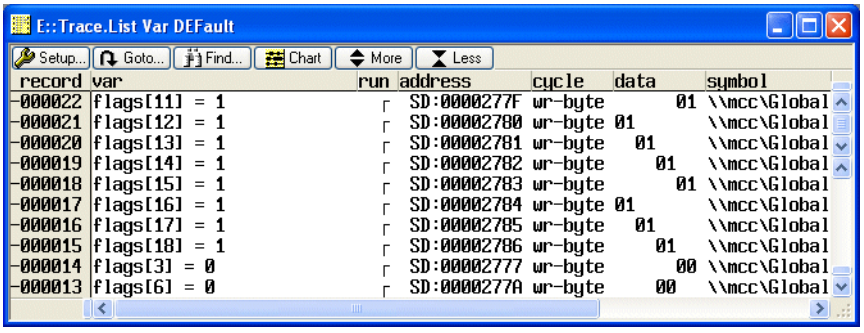
Example 1: External trigger input TRIGGER A plus default items.

```
Trace.List Trigger.A DEFault
```

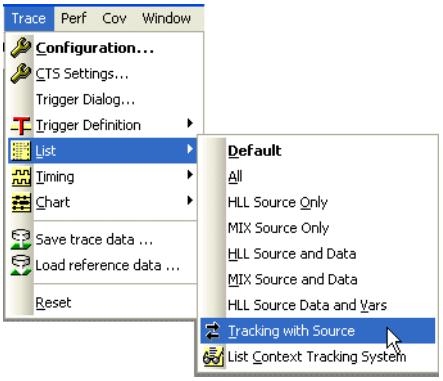


Example 2: Symbolic display of data accesses as HLL variables plus default items.

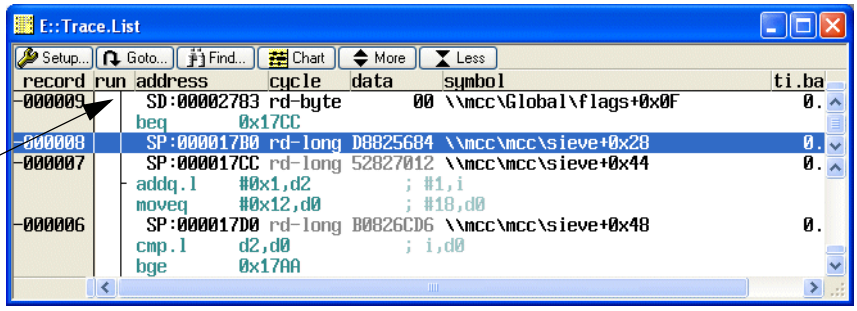
```
Trace.List Var DEFault
```



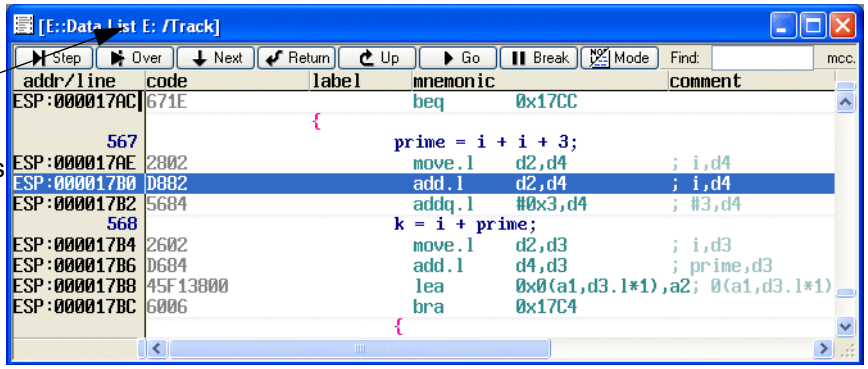
Correlate Trace List to Source



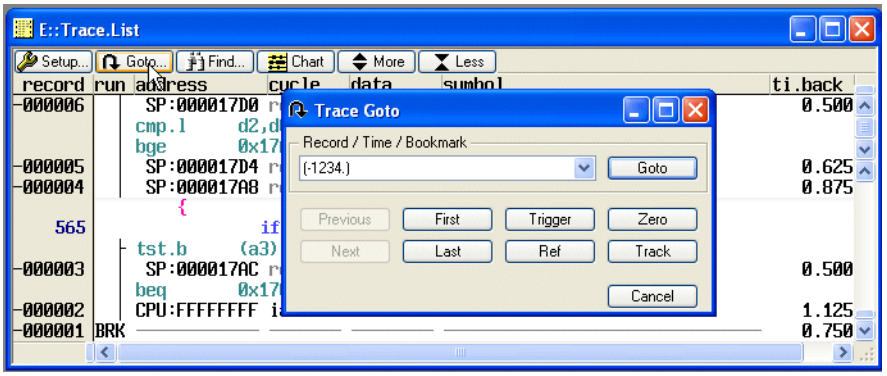
Active Window



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

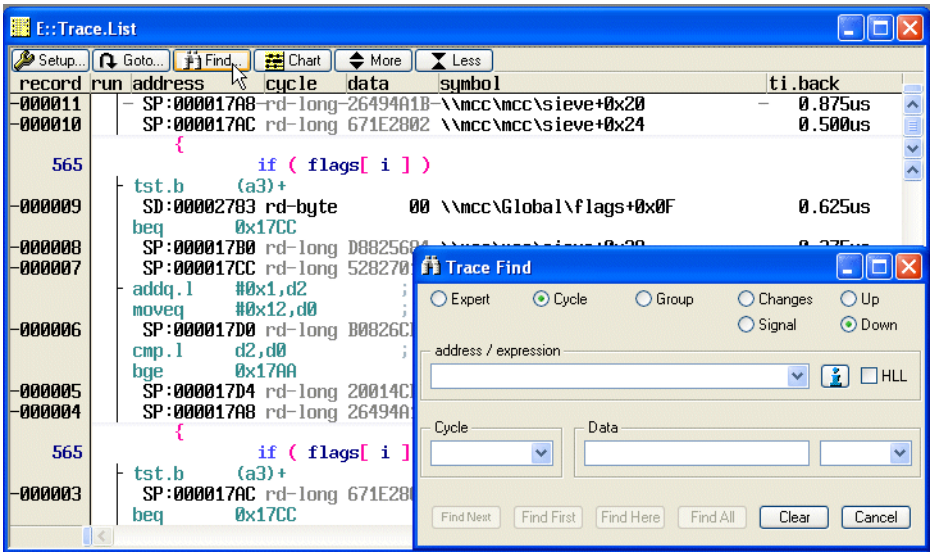


Browse through the Trace Buffer

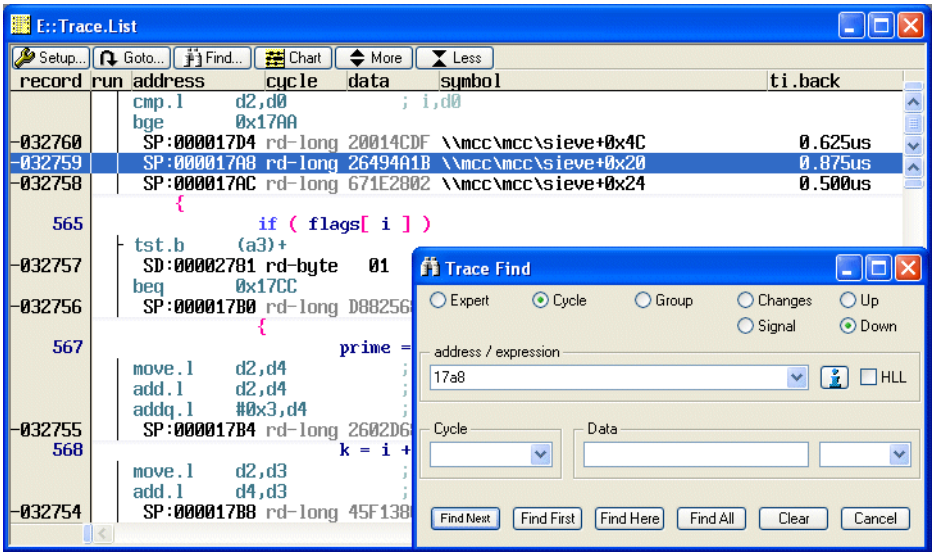


Pg ↑	Scroll page up
Pg ↓	Scroll page down
Ctrl - Pg ↑	Go to the first record
Ctrl - Pg ↓	Go to the last record

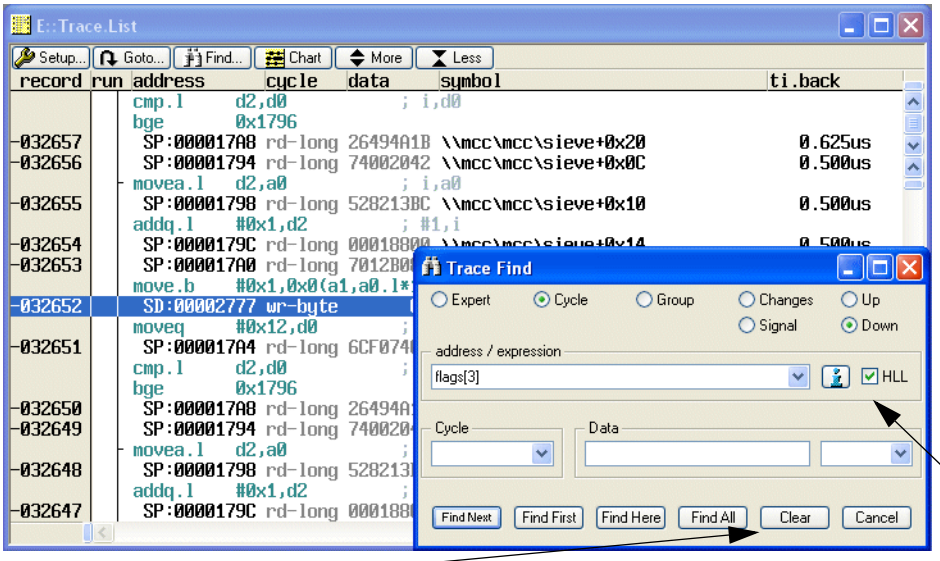
Find a Specific Record



Example 1: Find a specific Address



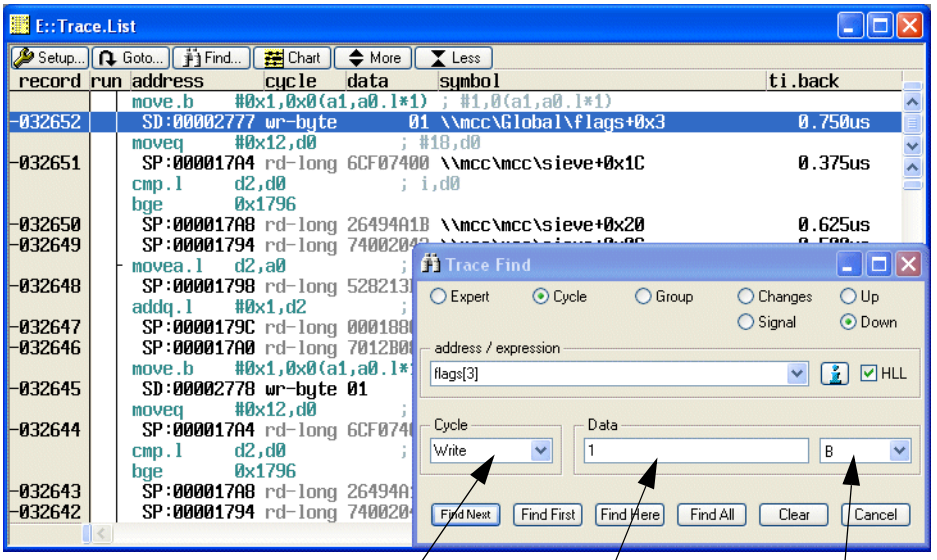
Example 2: Find a Variable



Push ***Clear*** to reset the settings in the Trace Find window

If the check box ***HLL*** is ON, you can look for any hll symbol

Example 3: Find the CPU Cycle where a specific Value is written to a specific Variable



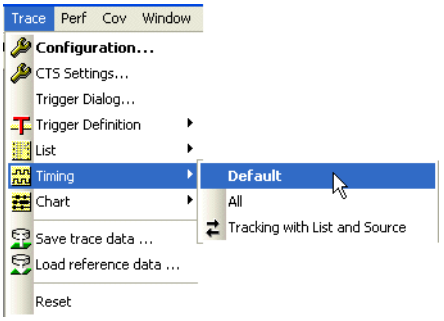
Define the data

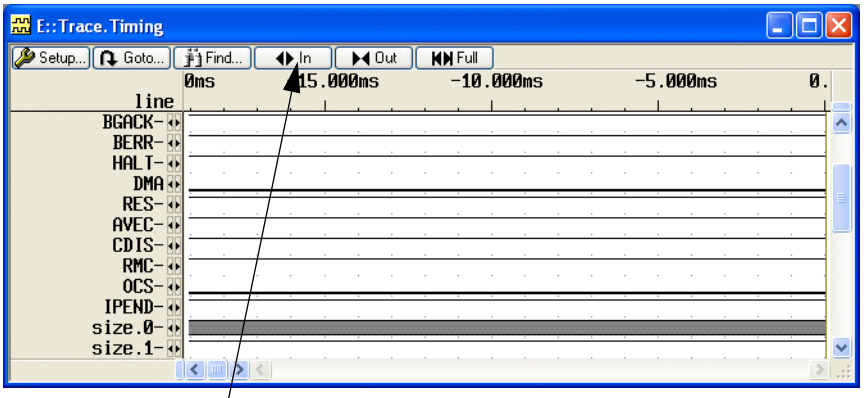
Define the CPU cycle

Define the data width (position on the data bus)

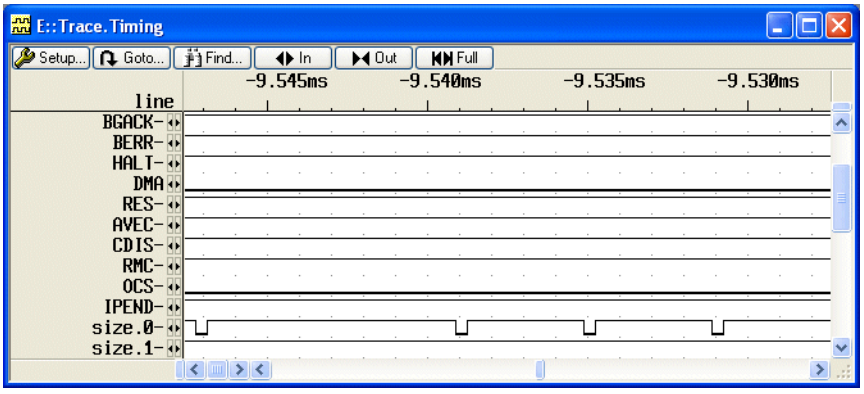
Trace.Find	Find a specific entry in the Trace

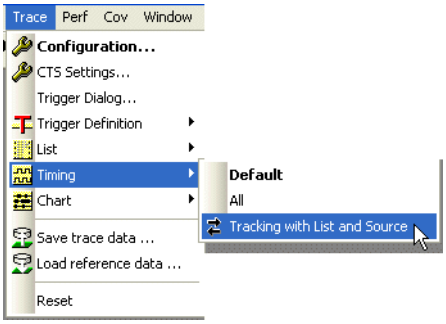
Trace.Timing

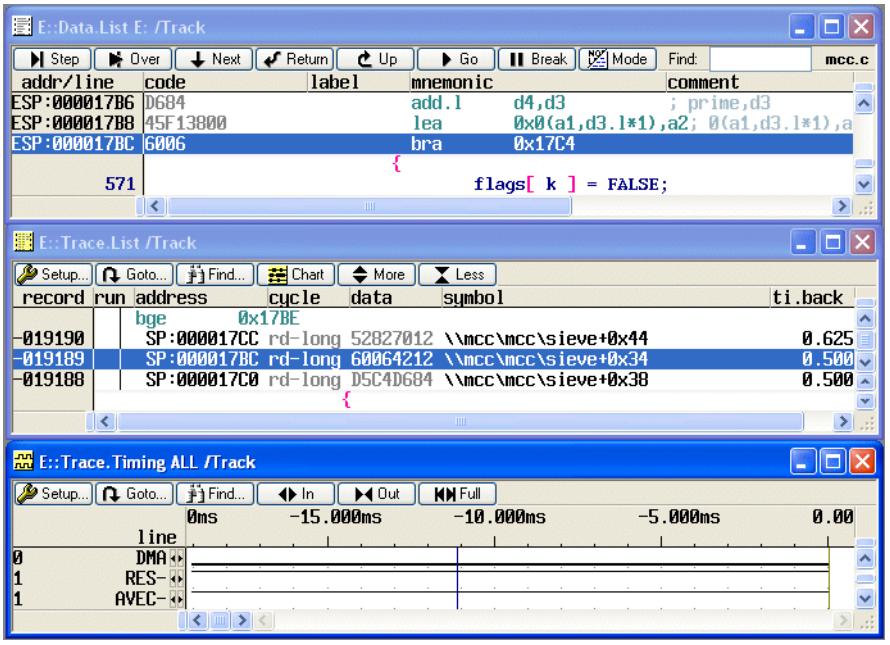




Use ***In*** to modify the horizontal scaling factor







All windows are opened with the ***/Track*** option.

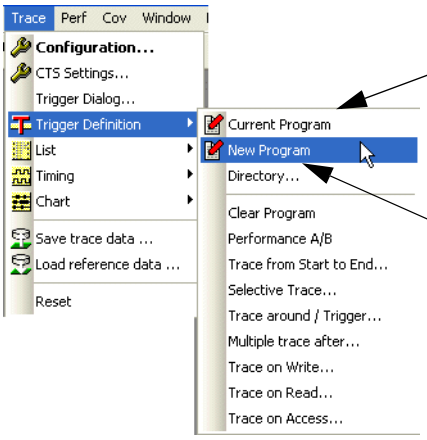
The cursor in the inactive window will always follow the cursor in the active window.

Trace.Timing	Display trace contents as timing diagram

Trigger Programming

Standard extension for trigger programs is **.TS**.

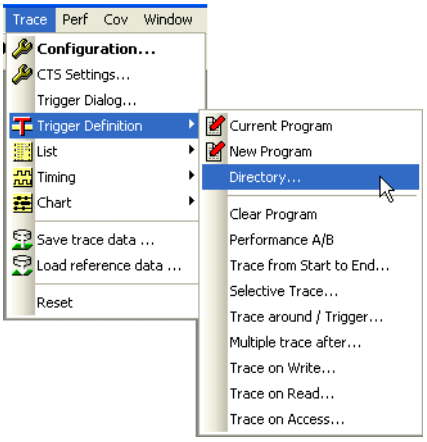
The Trigger Programming Window

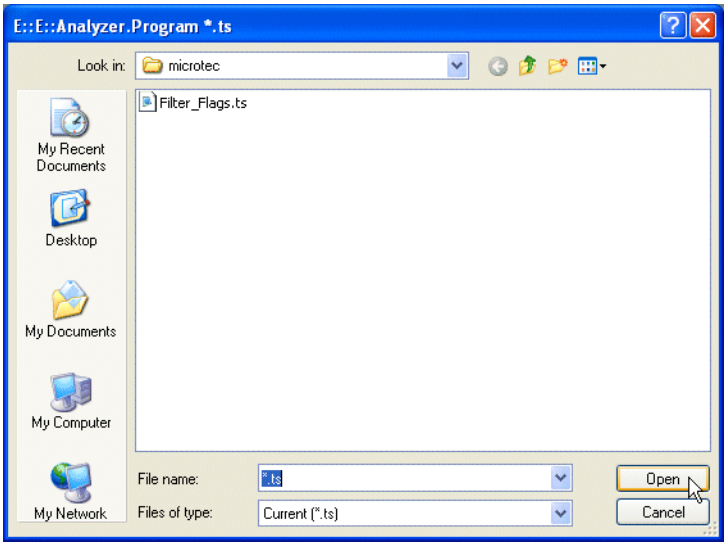


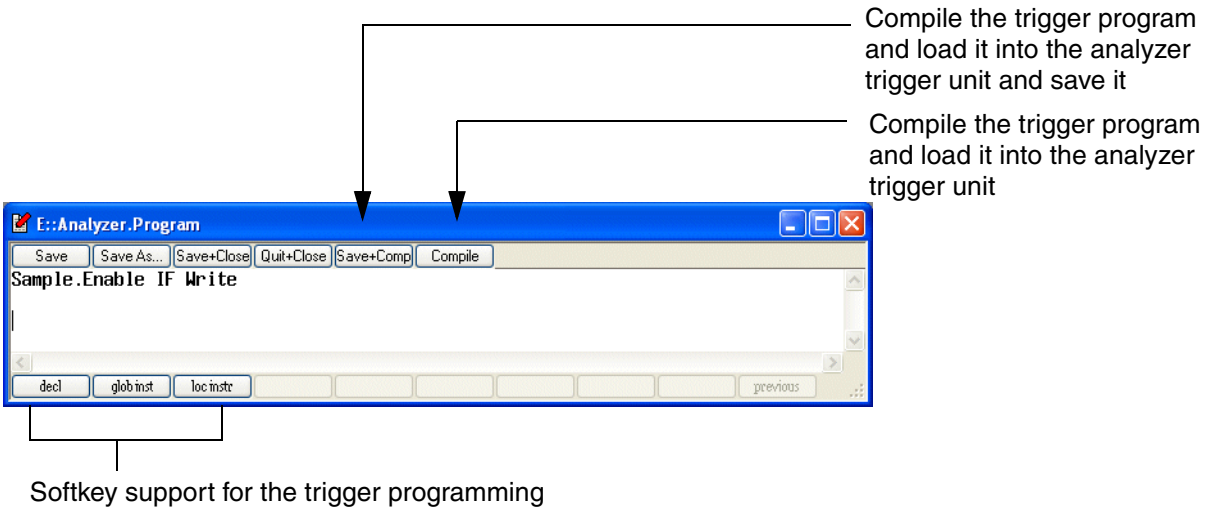
Open the programming window with the current active trigger program

Open the programming window to write a new trigger program

Open the programming window for an already existing trigger program.







Analyzer.Program	Open an Analyzer Program window to program the Analyzer Trigger Unit.
Analyzer.ReProgram	Program a ready-to-run trigger program into the Analyzer Trigger Unit.

Example:

```
Sample.Enable IF Write
```

Output command/ Action — **Condition**

Structure of instructions for the analyzer trigger unit:

<output_command> if <condition>

A condition is a combination of **input events**.

If no condition is defined, the output command is always executed.

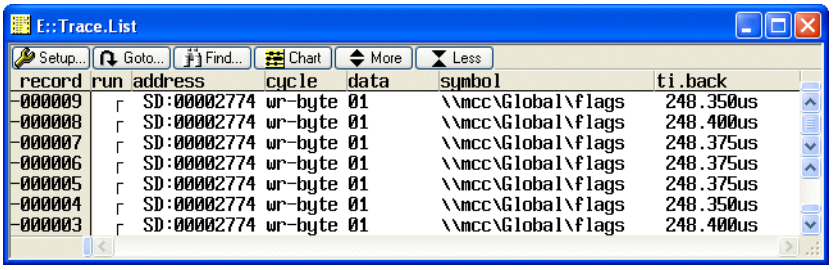
```
Sample.Enable
```

Output command

If a condition is defined, the output command is only executed when the condition is true.

Example: Sample all write accesses to the address flags

```
ADDRESS AlphaBreak flags

Sample.Enable IF AlphaBreak&&Write
```



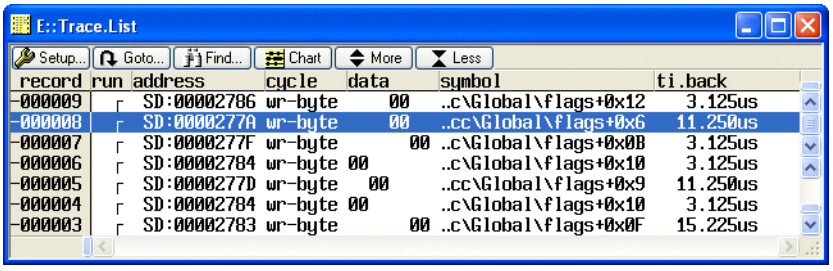
Trigger Conditions

The following logical operators are available to combine input events to a trigger condition:

AND	&&
OR	||
XOR	^^
NOT	!

Example: Sample all cycles where a 0 is written to the array flags

```
ADDRESS AlphaBreak V.RANGE(flags)
DATA.B Zero 0

Sample.Enable IF AlphaBreak&&Write&&Zero
```



```
ADDRESS AlphaBreak flags
EVentCouNTer ev1 5000.

Count.Enable ev1, Sample.Enable IF AlphaBreak
Trigger.A IF ev1&&AlphaBreak
```

Several output commands can be activated if a condition becomes true. They are separated by comma.

Structure of a Trigger Program

A trigger program has the following structure:

- Declarations
- Global Instructions
- Local Instructions

Declarations

Declarations define an input event which need to be declared (assign name and value to an input event).

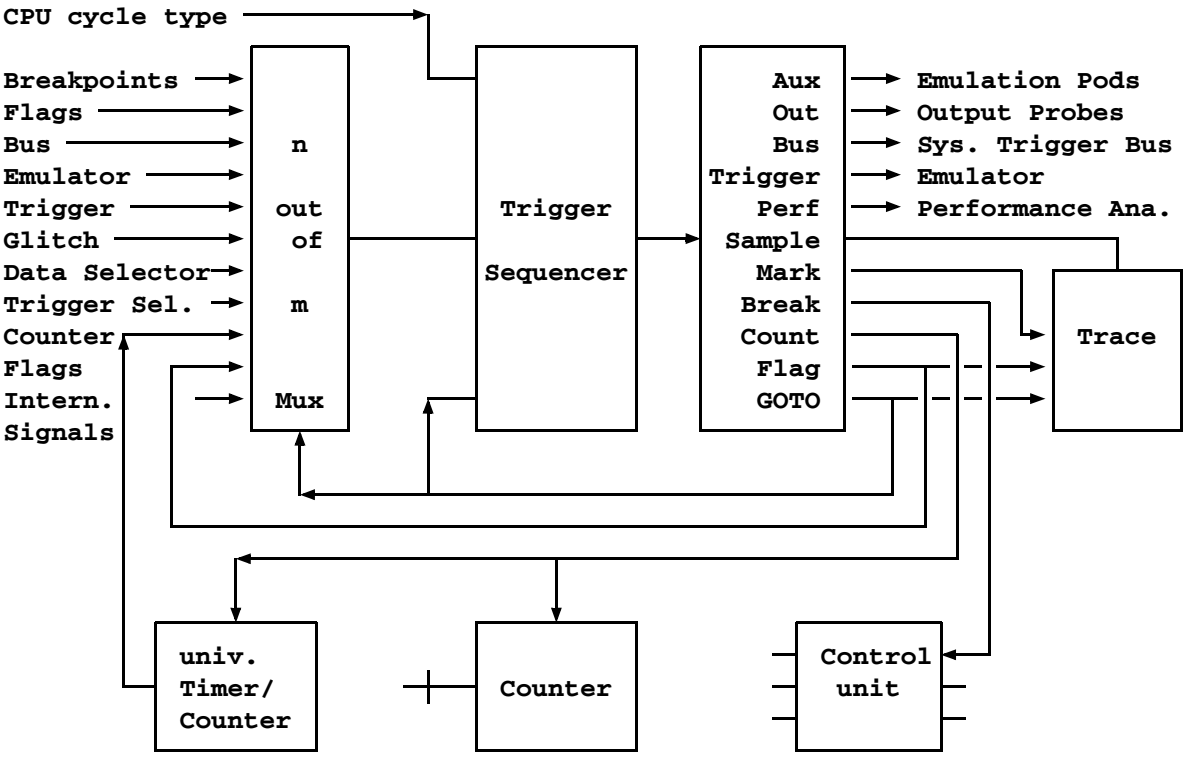
ADDRESS AlphaBreak flags

DATA.B zero 0

Global Instructions

Global Instructions are valid in all levels of a trigger program.

All global instructions are executed at the same time.

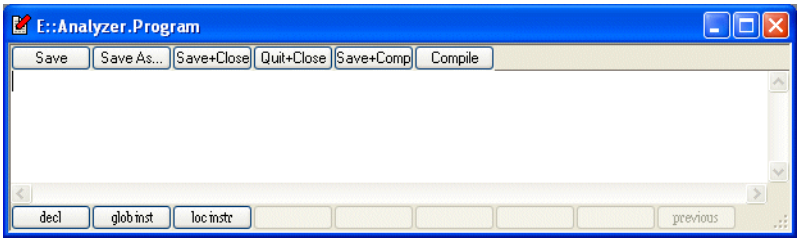


Hardware structure of the trigger unit

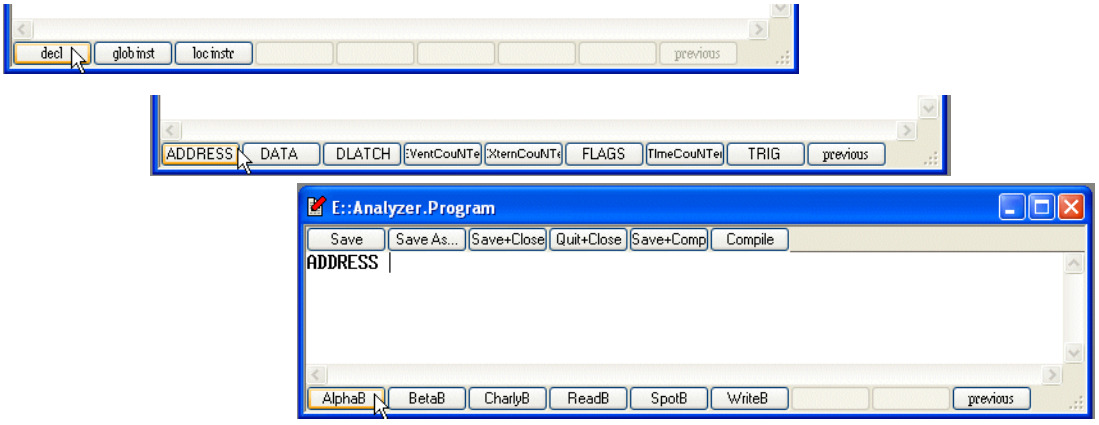
Local Instructions are assigned to a trigger level. They are only activated, when their trigger level is active.

HA120	8
HAC	2 (3)
ECC8	4

The softkey support for the trigger programming reflects this structure:



Declarations:



Selective Tracing

Output commands to control selective tracing:

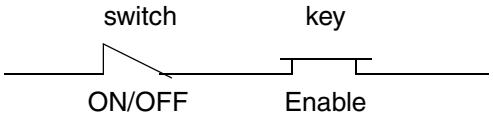
Sample.Enable [if <condition>]

Sample.ON [if <condition>]

Sample.OFF [if <condition>]

The instructions Sample.ON, Sample.OFF and Sample.Enable work as a controlled switch and a key in series.

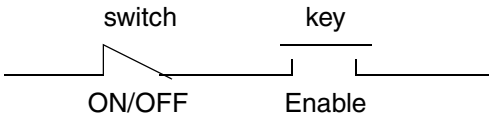
Example 1: No sample command in the trigger program



All cycles are sampled

Example 2: Only Sample.Enable commands, no Sample.ON/OFF in the trigger program

We recommend to use **Sample.enable** whenever possible**.**

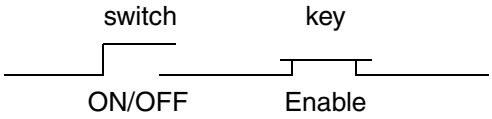


Sampling is controlled via the **Sample.enable** comm

Sample only write accesses to the variable flags

```
ADDRESS AlphaBreak V.RANGE(flags)

Sample.Enable IF AlphaBreak&&Write
```

Example 3: Only Sample.ON/OFF instructions, no Sample.Enable in the trigger program



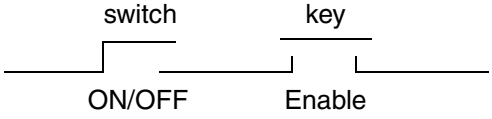
Sampling is controlled via the **Sample.ON/OFF** command

The initial state after RESet or Init is Sample.ON

Example: Sample only the function sieve

```
ADDRESS AlphaBreak P:0x1776          ; Call of sieve
ADDRESS BetaBreak P:0x17DA           ; Return from sieve

Level1:
     Sample.OFF                      ; Switch of sampling
     CONTinue
Level2:
     Sample.ON IF AlphaBreak         ; Start sampling (next cycle)
     Sample.OFF IF BetaBreak         ; Stop sampling
```

Example 4: Sample.ON/OFF instructions and Sample.Enable in the trigger program

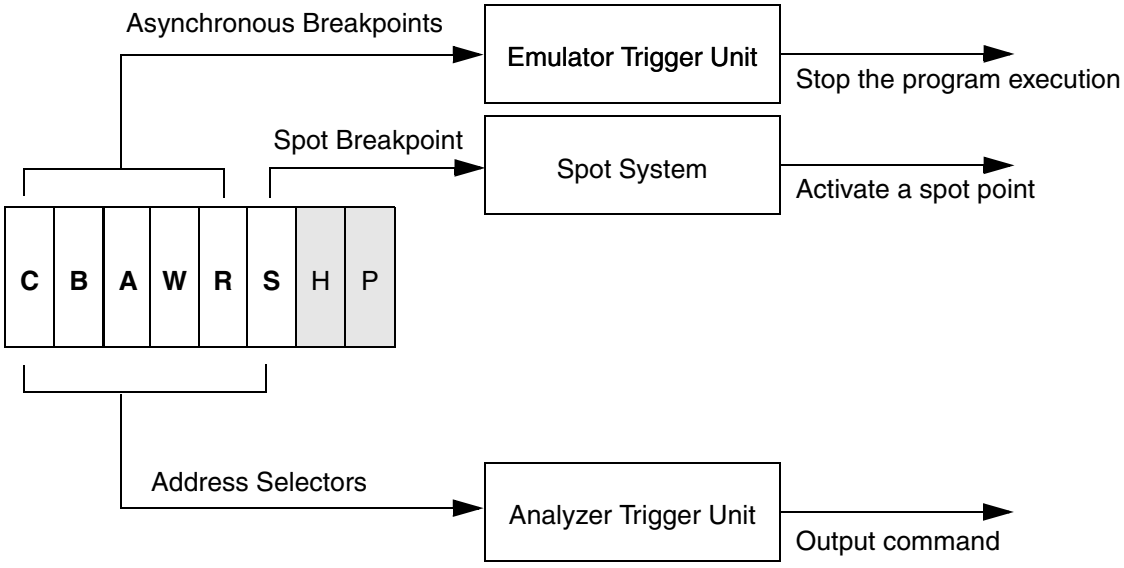


Sampling is controlled via the Sample.ON/OFF and Sample.Enable command

Address selectors are used to mark addresses that should be used e.g. for selective tracing.

The following address selectors are available:

AlphaBreak	HA120, ECC8
BetaBreak	HA120, ECC8
CharlyBreak	HA120, ECC8
SpotBreak	HA120
ReadBreak	HA120
WriteBreak	HA120

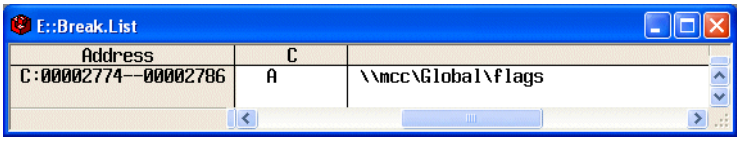


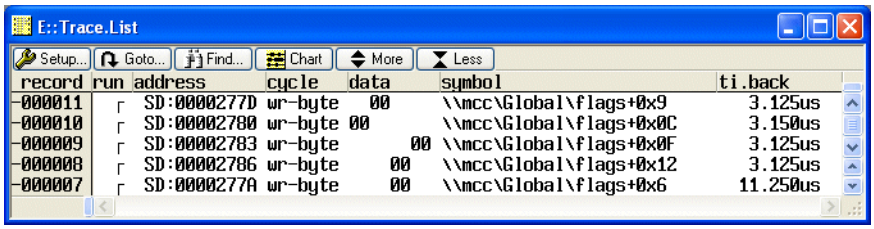
If a breakpoint is used as an address selector, it has to be disabled as a breakpoint.

Declaration: ADDRESS <breakpoint> <address>

It is not necessary to declare the address selectors, if the corresponding breakpoint is set outside of the trigger program.

```
ADDRESS AlphaBreak V.RANGE(flags)


Sample.Enable IF AlphaBreak&&Write
```





If a breakpoint type is used as an address selector, all set breakpoints of this type are deleted, when the trigger program is compiled.

More examples:

ADDRESS AlphaBreak 1000

ADDRESS AlphaBreak V.RANGE(flags)

ADDRESS AlphaBreak V.RANGE(flags[4])

ADDRESS AlphaBreak sieve func1 func2

ADDRESS AlphaBreak sieve||func1||func2 ;Identical to previous

Declaration: Data.<width> <name> <data>

Examples for available data selectors:

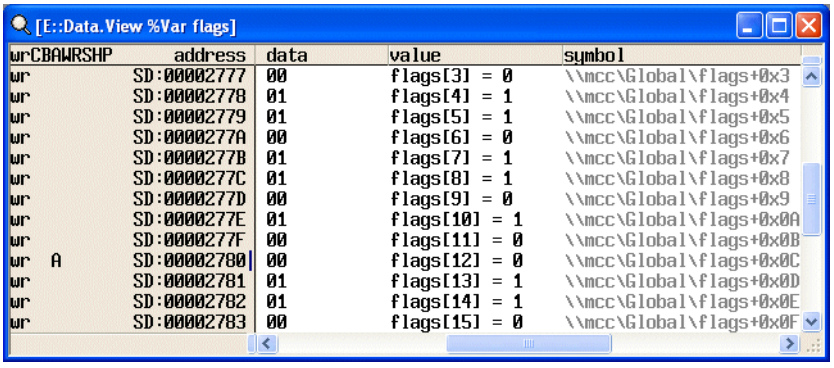
B0	Byte transfer, data positioned at D0--7
B1	Byte transfer, data positioned at D8--15
B	Byte (dynamic data selector), data positioned anywhere
W0	Word transfer, data positioned at D0--15
W1	Word transfer, data positioned at D15--D31
W	Word transfer (dynamic data selector), any alignment, any bus size
L	Long transfer

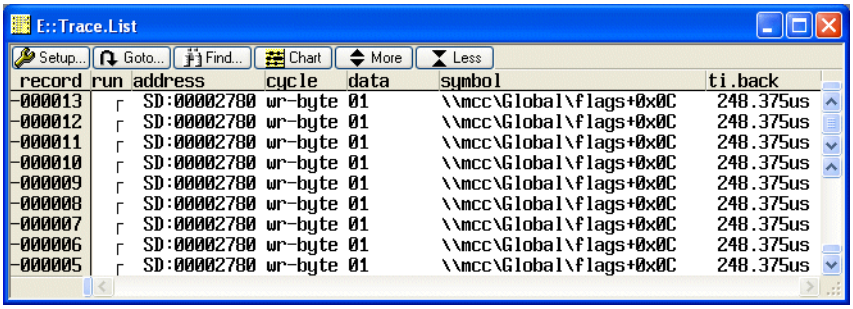
Available data selectors:

2 * 16 bit 1 * 32 bit on each trigger level	HA120
2 * 8 bit 1 * 16 bit on each trigger level	ECC8

Sample all cycles where a 01 is written to flags[12].

```
ADDRESS AlphaBreak V.RANGE(flags[12])
DATA.B one 01

Sample.Enable  IF AlphaBreak&&one&&Write
```





More examples:

Data.B Error_Val 0x0ff; hex. value

Data.B Error_Val 0x0xf; hex mask

Data.B Error_Val 0y0xxxx1xx0; binary mask

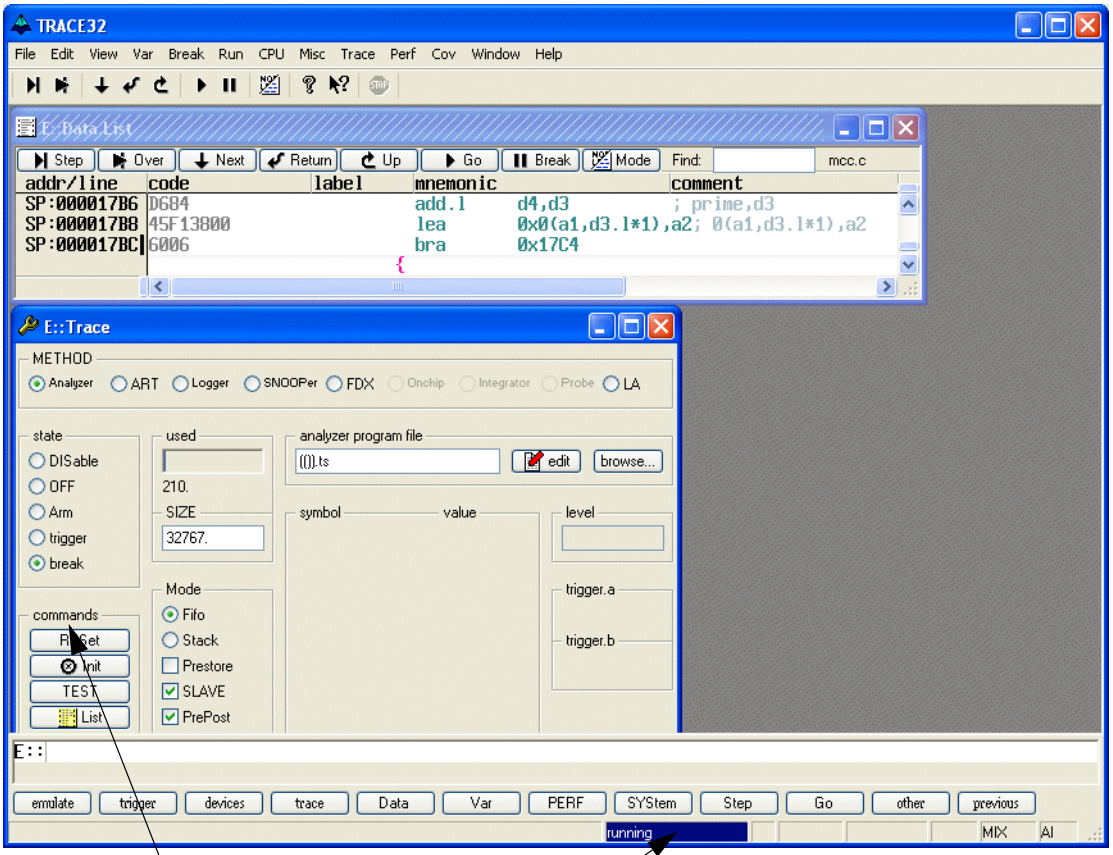
Data.B Error_Val 11 22 33 ; one of the 3 values

Break IF <condition>

Stops the recording in the trace buffer and deactivates the analyzer trigger unit at the specified condition.

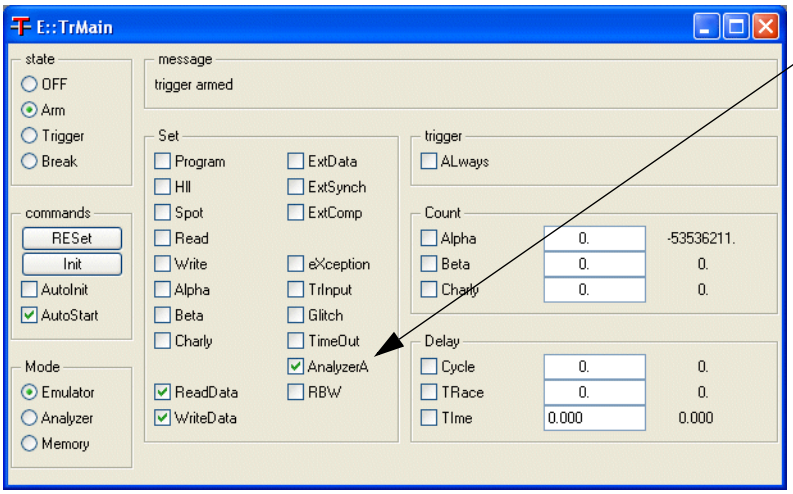
Example: Stop the trace when 0 is written to the array flags

```
ADDRESS AlphaB v.range(flags)
DATA.B Zero 0

BREAK IF AlphaB&&Write&&Zero
```



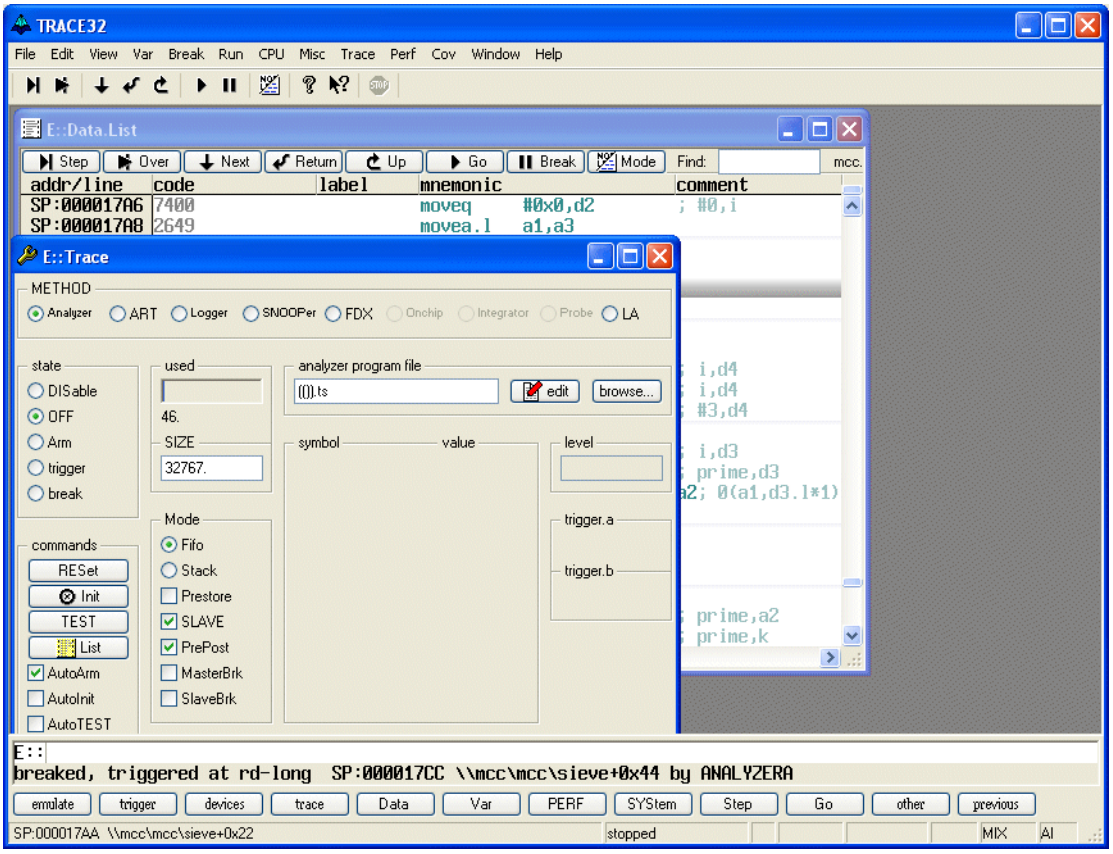
Trigger.A IF <condition>

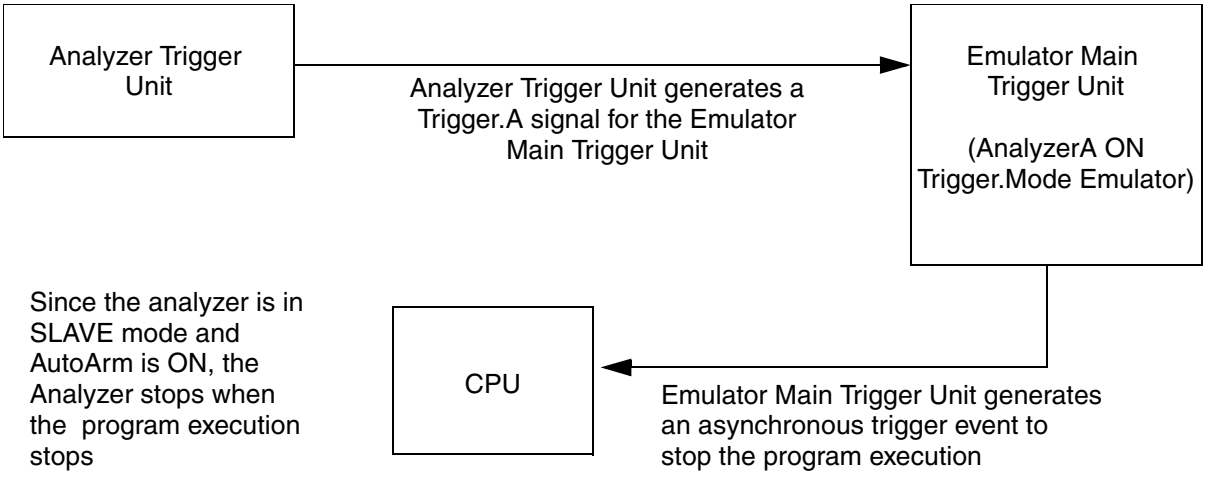
Stop the program at the specified condition



The program execution can only be stopped by an Trigger.A command if AnalyzerA is set in the Emulator Main Trigger Unit

```
ADDRESS AlphaBreak V.RANGE(flags)
DATA.B Zero 0

Trigger.A IF AlphaBreak&&Write&&Zero
```





Available event counters:

HA120	3 * 48 bit restart cap.	Max. Count 5.5e11
ECC8	2 * 16 bit or + 1 * 32 bit restart cap.	Max. Count 65535 or 4.2e9
HAC	2+1	65535

Declaration: EVentCouNTer <name> [<event>]

Instructions:

Counter.Increment <name> [if <condition>];Counter is incremented whenever the condition is true

Counter.ON <name> [if <condition>];Counter is incremented each CPU cycle until Counter.OFF

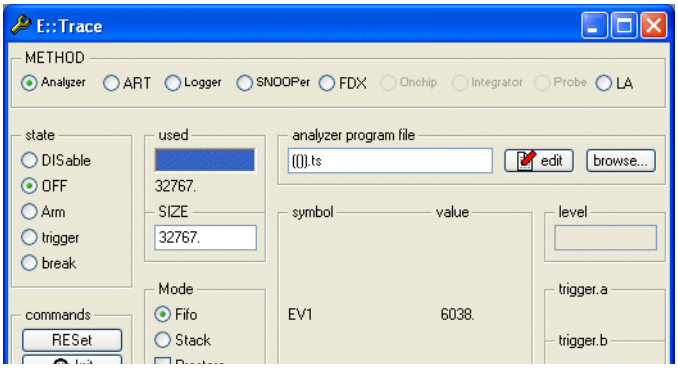
Counter.OFF <name> [if <condition>]

Counter.Restart <name> [if <condition>];Counter is reset to 0

Example 1

Counter ev1 counts how often something is written to flags[4].

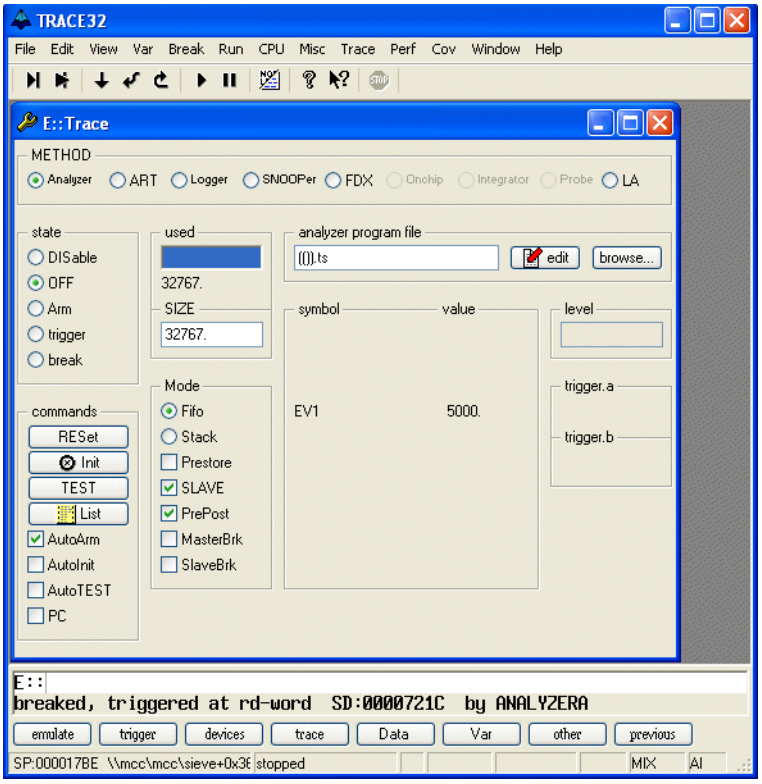
```
ADDRESS AlphaBreak V.RANGE(flags[4])
EVentCouNTer ev1

Counter.Increment ev1 if AlphaBreak&&Write
```



The program execution is stopped, after 5000. write accesses to the variable flags.

```
ADDRESS AlphaBreak V.RANGE(flags)
EVentCouNTer ev1 5000.

Counter.Increment ev1 IF AlphaBreak&&Write
Trigger.A IF ev1
```

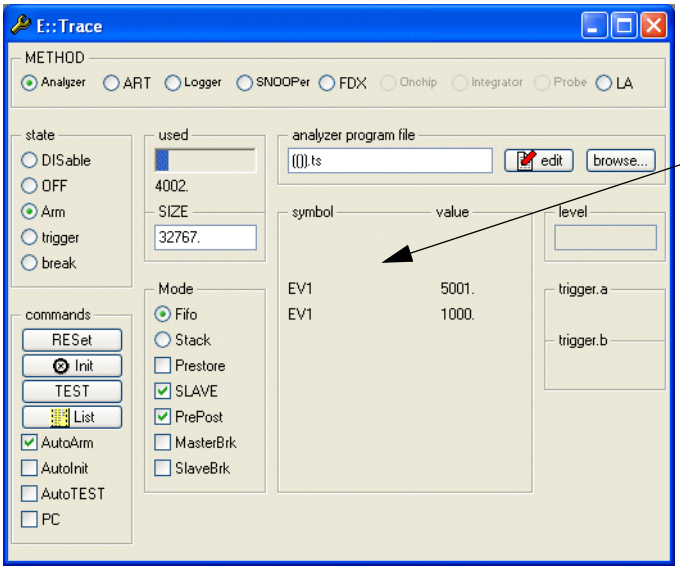
The input event <counter name> is **TRUE** when the counter has reached its final value.



The trace buffer samples the 1000. up to the 5000. write access to flags[4].

```
ADDRESS AlphaBreak V.RANGE(flags[4]
EVentCouNTer ev1 1000.--5000.

Counter.Increment ev1 IF AlphaBreak&&Write
Sample.Enable IF ev1&&AlphaBreak&&Write
```

The input event <counter name> is **TRUE** when the counter value is within the specified counter range.



If the counter value is a range, t
the analyzer trigger unit needs
2 counters

Available time counters:

Event counter and time counter share the same resources.

HA120	**3 * 48 bit Restart cap.**	**Max. time 3.8 hours**	**Resolution 100 ns**
ECC8	**2 * 16 bit or 1 * 32 bit restart cap.**	**Max. time 6.5 ms or 7.1 min**	**Resolution 100 ns**
HAC	**1**	**65 ms**	**Resolution 100 ns**

Time values can be entered in the following units:

Nanoseconds (ns)

Microseconds (µs)

Milliseconds (ms)

Seconds (s)

Kiloseconds (ks)

Declaration: TImeCouNTer <name> [<time>]

Commands:

Counter.Increment <name> [if <condition>];

Counter.ON <name> [if <condition>]

Counter.OFF <name> [if <condition>]

Counter.Restart <name> [if <condition>]

Stop the program execution, if the function sieve need longer, then 250.us.

```
ADDRESS AlphaBreak sieve
ADDRESS BetaBreak P:0x17DA    ; end of sieve
TImeCouNTer sievec 250.us

start:
     Counter.OFF sievec
          CONTinue

     level1:

     Counter.ON sievec IF AlphaBreak
     Counter.Restart sievec IF BetaBreak
     Trigger.A IF sievec&&!BetaBreak
```

Available trigger levels:

HA120	**8**
ECC8	**4**
HAC	

- A trigger level start with a label
- A trigger level ends at the next label or with the end of the program
- Only one trigger level can be active
- The first level, that is active, when a trigger program is started is the level named “start”. If no level “start” is used in the trigger program, the first level is the active level.

Commands to change the trigger levels:

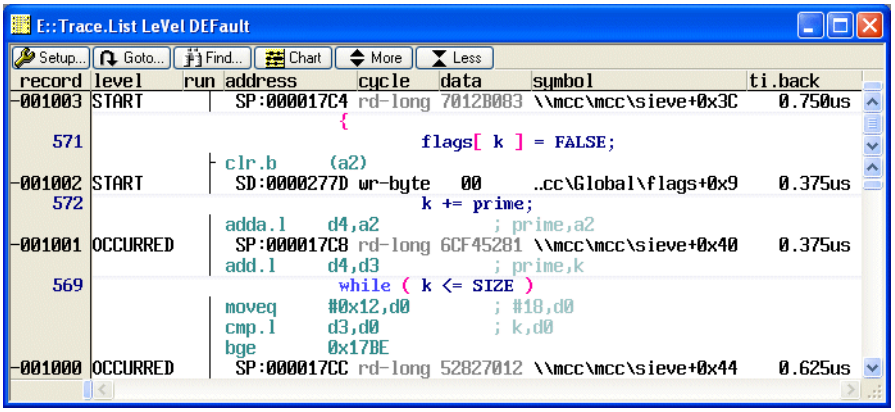
CONTinue - sequential level switch, continue with the next trigger level

GOTO <label> - free level change

Sample another 1000. records to the trace buffer, after the first write access to the variable flags.

```
ADDRESS AlphaBreak V.RANGE(flags)
EVentCouNTer delay 1000.

start:
     CONTinue IF AlphaBreak&&Write

occured:
     Counter.Increment delay
     BREAK IF delay
```

```
Trace.List LeVel DEFault
```



Available markers:

HA120	**A, B, C**
ECC8	**A, B**
HAC	

Commands for markers:

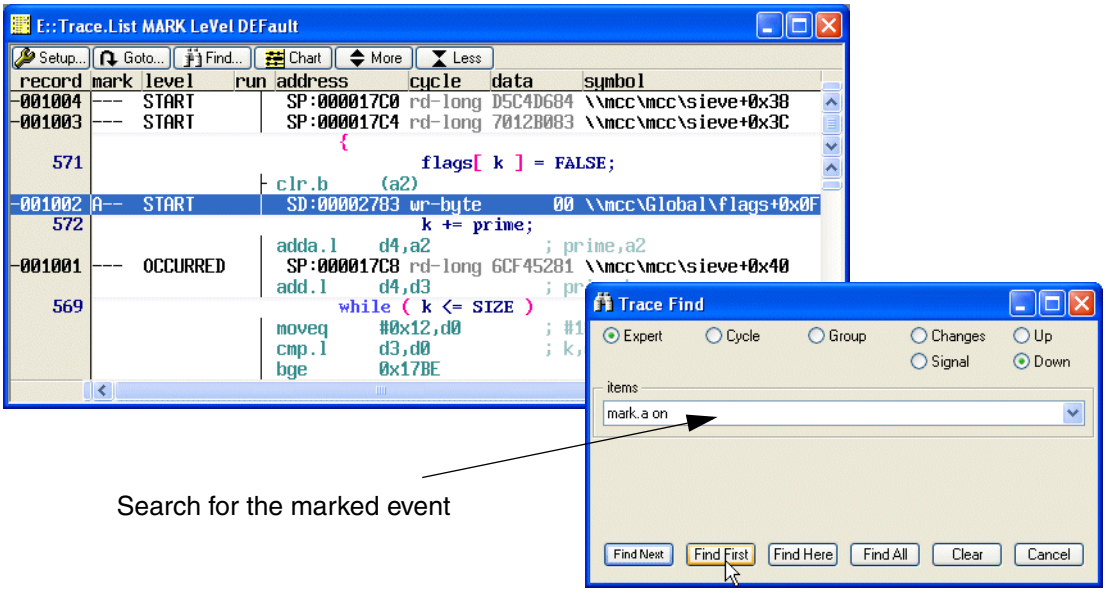
Mark.<name> [if <condition>]

Example 1

Sample another 1000. records to the trace buffer, after the first write access to the variable flags and mark this write access in the trace buffer.

```
ADDRESS AlphaBreak V.RANGE(flags)
EVentCouNTer delay 1000.

start:
     Mark.A IF AlphaBreak&&Write
     CONTinue IF AlphaBreak&&Write

occured:
     Counter.Increment delay
     BREAK IF delay
```

```
Trace.List MARK LeVel DEFault
```



Search for the marked event

Sample only the function sieve.

```
ADDRESS AlphaBreak sieve
ADDRESS BetaBreak P:0x17DA

start:
     Mark.A IF AlphaBreak
     Sample.Enable IF AlphaBreak
     CONTinue IF AlphaBreak

infunc:
     Sample.Enable
     Mark.B IF BetaBreak
     GOTO start IF BetaBreak
```

	HA120	HAC	ECC8
Trigger Inputs	2 channels / 2 * 8 bit 1 coax input A	1 channels / 12 * 8 bit	1 channel / 1*8 bit 1 coax input A

Declaration: TRIG.A <name> <data>;Input TRIGGER A

TRIG.B <name> <data>;Input TRIGGER B

INOUTA;Coax input A

```
TRIG.A   T_SEL1 0y0xxxx11xx

BREAK        IF        T_SEL1


BREAK        IF        INOUTA
```

More examples:

TRIG.A Error_Val 0x0ff; hex. value

TRIG.A Error_Val 0x0xf; hex mask

TRIG.A Error_Val 0y0xxxx1xx0; binary mask

TRIG.A Error_Val 11 22 33 ; one of the 3 values

	HA120	HAC	ECC8
Trigger Outputs	1 coax output A 2 outputs (C, D)		1 coax output A 2 outputs (C, D)

Commands:

OUT.<pin> [if <condition>]

Example

Stimulate coax output A if function sieve is entered.

```
ADDRESS AlphaBreak sieve

OUT.A        IF        AlphaBreak
```

Stimulate coax output A for 10.ms every time when the function sieve is entered.

```
ADDRESS AlphaBreak sieve
TImeCouNTer C_Delay 10.ms

start:
     Counter.Restart C_Delay
     CONTinue IF AlphaBreak
level1:
     OUT.A IF !C_Delay
     Counter.Increment C_Delay
     GOTO start IF C_Delay
```

Rule 1:
Instructions are executed from left to right.

```
Counter.ON count1, Counter.OFF count1 IF AlphaBreak
```

Rule 2:
Instructions are executed top down.

```
GOTO Count_Level
GOTO Error_Level IF INOUTA

Count_Level:
.
Error_Level:
.
.
```

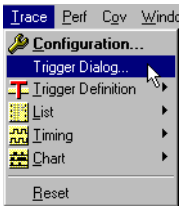
Rule 3:
Global statements have lower priority than local statements. (see Rule 2)

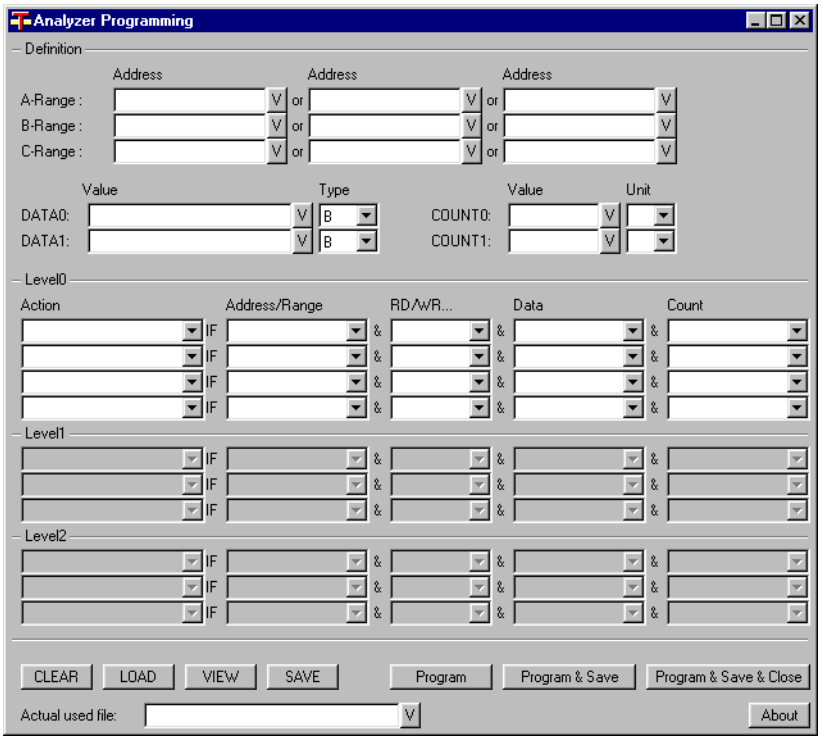
Rule 4:
Commands that change the trigger level (CONTinue, GOTO) can not be combined with other output commands. This rule applies also to the BREAK command.

The Analyzer Programming Dialog Window

- The intension of the Analyzer Programming Dialog Window is to provide an intuitive and easy to use interface to program the Analyzer Trigger Unit.
- The Analyzer Programming Dialog Window does not represent the full functionality of the Analyzer Trigger Unit.

How to Start





First Example

Example:

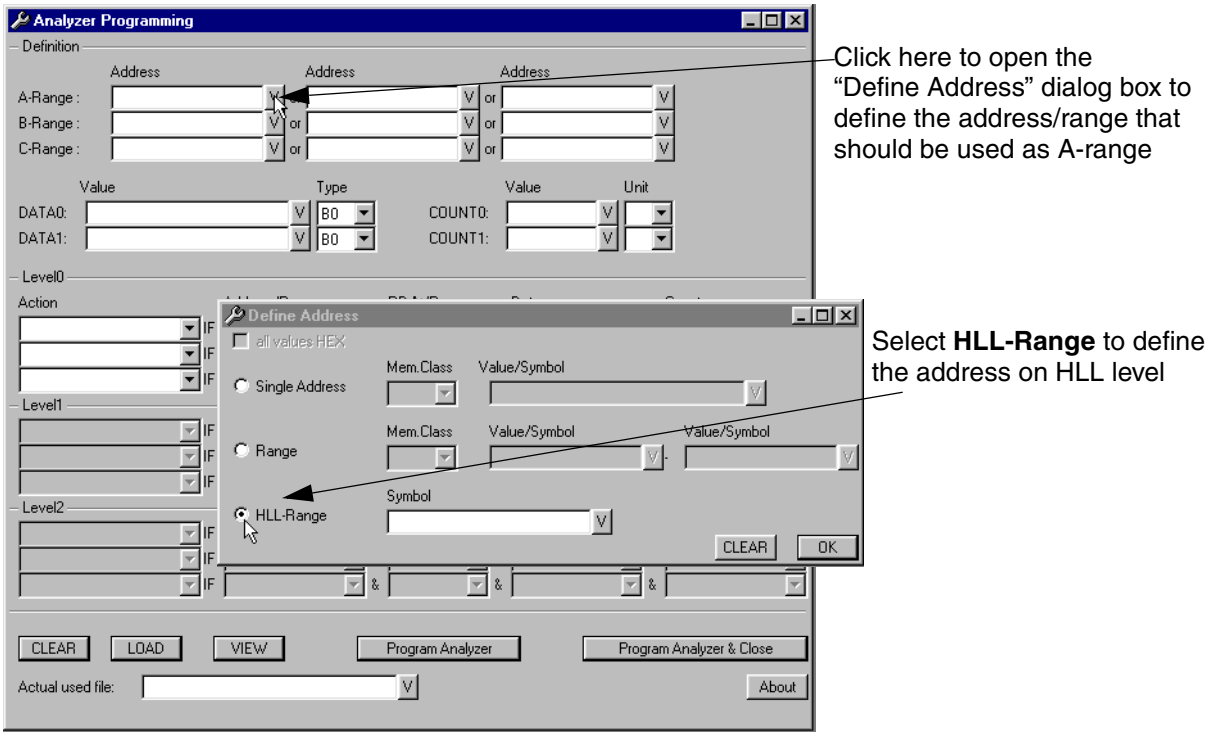
Sample all write accesses to the variable flags.

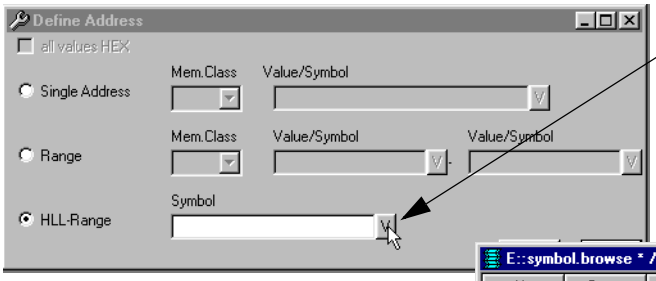
1. Define the address range (here the address range of the variable flags).
2. Select the action (here selective tracing) and define the condition on which the action should be executed (here write accesses to the address/range defined in step 1).

Define the Address Range

The Analyzer Trigger Dialog Box allows the definition of 3 different address ranges, that can be used by a Trigger Program:

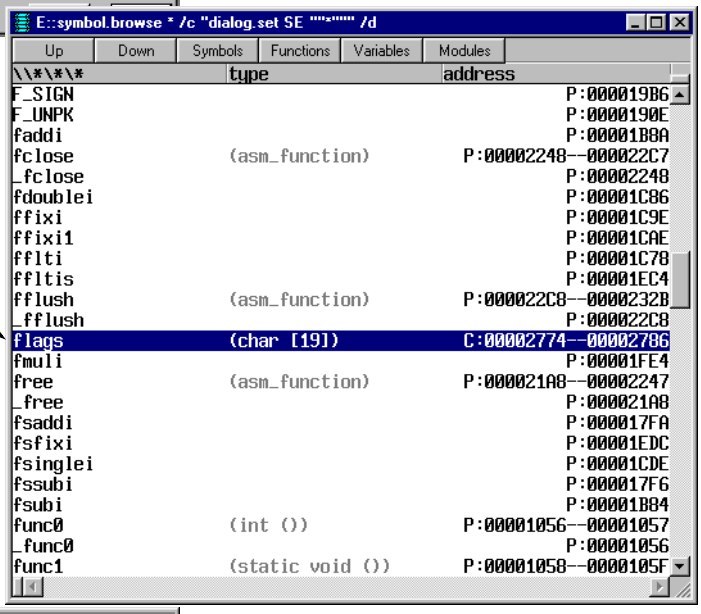
A-Range, B-Range, C-Range

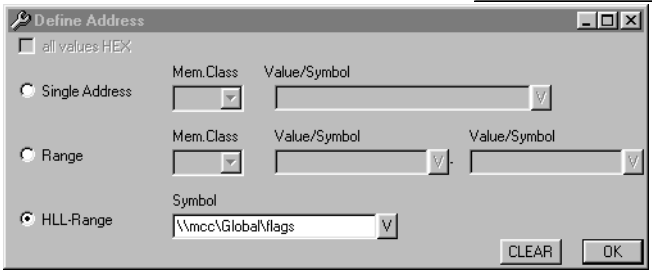




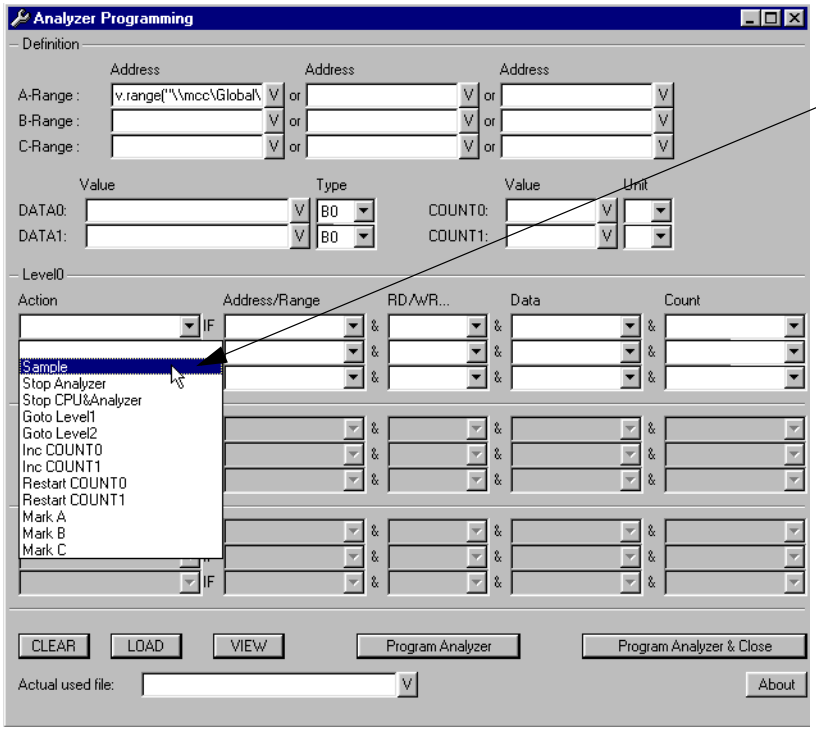
Click here to open the browser window for all HLL symbols

Select the variable flags by a double click
(fast browing through first letters possible)





An Analyzer Trigger Program can execute **Actions** depending on the current state of the user program and the target hardware.

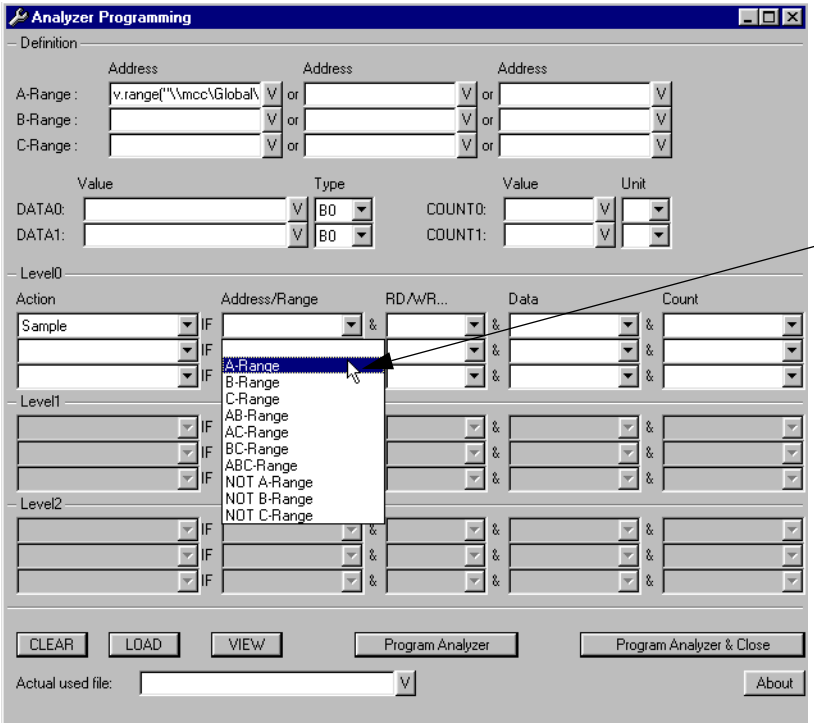


Select the action **Sample** to do a selective trace

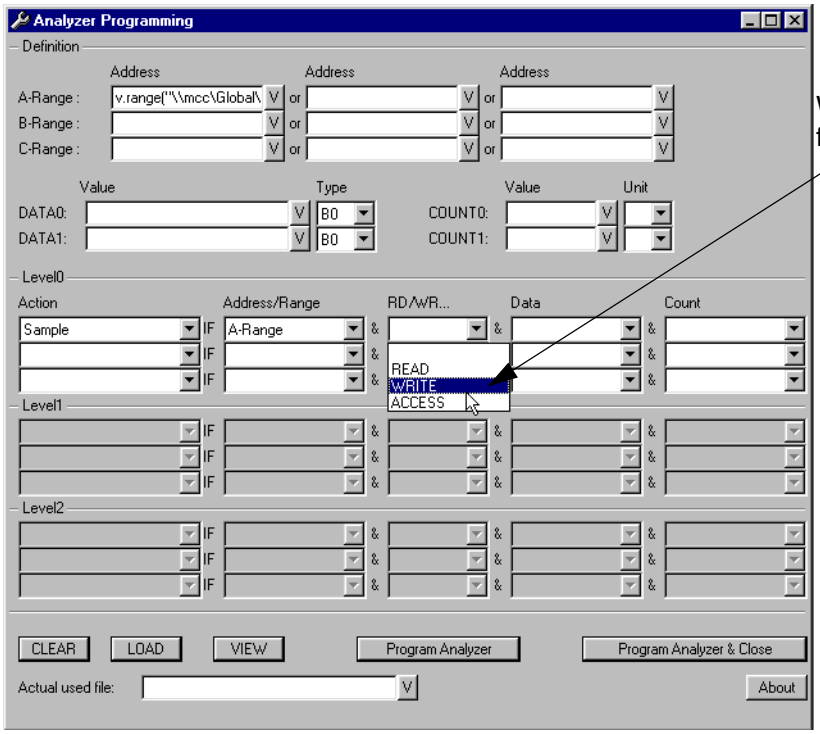
The current state of the user program/target hardware can be described by different characteristics. The different characteristics can be combinded by a LOGICAL AND to form the condition on which the action should be executed.

The following characteristics can be combined to form a condition:

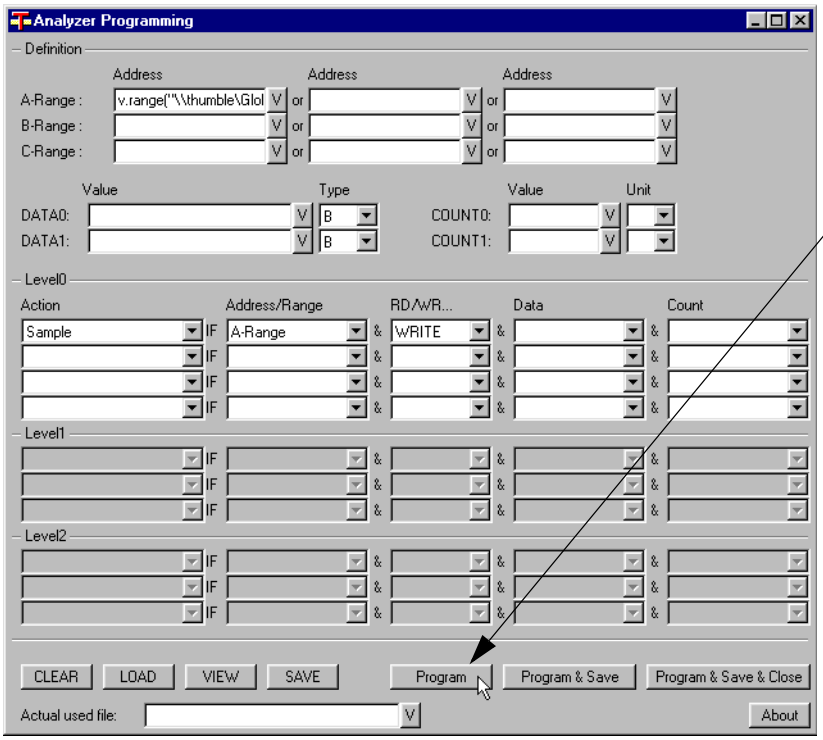
Address/Range	Address on the address bus
RD/WR	Cycles type Read access, write access, any access …
Data	Data on the data bus
Count	State of an event or time counter



The address/range defined by A-Range is the characterisic for address bus

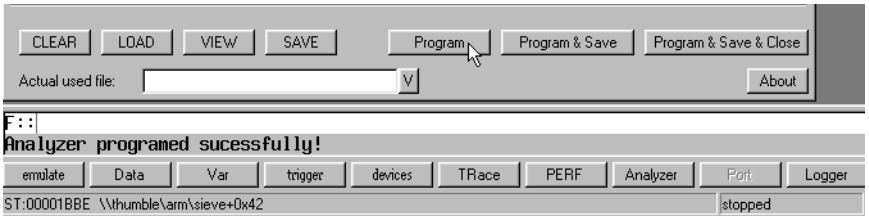


Write access is the characteristic for the cycles type

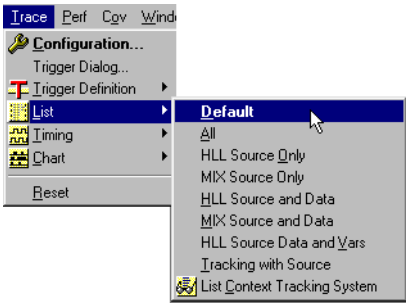


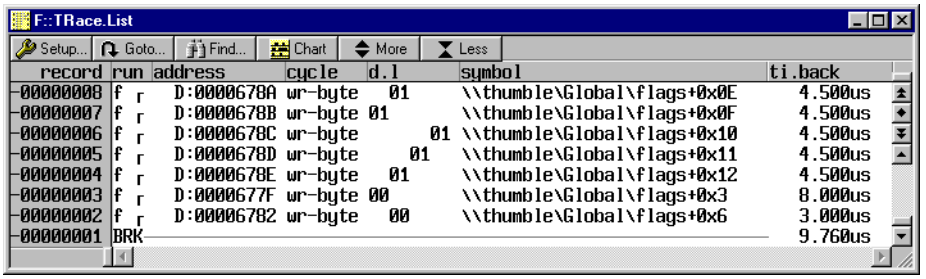
If the Analyzer Program is ready click here to program it into the Analyzer Trigger Unit

If the Analyzer Program is correct you get the message “Analyzer programmed successfully!”



Start and stop the user program.

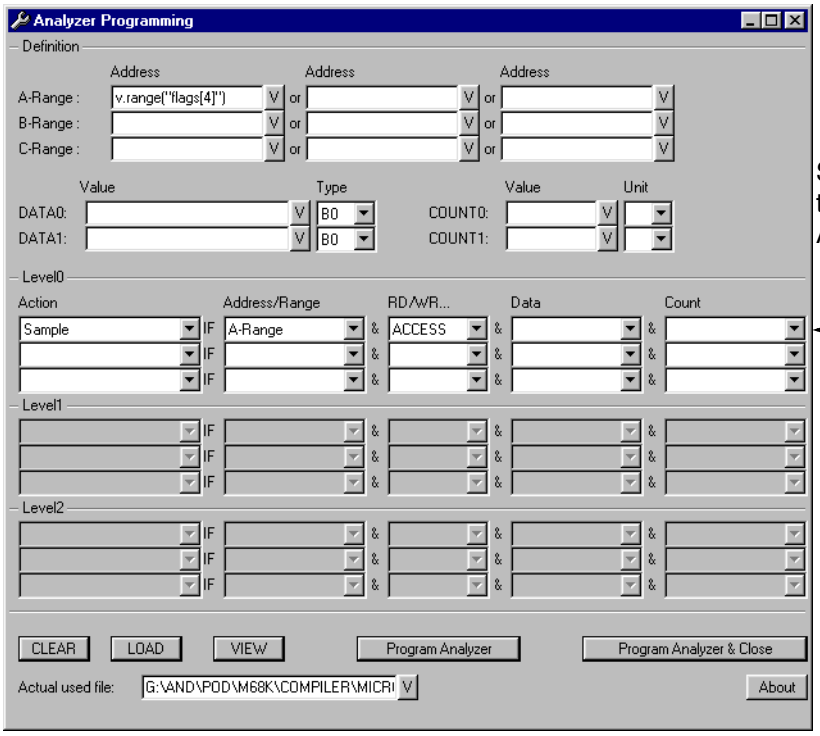




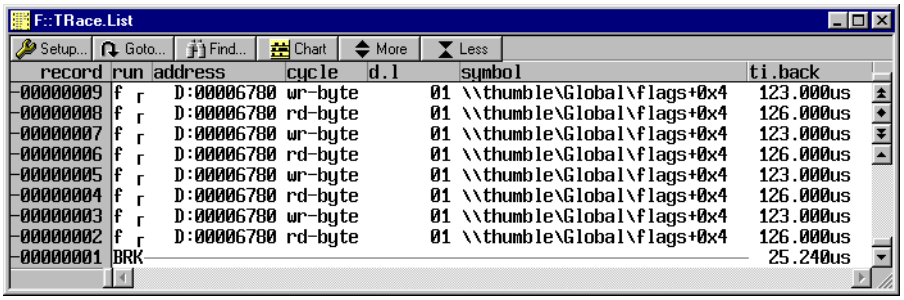
Sample any Access

1. Use A-Range to define the memory location/variable.
2. Select the action Sample and specify the condition by using the characteristic Address/Range and RD/WR…

Example: Sample all accesses to flags[4].



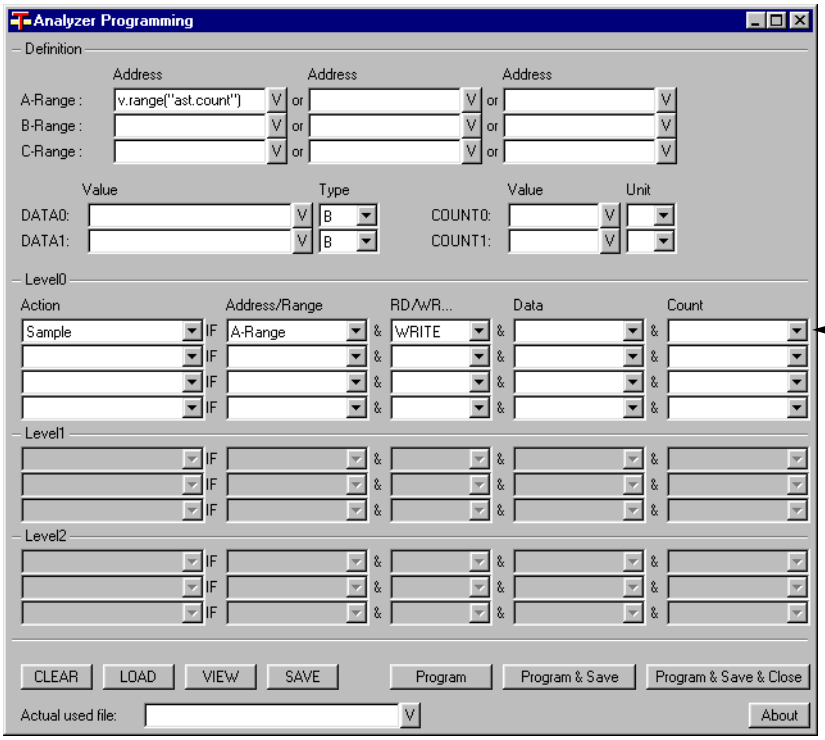
Sample all read and write cycles to the Address/Range defined by A-Range



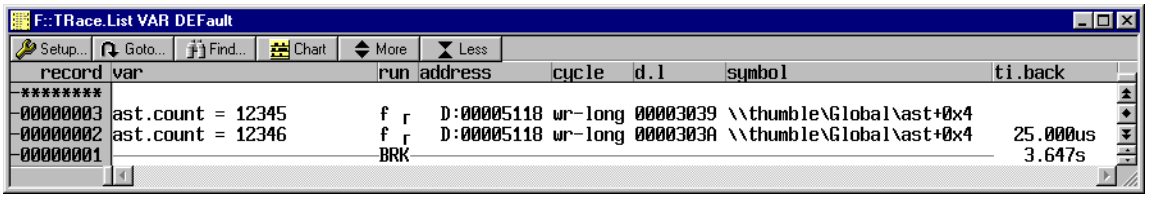
Sample all Write Accesses

1. Use A-Range to define the memory location/variable.
2. Select the action Sample and specify the condition by using the characteristic Address/Range and RD/WR…

Example: Sample all write accesses to ast.count.



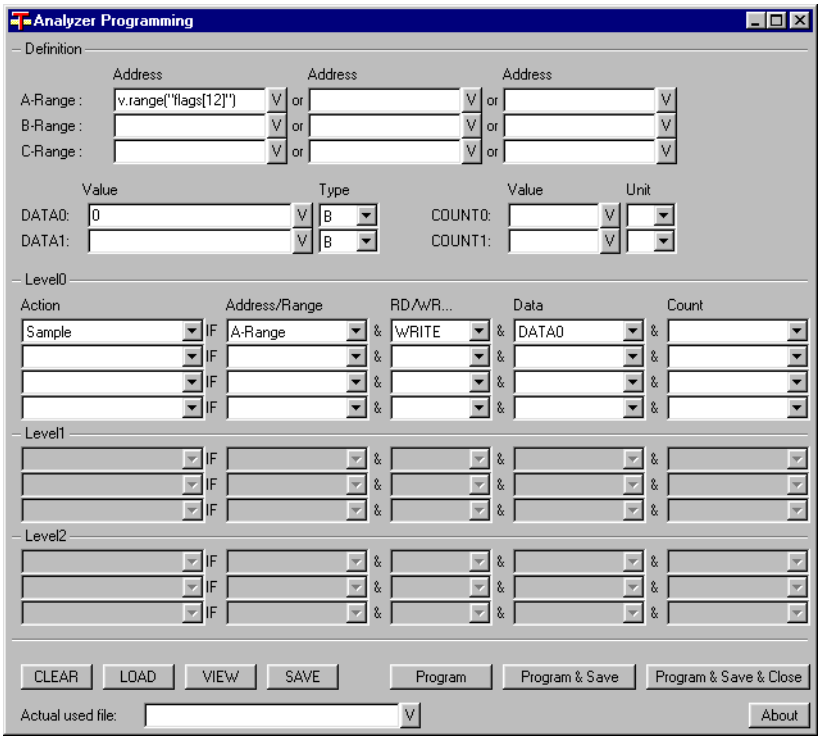
Sample all write cycles to the Address/Range defined by A-Range

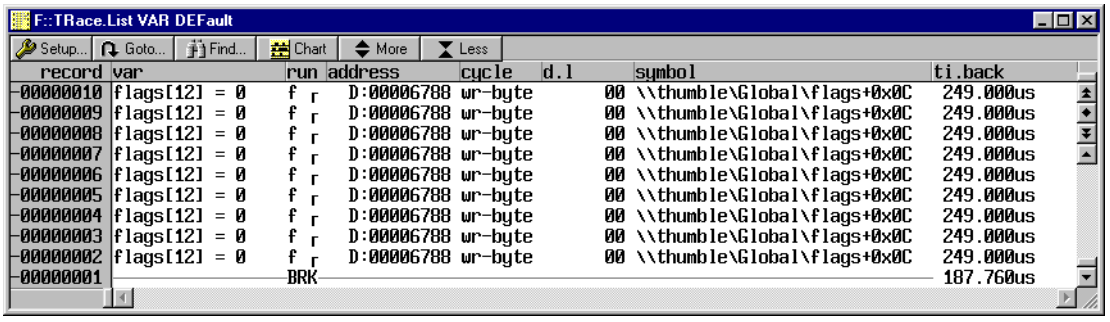


1. Use A-Range to define the memory location/variable.
2. Use DATA0 to define the data value.

   To define a data value it is necessary to know if the data value is transferred as a byte, a word or a long.
3. Select the action Sample and specify the condition by using the characteristic Address/Range and RD/WR… and DATA.

Example 1: Sample all cycles where a 0 is written to flags[12]





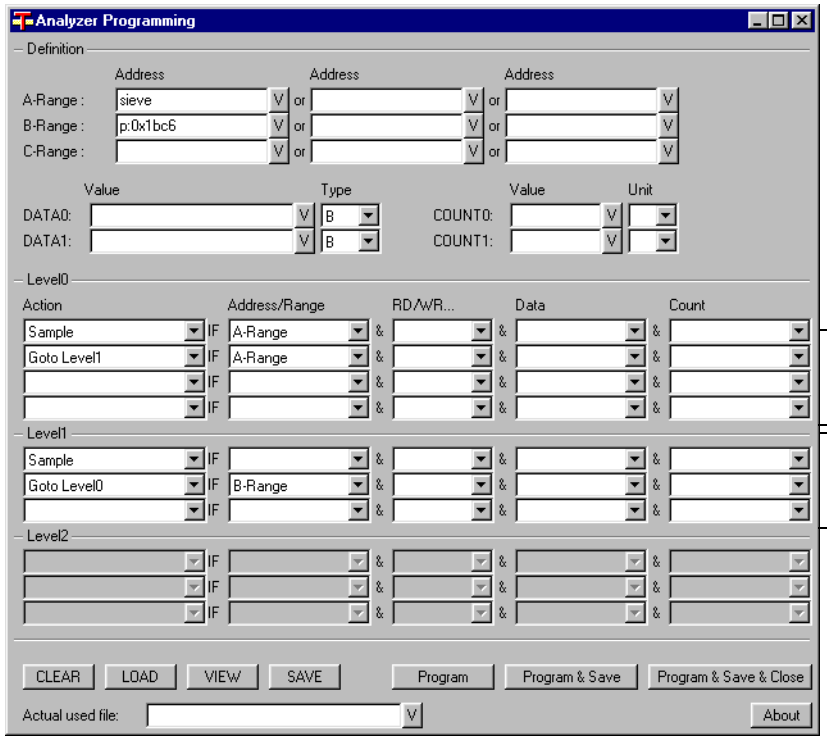
Sample a Function

1. Use A-Range to define the function entry.
2. Use B-Range to define the function exit.
3. Select a set of actions by using 2 trigger levels.

   In the trigger level Level0 the analyzer waits for the function entry and changes to Level1 when the function is entered.

   In the trigger level Level1 the analyzer samples all cycles and changes back to Level0 at the function exit.

Example 1: Sample only the function sieve.



Level0:
Sample only the function entry

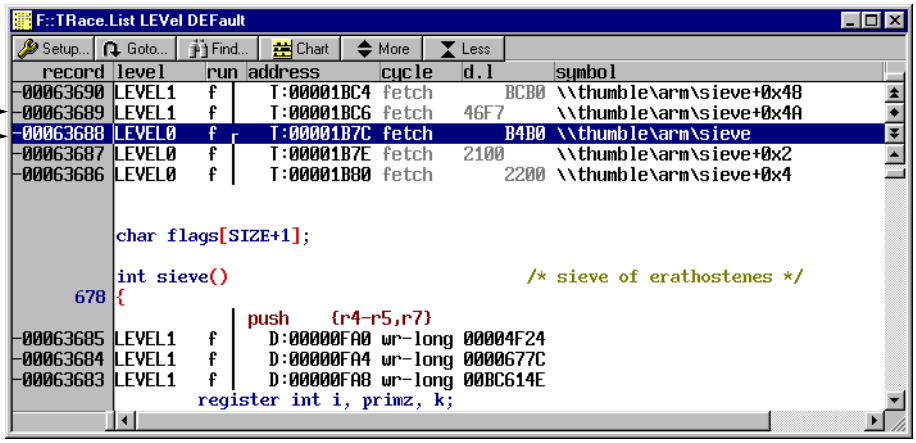
Change to Level1 at the function entry

Level1:
Sample all cycles

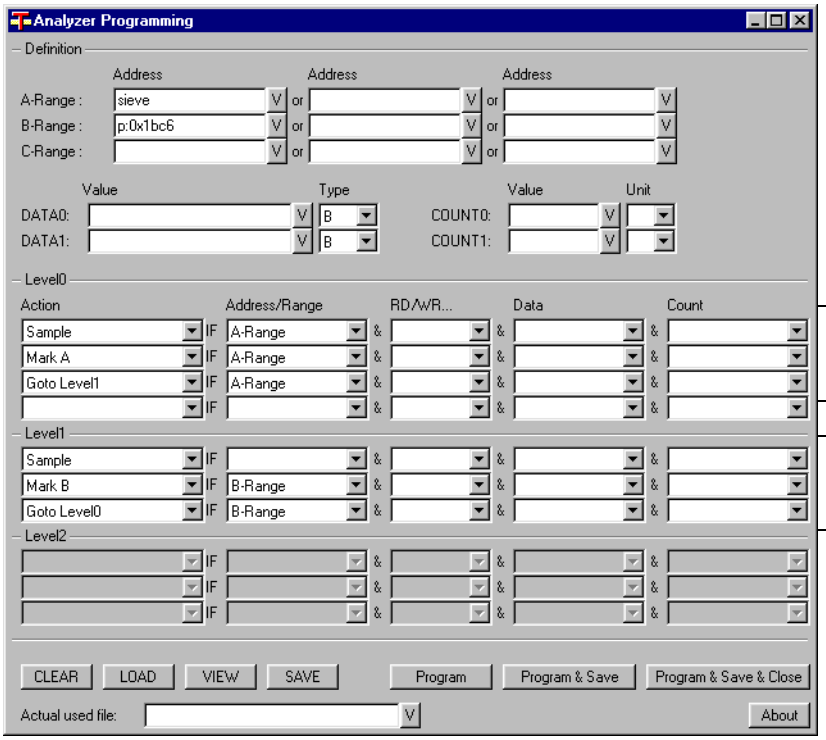
Change to Level0 at the function exit

Function exit

Function entry



Example 2: Sample only the function sieve and mark the function entry with an A marker and the function exit with a B marker.



Level0:
Sample only the function entry and mark it with an A marker

Change to Level1 at the function entry

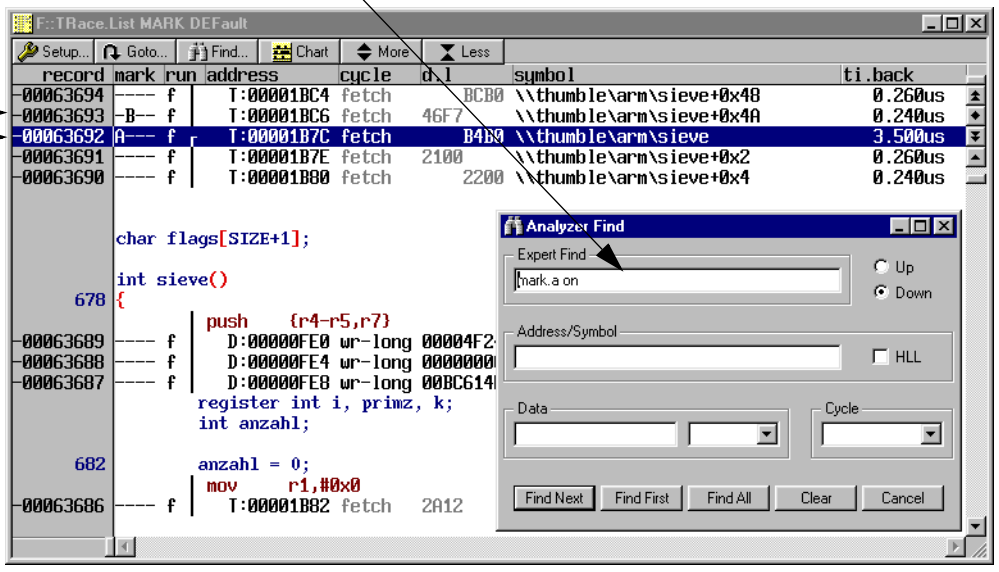
Level1:
Sample all cycles

Change to Level0 at the function exit and mark the function exit by a B marker

The “Analyzer Find” dialog box allows to search for the markers

Function exit

Function entry



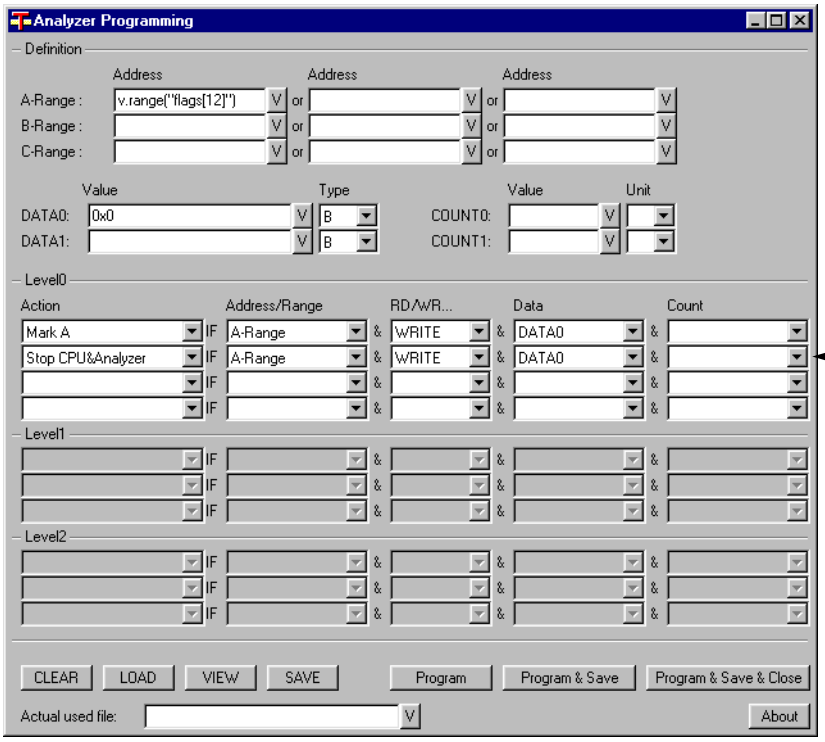
Stop the Program Execution

After a Wrong Data Value is Written

1. Use A-Range to define the memory location/variable.
2. Use DATA0 to define the data value.

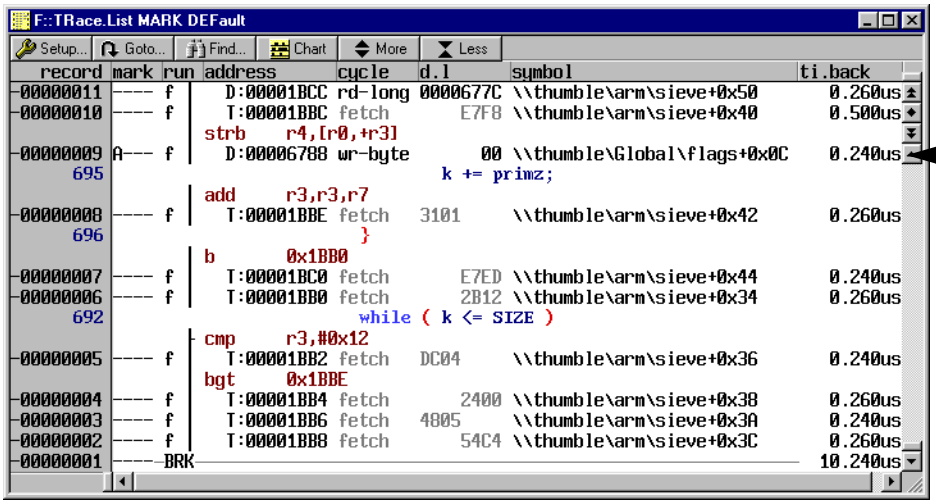
   To define a data value it is necessary to know if the data value is transferred as a byte, a word or a long.
3. Select the action Stop CPU&Analyzer and specify the condition by using the characteristic Address/Range and RD/WR… and DATA.

Example: Stop the program execution when 0 is written to flags[12]. Mark the record where the condition was true with an A marker.



Stop the program execution an the analyzer, when the data val defined by DATA0 is written to t address/range defined by A-Range.
Mark this cycle by an A marke

0 is written to flags[12] in the cycles marked by A. The CPU executes a few more cycles until the program execution stops



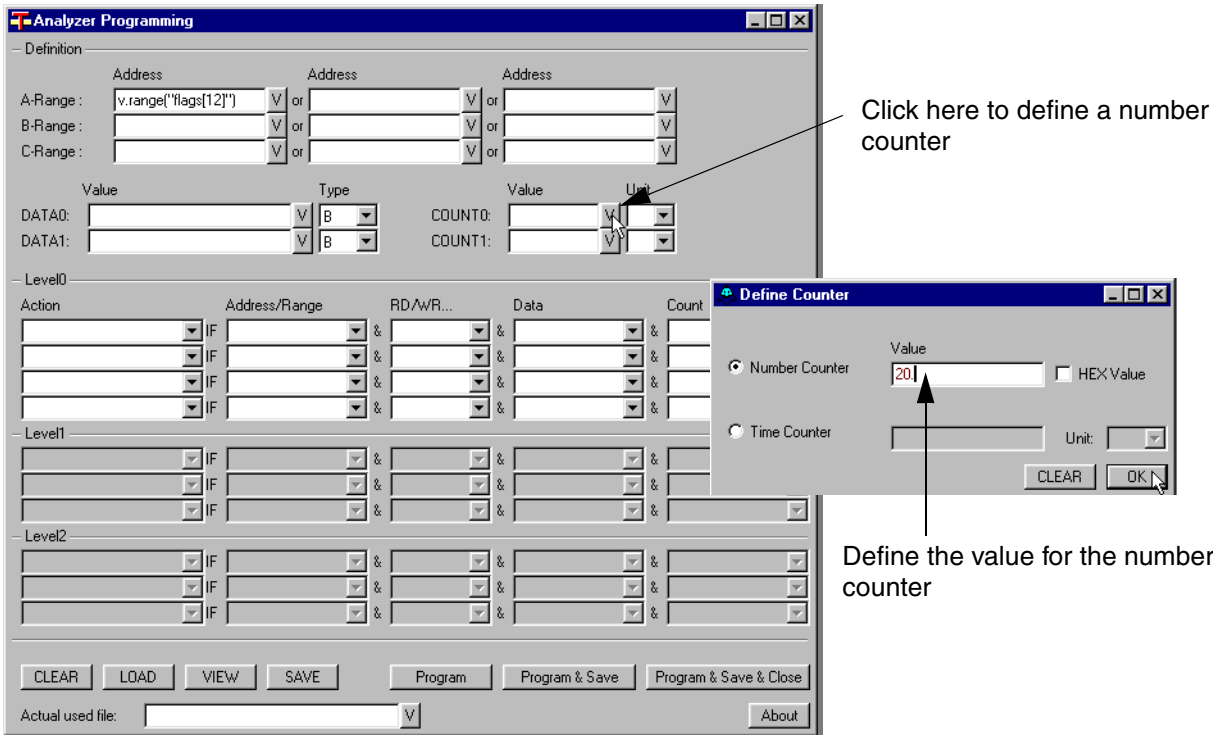
After n Accesses to a Memory Location/Variable

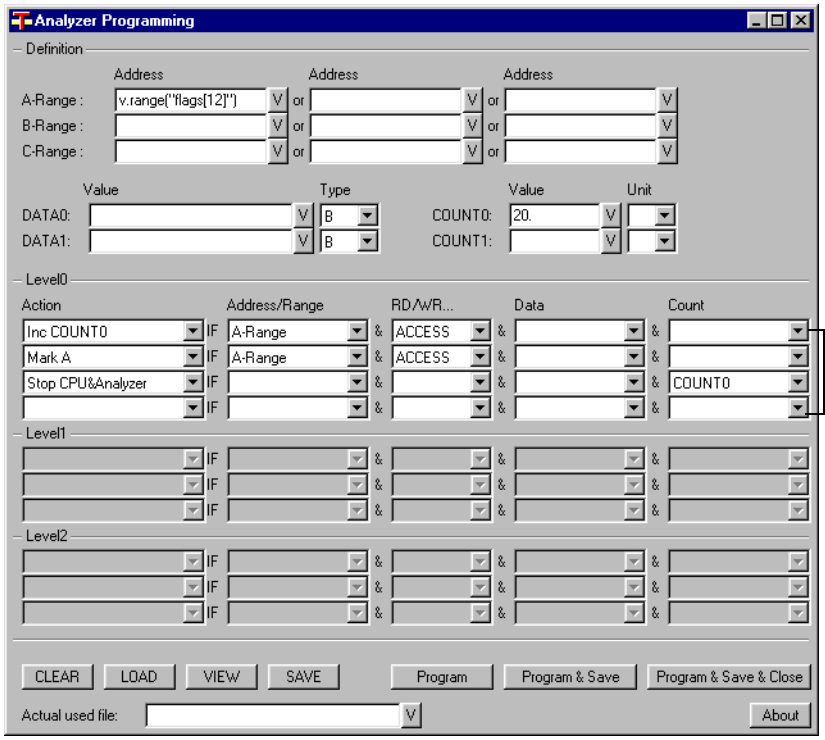
1. Use A-Range to define the memory location/variable.
2. Use COUNT0 to define a number counter, that counts the accesses to the memory location/variable.
3. Select a set of actions:

   To increment the number counter COUNT0 on each access to memory location/variable defined by A-Range

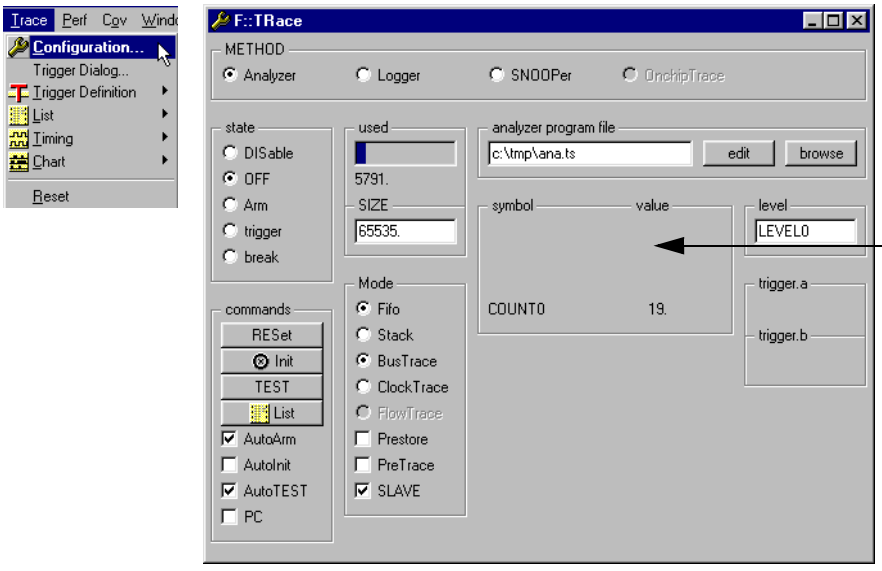
   To stop the program execution when the number counter COUNT0 has reached its final value

Example: Stop the program execution after flags[12] was accessed 20 times. Mark each access by an A marker.

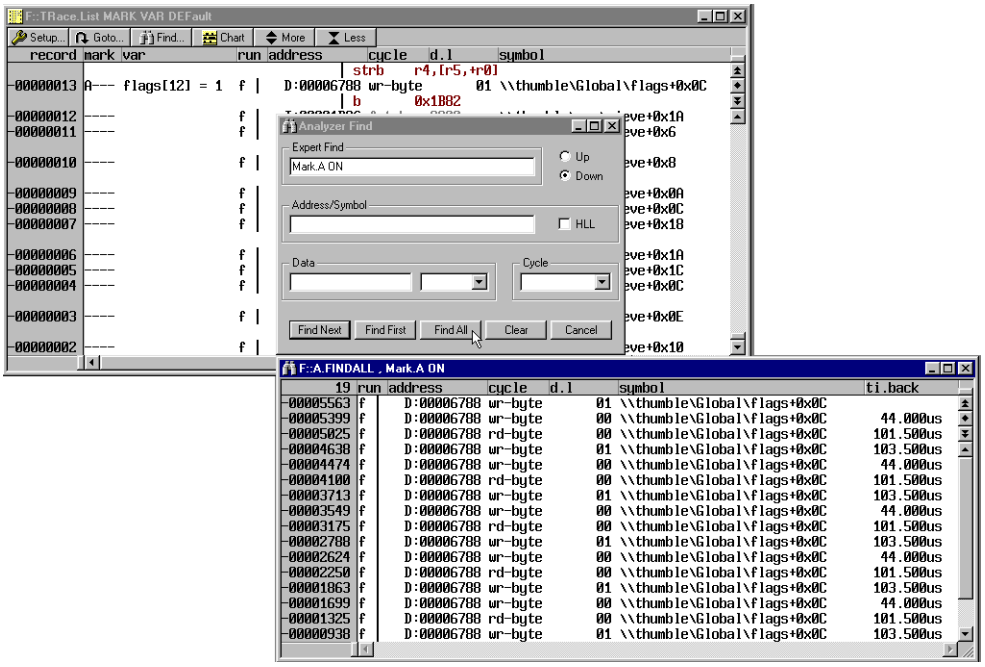




Increment the number counter COUNT0 on each access to the address/range defined by A-Range.
Mark each access to the address/range marked by A-Range by an A marker.
Stop the program execution when COUNT0 has reached its final value



The current state of the number counter can be watched in realtime in the Trace Configuration Window

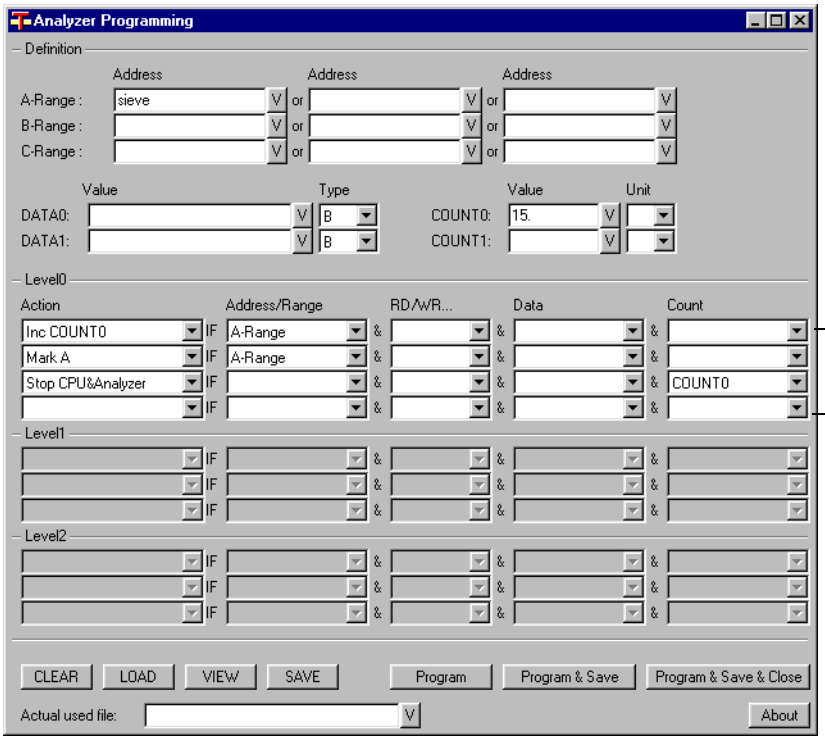


1. Use A-Range to define the function entry.
2. Use COUNT0 to define a number counter, that counts each function entry.
3. Select a set of actions:

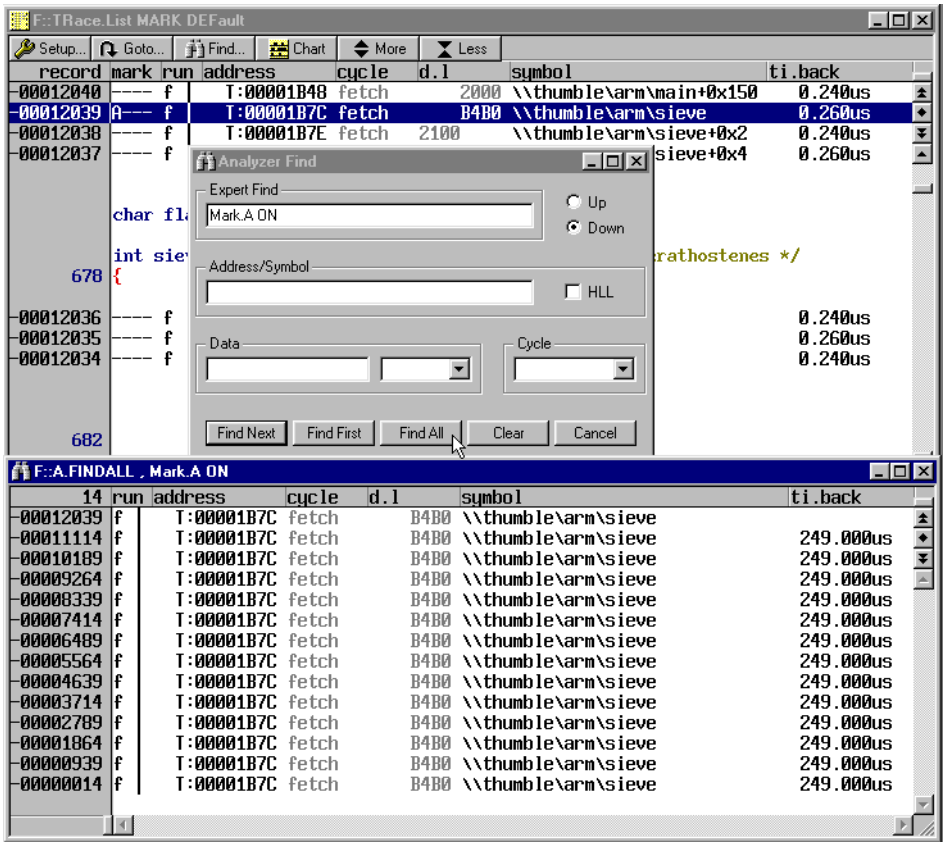
   To increment the number counter COUNT0 on each function entry

   To stop the program execution when the number counter COUNT0 has reached its final value

Example 1: Stop the program execution after the function sieve was entered the 15th time. Mark each function entry by an A marker.



Increment the number counter COUNT0 on entry to the function defined by A-Range.
Mark each function entry by an A marker.
Stop the program execution when COUNT0 has reached its final value



1. Use A-Range to define the memory location/variable.

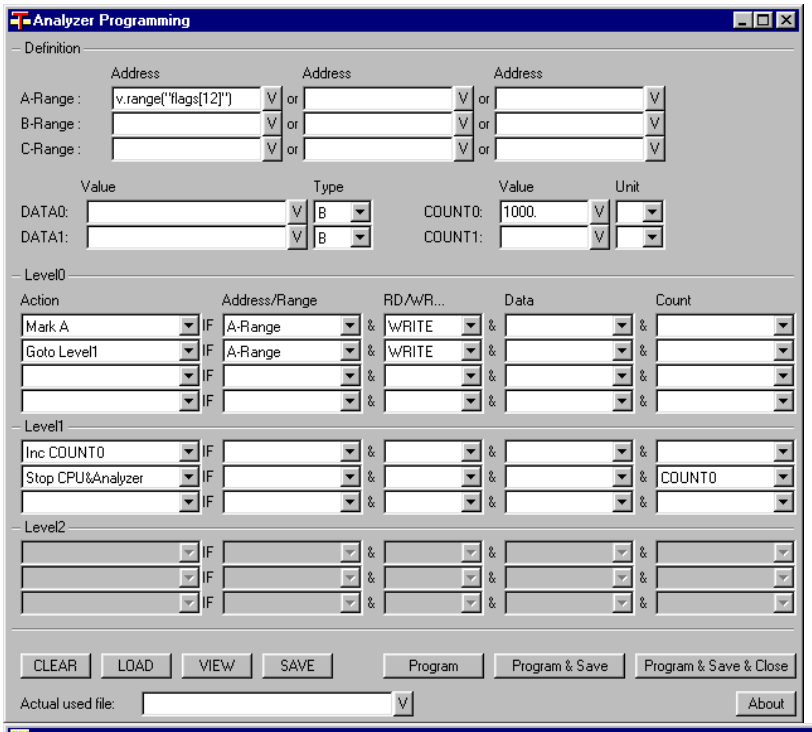
2. Use COUNT0 to define a number counter, that counts the number of records sampled to the trace buffer after the access to memory location/variable.

3. Select a set of actions by using 2 trigger levels.

   In the trigger level Level0 the analyzer program waits for he access to memory location/variable and changes to Level1 after the access.

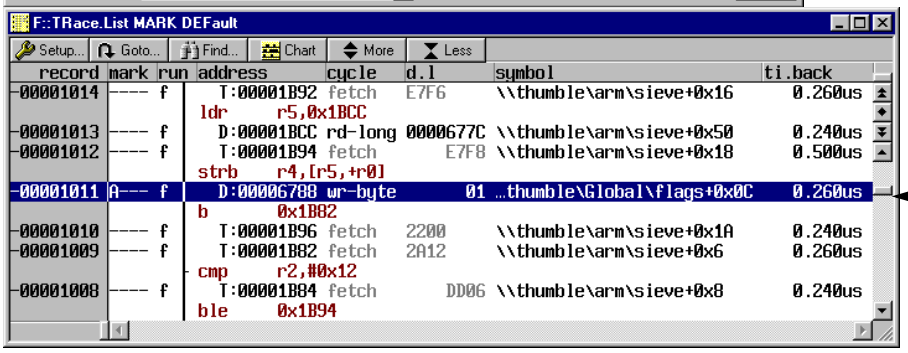
   In the trigger level Level1 the analyzer program samples the defined number of CPU cycles and stops the program execution afterwards.

Example: Sample 1000. cycles after a write access to the variable flags[12].



Level 0:
Mark the access to the memory location/variable with an A marker
Change to Level1 after the access to memory location/variable

Level 1:
Increment COUNT0 on each cycle
Stop the program execution, when COUNT0 has reached it final value



Sample another 1000 cycles after the write access to flags[12]

Statistic Functions

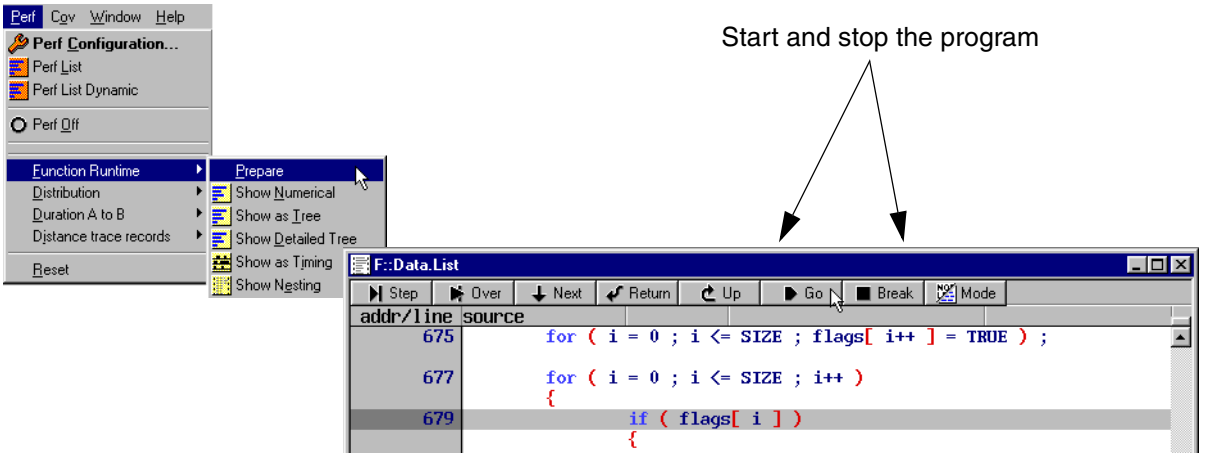
The statistic functions can be used to verify the runtime behavior of your embedded application.

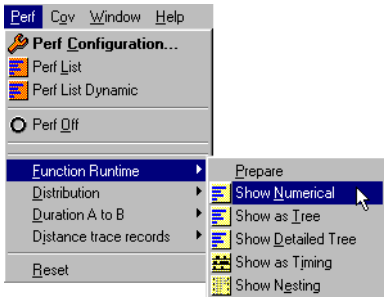
The statistic functions are based on:

- Selective tracing
- Time stamp on each sampled record

The statistic functions have a high precision and provide detailed information, but since they are based on the records sampled in the trace buffer their observation time is limited.

Numeric Analysis





Absolute measuring time

F::TRace.STATistic.FUNC

total: 34.591ms

range	time	min	max	avr	count	ratio	1% 2% 5% 10%
(root)	34.591ms	0.000	34.591ms	34.591ms	1.(-2)	0.000%	
rh8s\iarh8\main+0x2	34.591ms	0.000	34.591ms	34.591ms	1.(-1)	9.705%	
h8s\iarh8\func2+0x2	76.600us	76.600us	76.600us	76.600us	1.	0.208%	
h8s\iarh8\func1+0x2	10.500us	1.500us	1.500us	1.500us	7.	0.030%	
8s\iarh8\func2a+0x2	20.400us	20.400us	20.400us	20.400us	1.	0.058%	
8s\iarh8\func2b+0x2	141.100us	141.100us	141.100us	141.100us	1.	0.407%	
8s\iarh8\func2c+0x2	713.500us	713.500us	713.500us	713.500us	1.	2.062%	
8s\iarh8\func2d+0x2	48.400us	48.400us	48.400us	48.400us	1.	0.139%	
h8s\iarh8\func4+0x2	36.600us	36.600us	36.600us	36.600us	1.	0.105%	

Ratio of include time to total measuring time

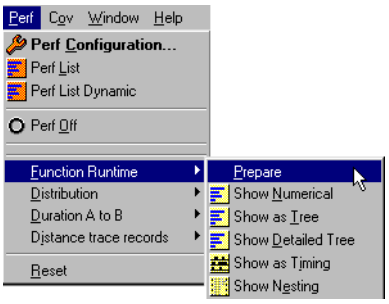
Number of calls of the function. A negative number indicates incompleted exits or entries

Average time from entry to exit

Longest time from entry to exit

Shortest time from entry to exit

Absolute time from function entry to exit added over the complete measuring time

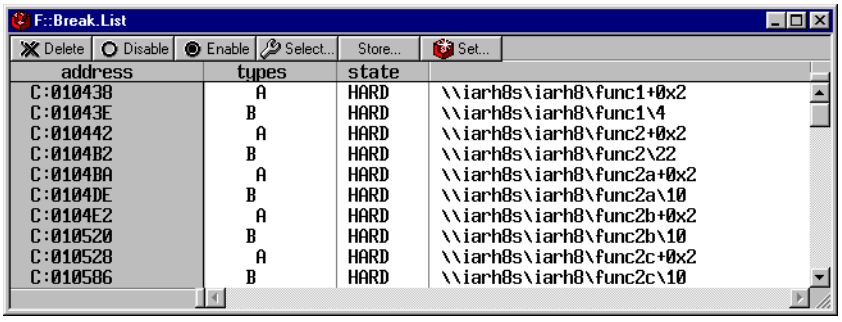


What does ***Function Runtime Prepare*** ?

1. All function entries are marked with an Alpha Breakpoint, all function exits are marked with a Beta Breakpoint.

   The breakpoints can be set automatically, if HLL-functions are loaded.

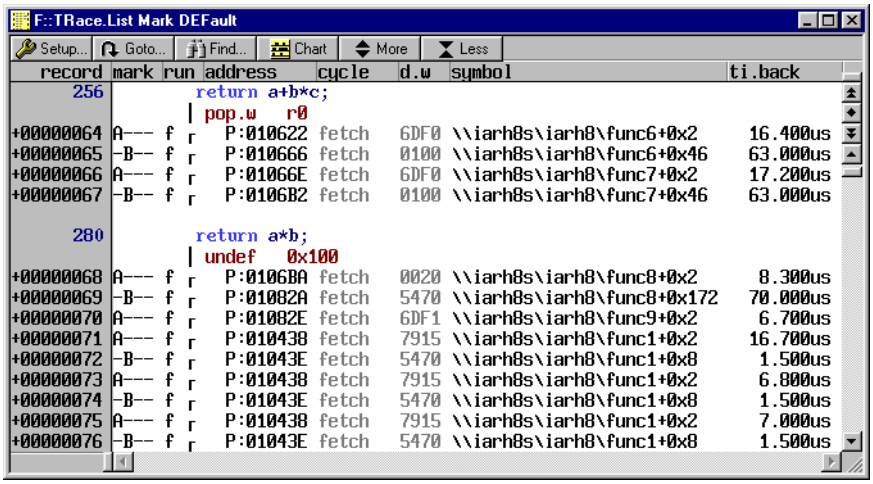
Break.SetFunc [<range>|<module>] [/<options>] Mark HLL functions



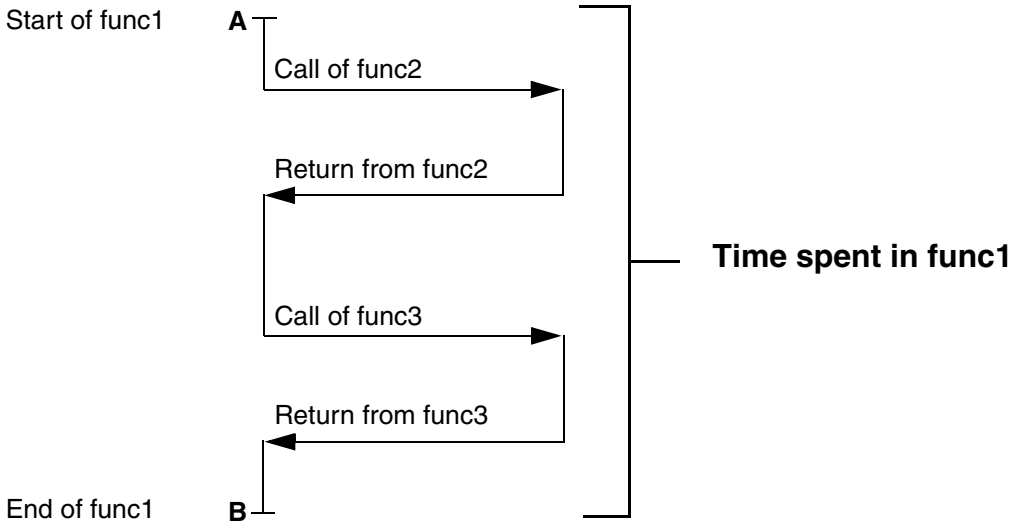
By default Alpha Breakpoints are set to ***function entry + prefetch offset***, to avoid that accesses to function entries caused by prefetches are sampled.

2. A trigger program is loaded, that samples only the function entries and exits. The function entries are marked with marker A and the function exits are marked with marker B.

```
Sample.enable IF AlphaBreak||BetaBreak

Mark.A IF AlphaBreak
Mark.B IF BetaBreak
```



- Initialize the trace buffer before you start the measurement.



F::TRace.STATistic.FUNC MIN MAX AVERAGE INTERNAL IAVERAGE COUNT

total: 34.591ms

range	min	max	avr	internal	iavr	count
(root)	0.000	34.591ms	34.591ms	0.000	0.000	1.(-2)
rh8s\iarh8\main+0x2	0.000	34.591ms	34.591ms	3.357ms	3.357ms	1.(-1)
h8s\iarh8\func2+0x2	76.600us	76.600us	76.600us	72.100us	72.100us	1.
h8s\iarh8\func1+0x2	1.500us	1.500us	1.500us	10.500us	1.500us	7.
8s\iarh8\func2a+0x2	20.400us	20.400us	20.400us	20.400us	20.400us	1.
8s\iarh8\func2b+0x2	141.100us	141.100us	141.100us	141.100us	141.100us	1.
8s\iarh8\func2c+0x2	713.500us	713.500us	713.500us	713.500us	713.500us	1.
8s\iarh8\func2d+0x2	48.400us	48.400us	48.400us	48.400us	48.400us	1.
h8s\iarh8\func4+0x2	36.600us	36.600us	36.600us	36.600us	36.600us	1.
h8s\iarh8\func3+0x2	0.100us	0.100us	0.100us	0.100us	0.100us	1.
h8s\iarh8\func5+0x2	7.100us	7.100us	7.100us	7.100us	7.100us	1.

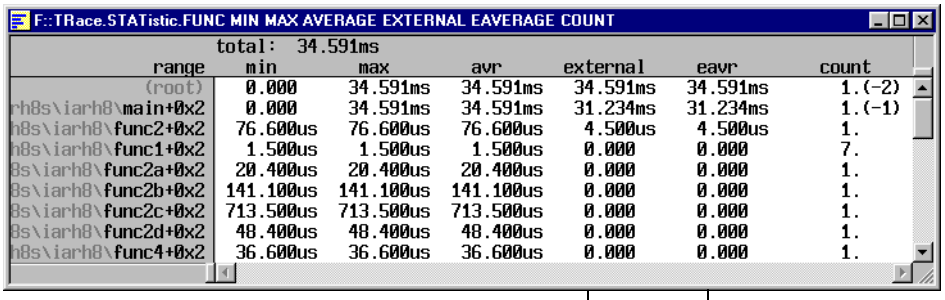
Average time within the function

Absolute time within the function
with regard to the complete measuring time

Average time from entry to exit (time in subroutines included)

Longest time from entry to exit (time in subroutines included)

Shortest time from entry to exit (time in subroutines included)



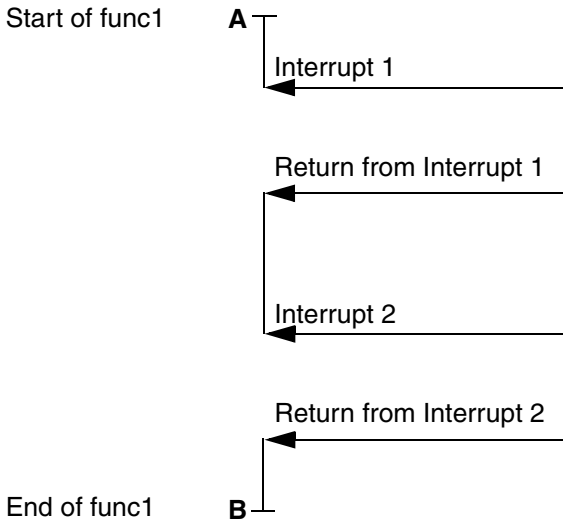
F::TRace.STATistic.FUNC MIN MAX AVERAGE EXTERNAL EAVERAGE COUNT

total: 34.591ms

range	min	max	avr	external	eavr	count
(root)	0.000	34.591ms	34.591ms	34.591ms	34.591ms	1.(-2)
rh8s\iarh8\main+0x2	0.000	34.591ms	34.591ms	31.234ms	31.234ms	1.(-1)
h8s\iarh8\func2+0x2	76.600us	76.600us	76.600us	4.500us	4.500us	1.
h8s\iarh8\func1+0x2	1.500us	1.500us	1.500us	0.000	0.000	7.
8s\iarh8\func2a+0x2	20.400us	20.400us	20.400us	0.000	0.000	1.
8s\iarh8\func2b+0x2	141.100us	141.100us	141.100us	0.000	0.000	1.
8s\iarh8\func2c+0x2	713.500us	713.500us	713.500us	0.000	0.000	1.
8s\iarh8\func2d+0x2	48.400us	48.400us	48.400us	0.000	0.000	1.
h8s\iarh8\func4+0x2	36.600us	36.600us	36.600us	0.000	0.000	1.

Average time spent in subfuntions

Absolute time spent in subfunctions
with regard to the complete measuring time



The statistic functions exclude interrupts and double fetches caused by interrupts, if the interrupt routines are marked as follows:

- Interrupt entry is marked by Alpha Breakpoint and a Charly breakpoint, interrupt exit is marked by Beta Breakpoint.

  Modified trigger program:

```
Sample.enable IF AlphaBreak||BetaBreak

Mark.A IF AlphaBreak
Mark.B IF BetaBreak
Mark.C IF CharlyBreak
```

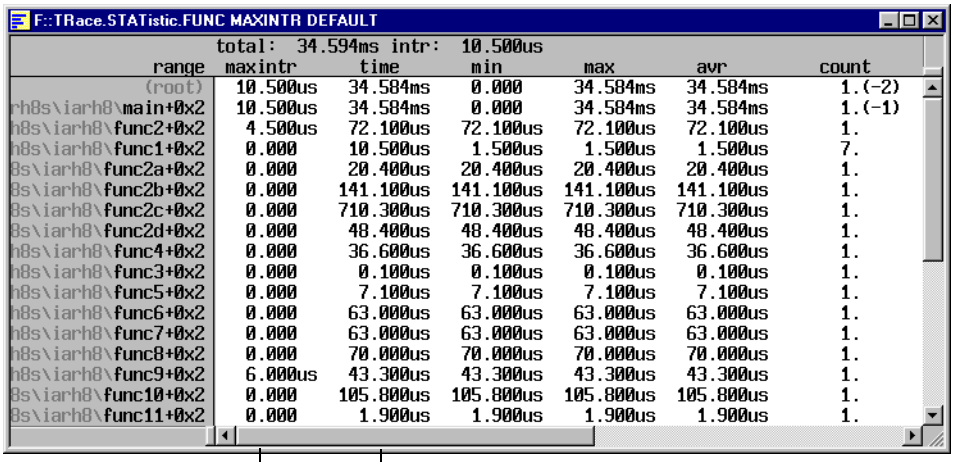
-

Break.SetFunc	[<range>|<module>] **/INTR**	Mark HLL functions

-

sYmbol.ForEach	<cmd> [<name_pattern>]	Symbol wildcard command

```
sYmbol.ForEach "Break.SetFunc * /INTR" INT*
```



F::TRace.STATistic.FUNC MAXINTR DEFAULT

total: 34.594ms intr: 10.500us

range	maxintr	time	min	max	avr	count
(root)	10.500us	34.584ms	0.000	34.584ms	34.584ms	1.(-2)
rh8s\iarh8\main+0x2	10.500us	34.584ms	0.000	34.584ms	34.584ms	1.(-1)
h8s\iarh8\func2+0x2	4.500us	72.100us	72.100us	72.100us	72.100us	1.
h8s\iarh8\func1+0x2	0.000	10.500us	1.500us	1.500us	1.500us	7.
8s\iarh8\func2a+0x2	0.000	20.400us	20.400us	20.400us	20.400us	1.
8s\iarh8\func2b+0x2	0.000	141.100us	141.100us	141.100us	141.100us	1.
8s\iarh8\func2c+0x2	0.000	710.300us	710.300us	710.300us	710.300us	1.
8s\iarh8\func2d+0x2	0.000	48.400us	48.400us	48.400us	48.400us	1.
h8s\iarh8\func4+0x2	0.000	36.600us	36.600us	36.600us	36.600us	1.
h8s\iarh8\func3+0x2	0.000	0.100us	0.100us	0.100us	0.100us	1.
h8s\iarh8\func5+0x2	0.000	7.100us	7.100us	7.100us	7.100us	1.
h8s\iarh8\func6+0x2	0.000	63.000us	63.000us	63.000us	63.000us	1.
h8s\iarh8\func7+0x2	0.000	63.000us	63.000us	63.000us	63.000us	1.
h8s\iarh8\func8+0x2	0.000	70.000us	70.000us	70.000us	70.000us	1.
h8s\iarh8\func9+0x2	6.000us	43.300us	43.300us	43.300us	43.300us	1.
8s\iarh8\func10+0x2	0.000	105.800us	105.800us	105.800us	105.800us	1.
8s\iarh8\func11+0x2	0.000	1.900us	1.900us	1.900us	1.900us	1.

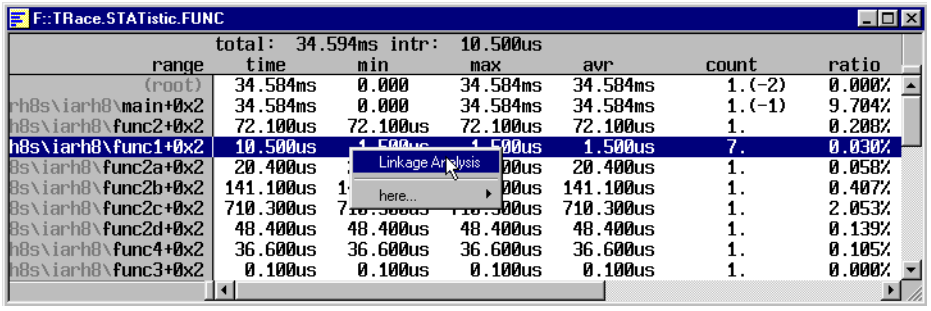
Absolute time from function entry to exit added for the complete measuring time minus time spent in interrupts

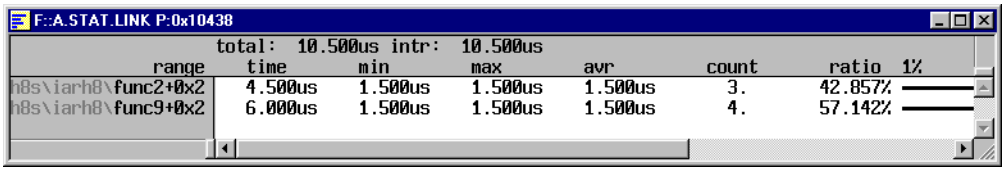
Maximum time, the function was interrupted.

<trace>.STATistic.Func [%<format>][<item> …] [/<options>] Function runtime analysis

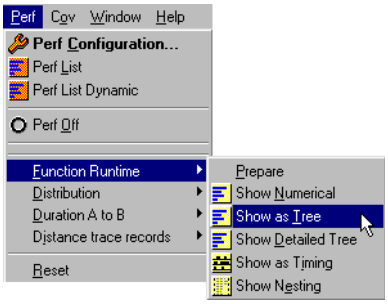
Linkage Analysis

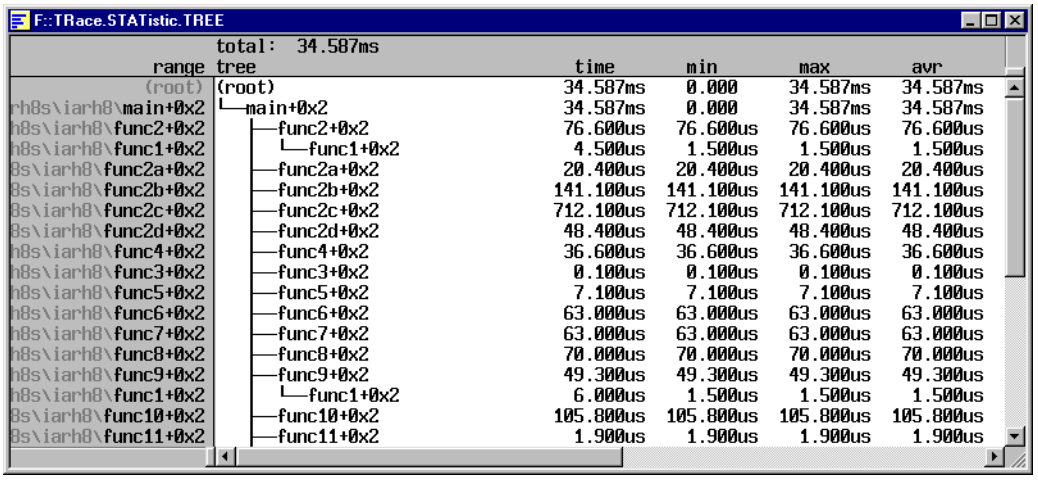
The linkage analysis displays the caller of a specific function.





Tree Analysis





Statistic Distribution of a Single Event

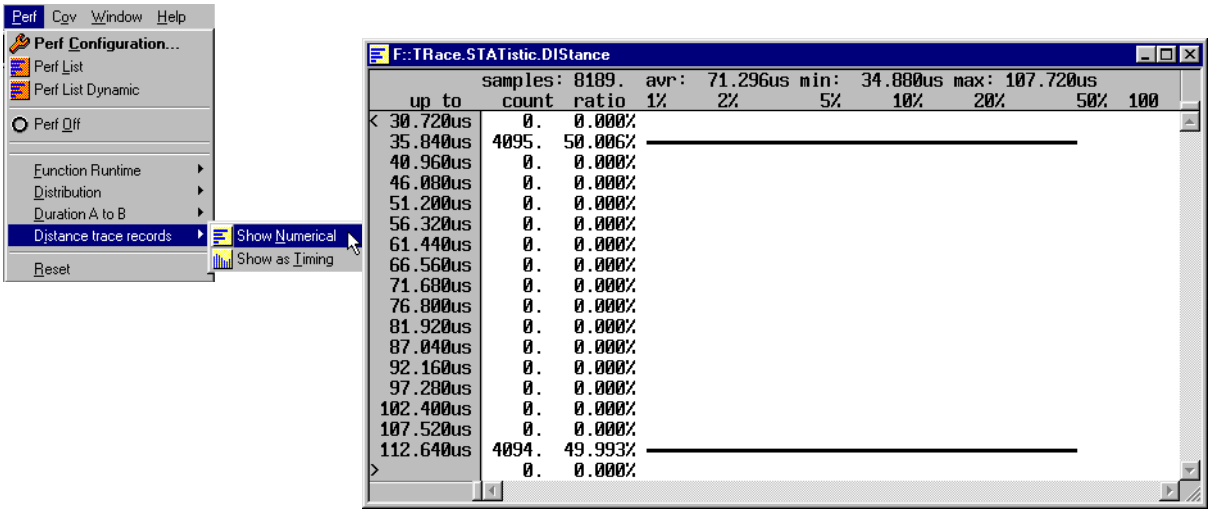
The statistic distribution of the time distance between the analyzer records is analyzed.

- Make a selective trace of a specific event
- Analyse the time distance

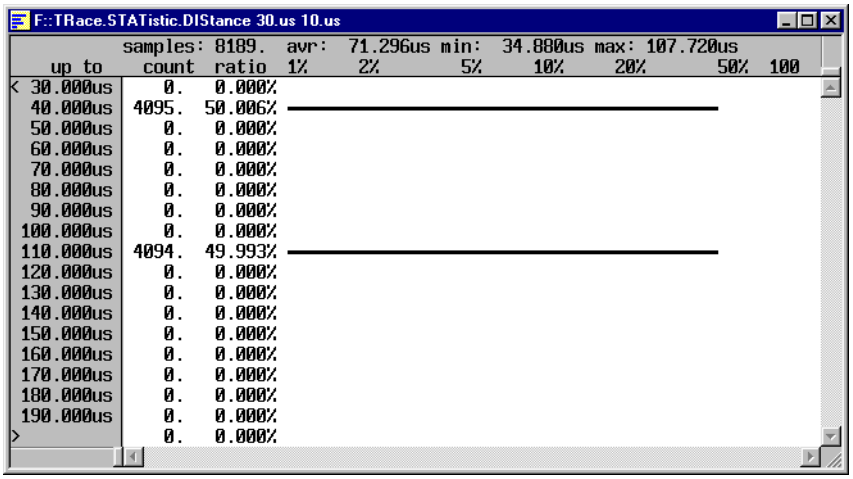
Example:

Analyse the statistic distribution of write accesses to flags[3].

```
ADDR AlphaBreak V.RANGE(flags[3])

Sample.enable IF AlphaBreak&&Write
```



TRace.STATistic.DIStance 30.us 10.us



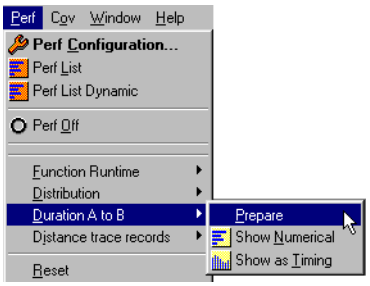
<trace>.STATistic.DIStance	[<timemin>] [<inc>]	Time interval for a single event

Statistic Distribution between 2 Events

The statistic distribution of the time distance between the record marked with A and the record marked with B is analyzed.

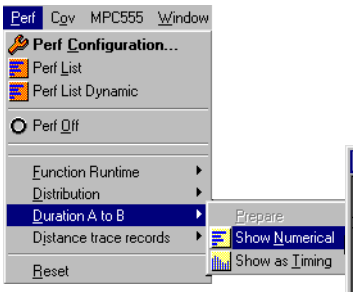
- Set an AlphaBreak to the first event
- Set a BetaBreak to the second event
- Prepare Analyzer Trigger Unit
- Analyse the time distance

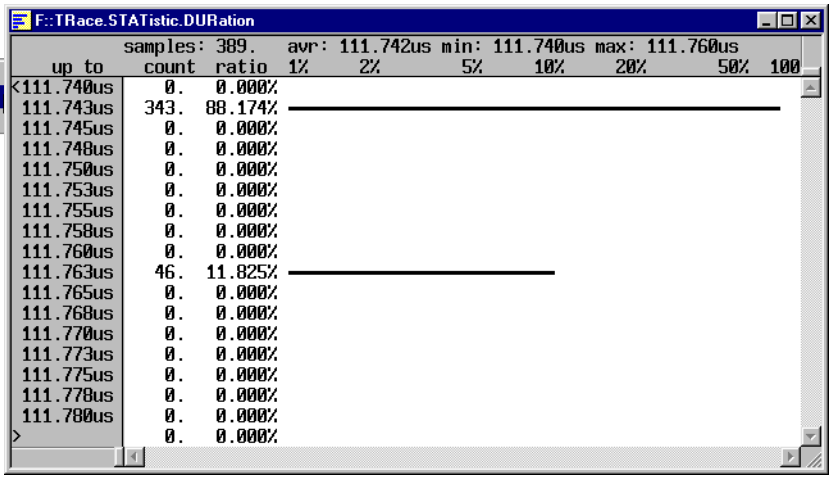
Example:



Generates the following trigger program:

```
Sample.enable IF AlphaBreak||BetaBreak
              Mark.A IF AlphaBreak
              Mark.B IF BetaBreak
```





<trace>.STATistic.DURation [<timemin>] [<inc>] Time between two events

For more complex events, write your own trigger program.

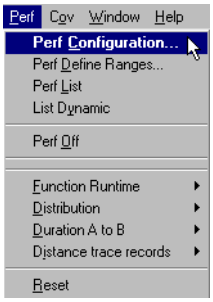
The Performance Analyzer

The performance analyzer displays the percentage of time spent in the different functions, modules etc.

Compared to the statistic function, the performance analyzer offers a long time measurement, but with less information.

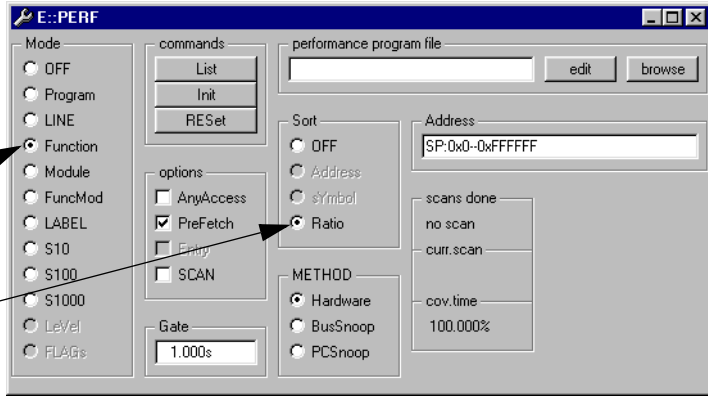
Function Performance Analysis

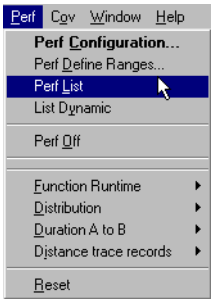
Example

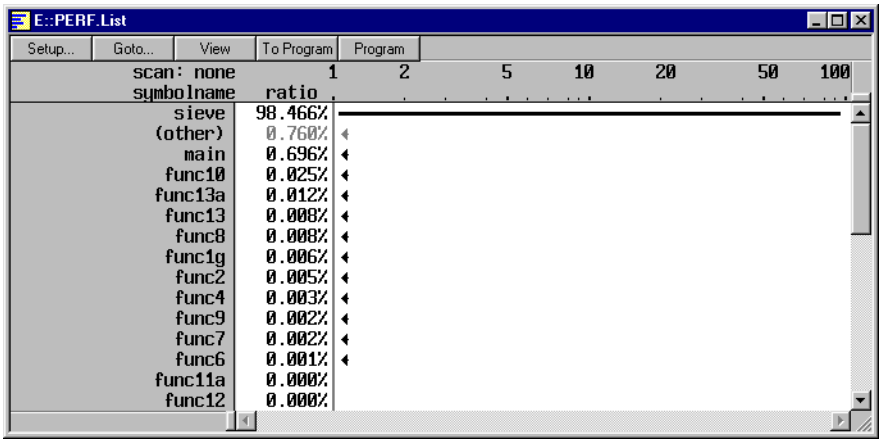


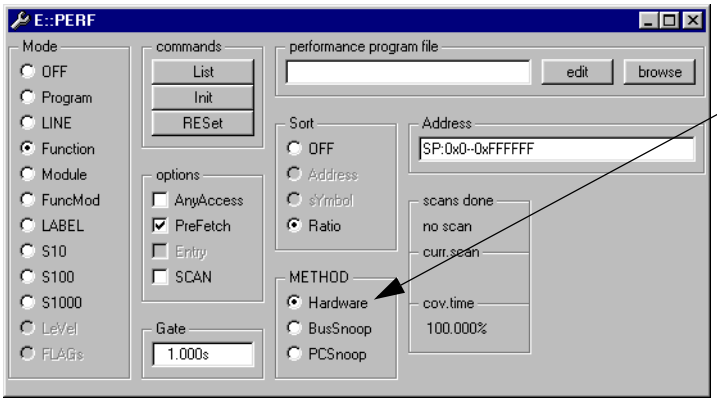
Measure the function performance

Display the result by ratio





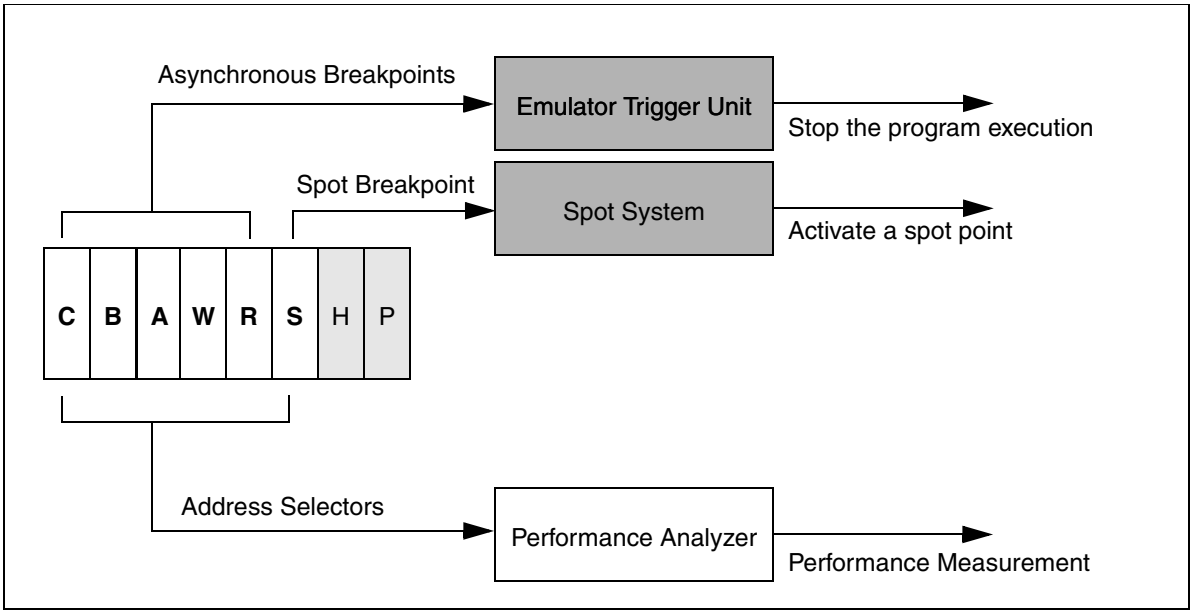




Default measurement method for TRACE32-ICE is **Hardware**

If the measurement method is Hardware, the performance analyzer can be seen as an array of 64 counters. Only one counter is getting the clock to count, all other counters are stopped. The resolution of the counter is 1 µs.

The counter selection is done by using the asynchronous breakpoints.



The asynchronous breakpoints are automatically disabled in the Emulator Main Trigger Unit and the Spot System is switched off as long as the Performance Analyzer is active.

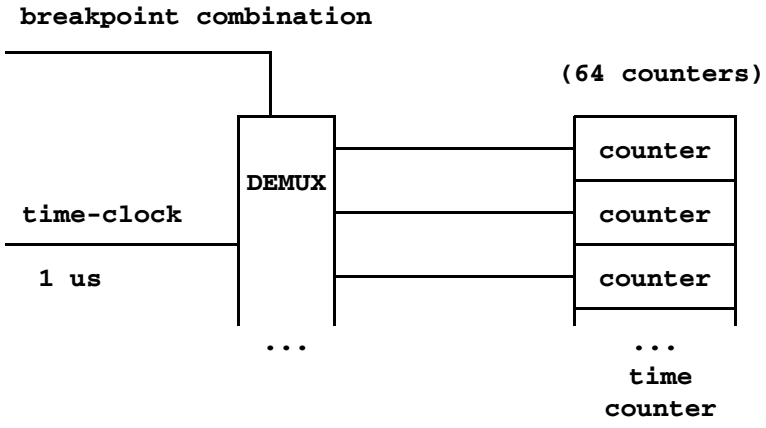
The breakpoints are automatically set to the functions, modules etc. if HLL information is loaded.

E::Break.List

Address	C	
C:0000105A--0000105F	RS	\\mcc\mcc\func1+0x2
C:00001062--00001069	R	\\mcc\mcc\func1g+0x2
C:0000106C--000010CD	WR	\\mcc\mcc\func2+0x2
C:000010D0--000010D1	WRS	\\mcc\mcc\func3+0x2
C:000010D4--000010ED	W S	\\mcc\mcc\func4+0x2
C:000010F0--00001105	W	\\mcc\mcc\func5+0x2
C:00001108--0000114D	AW	\\mcc\mcc\func6+0x2
C:00001150--000011B1	AW S	\\mcc\mcc\func7+0x2
C:000011B4--0000132F	AWRS	\\mcc\mcc\func8+0x2
C:00001332--00001381	AWR	\\mcc\mcc\func9+0x2
C:00001384--00001517	A R	\\mcc\mcc\func10+0x2
C:0000151A--0000155D	A RS	\\mcc\mcc\func11+0x2
C:00001560--0000158F	A S	\\mcc\mcc\func11a+0x2
C:00001592--00001599	A	\\mcc\mcc\func12+0x2
C:0000159C--000015E3	BA	\\mcc\mcc\func13+0x2
C:000015E6--00001651	BA S	\\mcc\mcc\func13a+0x2
C:00001654--00001787	BA RS	\\mcc\mcc\main+0x2
C:0000178A--000017DB	BA R	\\mcc\mcc\sieve+0x2

With 6 address selectors up to 64 functions can be marked with different breakpoint combinations.

The functions are marked according the predefinded prefetch offset (SETUP.PreFetch) if the PreFetch check box is switched on.



The measurement is made in intervals, after each interval the results in the display windows are refreshed. The default interval is 1 s.

Performance Scanning

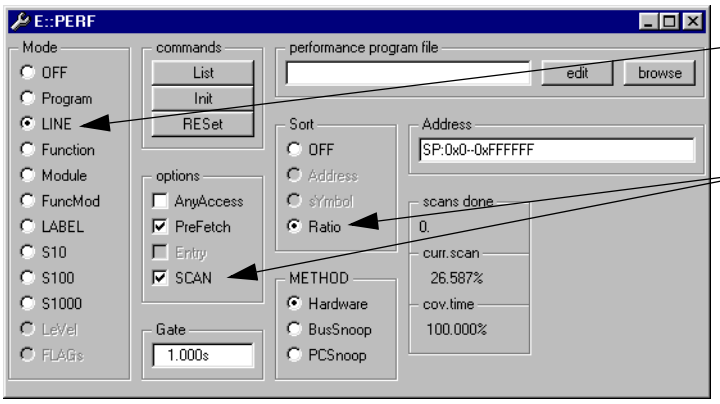
If the number of symbols exceed the number of counters, the performance analyzer can run in scan mode.

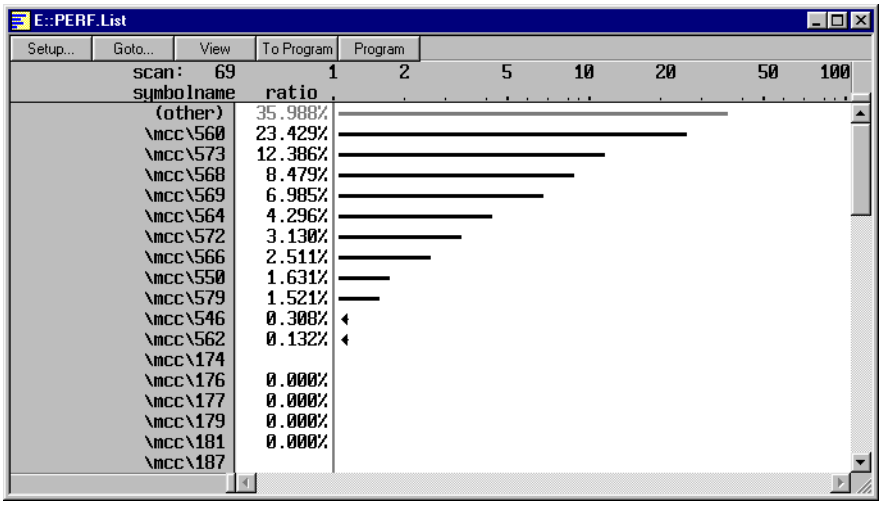
Performance Scanning:

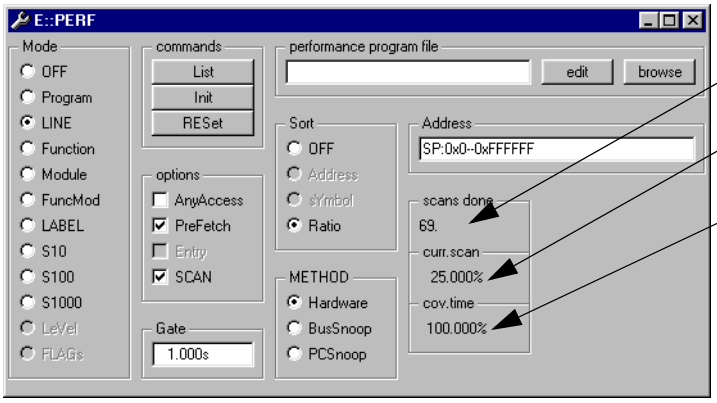
1. The first 64 areas are marked.
2. After each measurement interval the 32 less time consuming functions are thrown out of the measurement.

   The next 32 areas are marked
3. The result of the performance scanning is a list of the 32 most time consuming areas.

Example





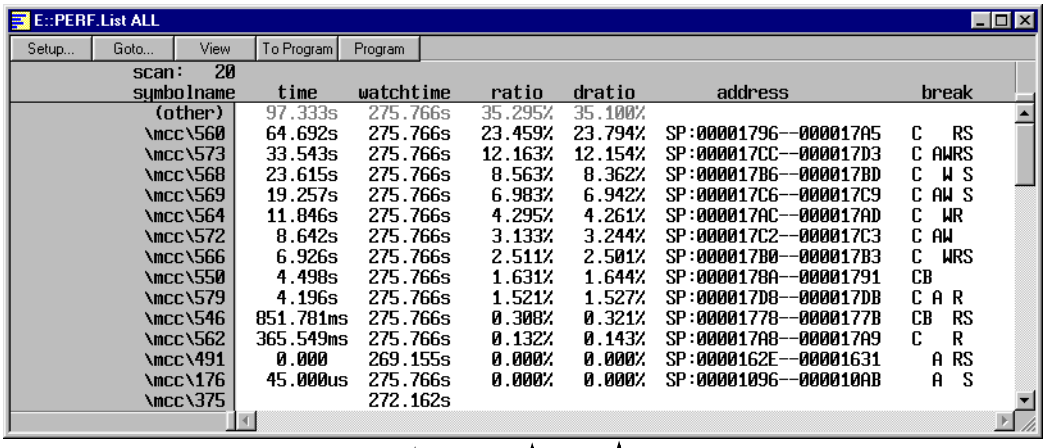


Number of complete scans

Percentage of the areas that where scanned in the current scan

Time covered by the currently measu 64 ranges

PERF.List ALL



Ratio of time spent in this range in the last measurement interval (dynamic)

Ratio of time spent in this range

Time this range was watched

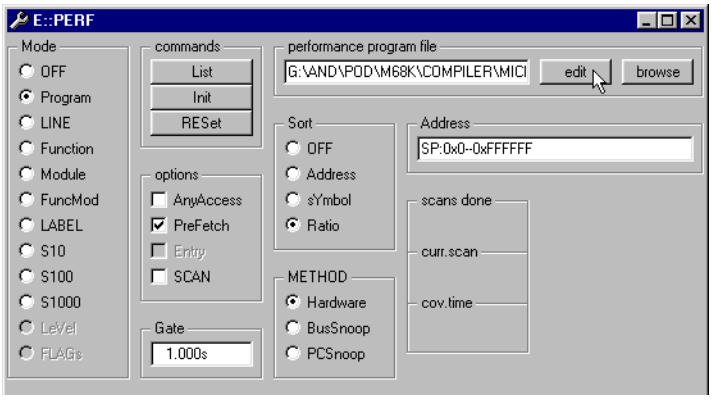
Time spent in the range

If a range is thrown out of the currently watched ranges, **not information about this range is stored.**

If the range reenters later the currently watched ranges, watchtime and time start at 0.

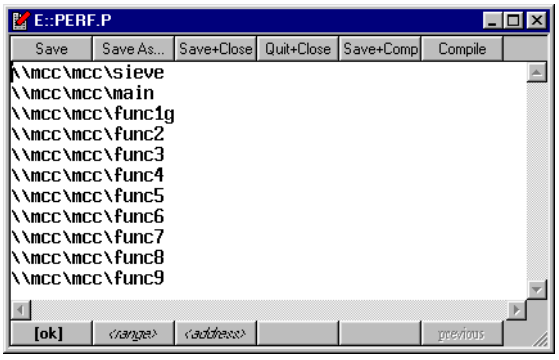
The ratio spent on this range depends on: time spent in the range and the time this range is watched. So if a very time consuming range is only watched for a short time, it can come up in the ratio list very quickly.

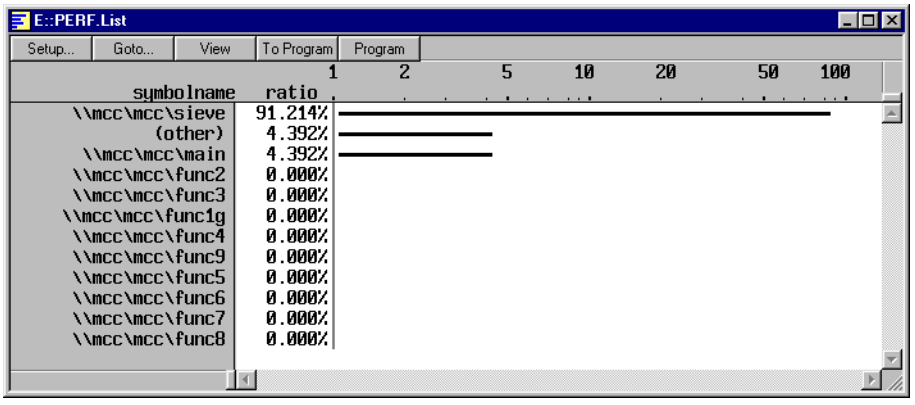
It is also possible to define spefic address ranges for the performance analyzer by using the performance analayzer programming window.



Compile the performance analyzer program an
load it into the performance analyzer and save

Compile the performance analyzer program an
load it into the performance analyzer unit





Context Tracking System (CTS)

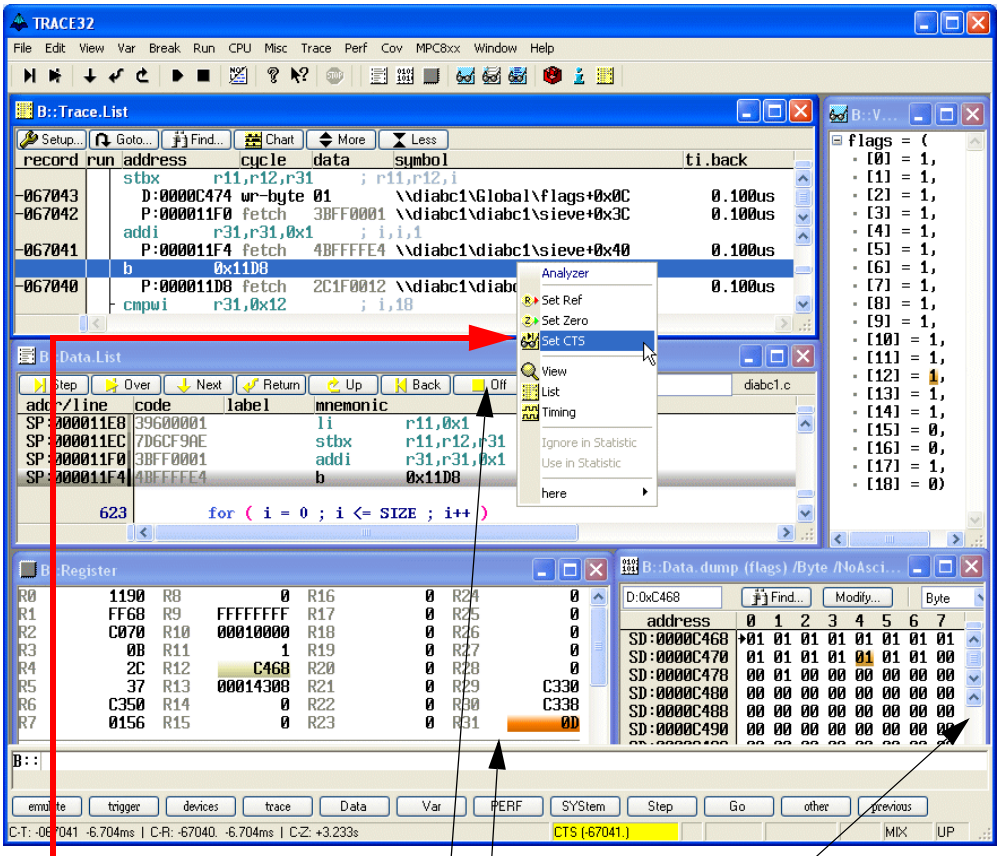
Context tracking is a technique that allows the context of the CPU/target to be reconstructed for a selected trace record based on the information sampled in the trace buffer. Context in this case means the contents of memory and registers.

The basic features of the Context Tracking System as described in this section require that:

- full program and data information (all read accesses) is available
- all CPU cycles until the program execution stopped are sampled to the trace buffer

More features and settings for CTS are described with the **CTS** command group.

Trace based debugging allows to re-run the program and data flow sampled into the trace buffer on the TRACE32 screen. Trace based debugging is set-up as follows:



1.) Select the recording point for which the state of the CPU/target should be reconstructed.

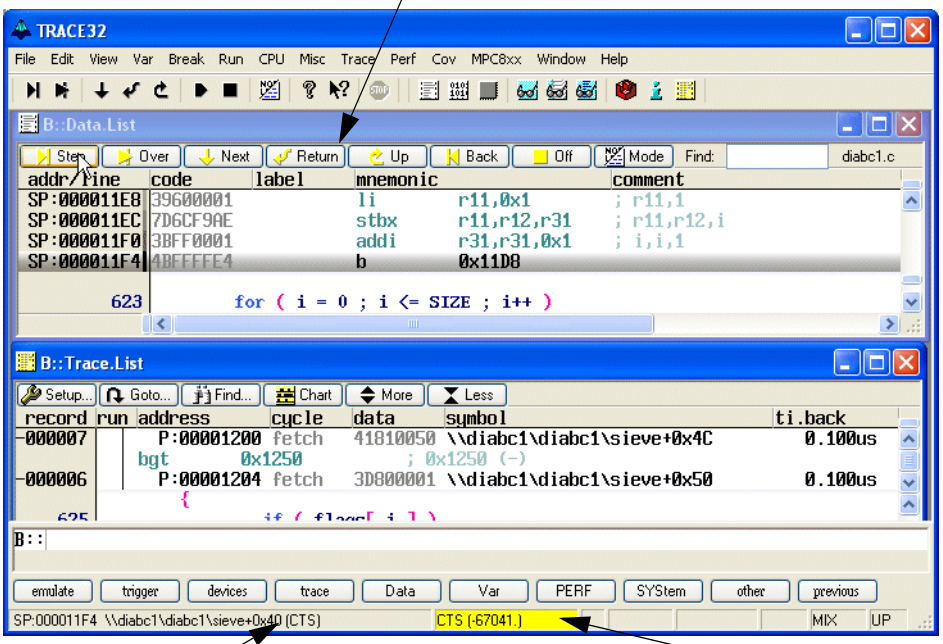
2.) The PC will be set automatically to this recording point.

3.) During the trace-based debugging the changes of memories, variables and registers can be watched

With active CTS the windows in the TRACE32 screen don't show the current state of the CPU and the current contents of the memory and registers. The windows display the memory, registers etc. as they were at the moment, when the selected record was sampled.

To avoid irritations there are a number of indicators that show you that CTS is active.

Yellow buttons in the Data.List window indicate that CTS is active



A (CTS) after the logical and symbolical address in the state line shows you, that CTS is active

A yellow field in the state line shows you, that you see the memories and registers as they were when record number -xxxx. was sampled

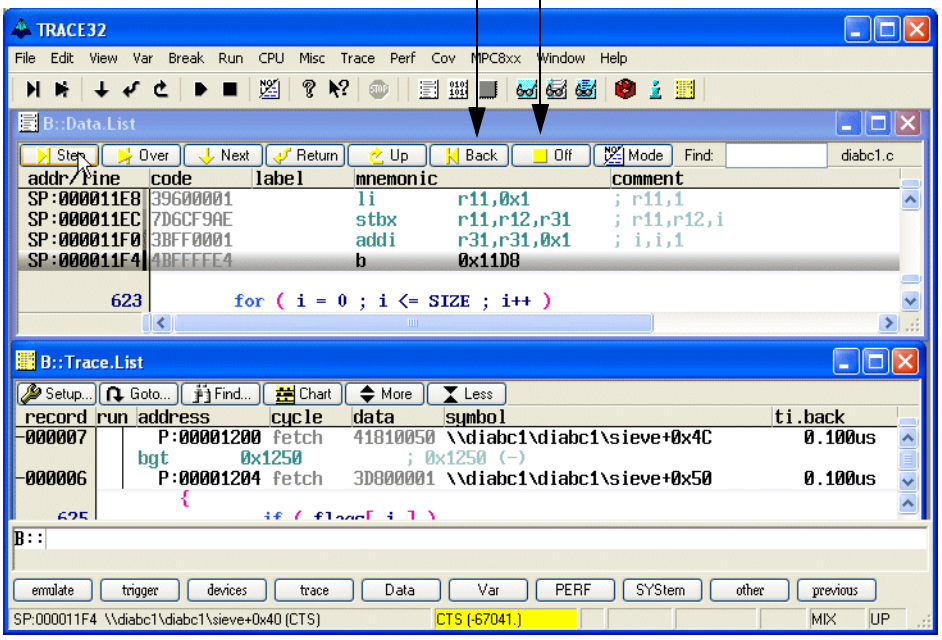
All basic debugging commands can be executed: **`Step, Step.Over, Go.Next, Go.Return, Step.Change, Var.Step.Change, Step.Till, Var.Step.Till.`**

CTS also supports a **`Step.Back`** command.

With **OFF** CTS is switched off. The current state of the CPU and the current values of registers and variables are displayed on the TRACE32 screen.

Step.Back

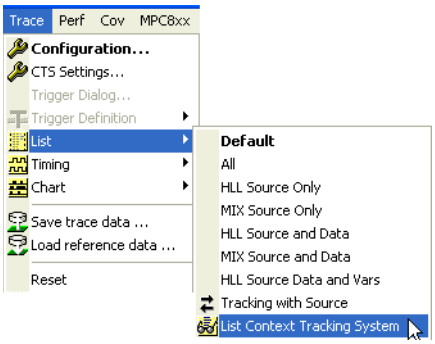
OFF (Switch CTS OFF)

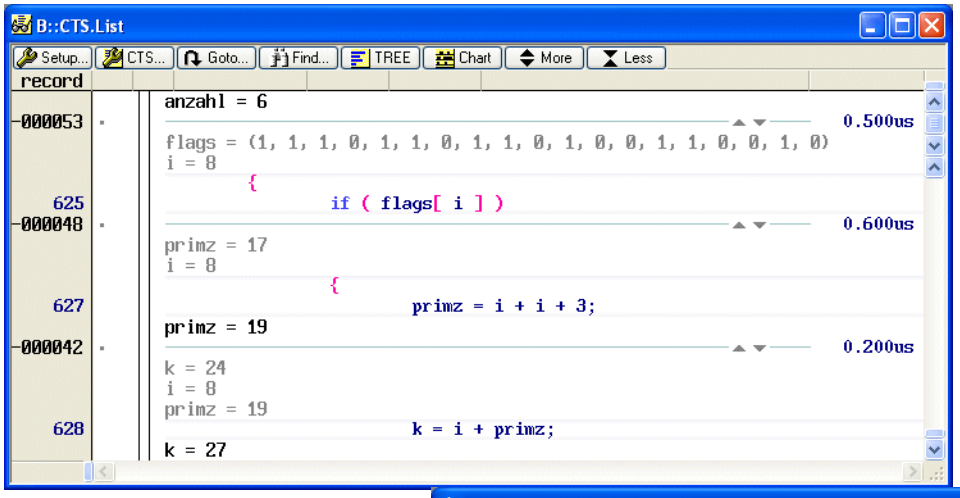


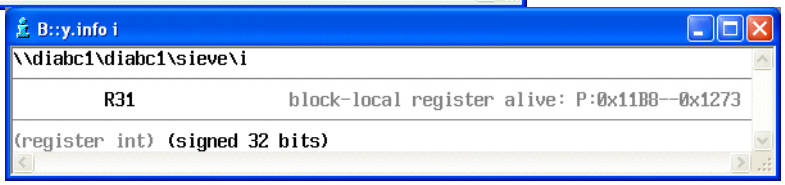
HLL Analysis of the Trace Contents

CTS provides also a number of features for hll trace display.

Analyse each HLL Step





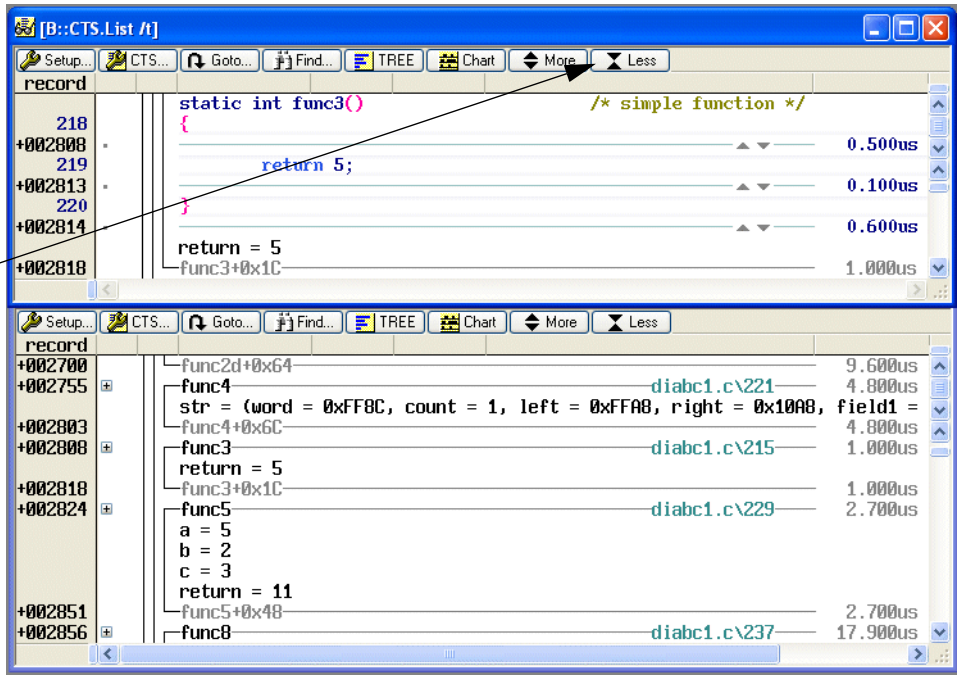


For each hll step the following information is displayed:

- 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.

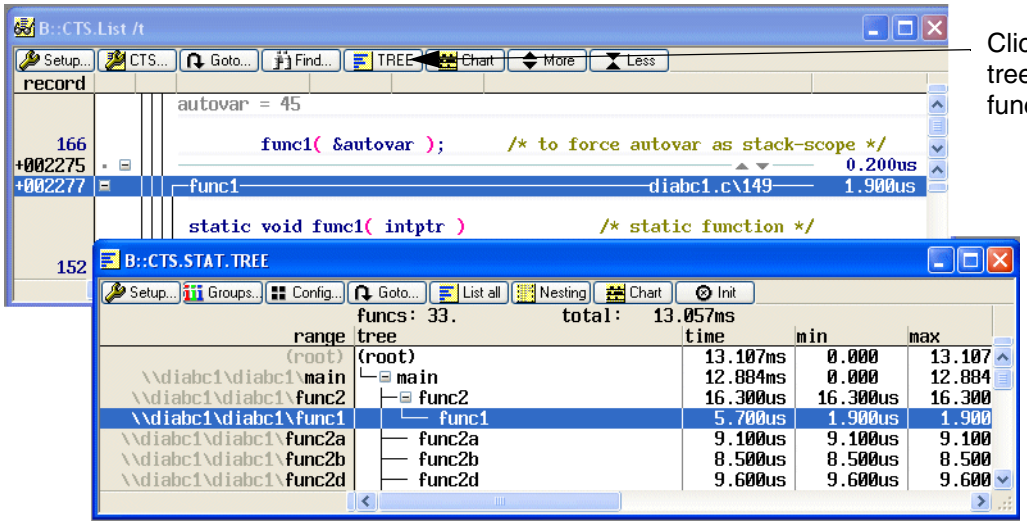
Push the ***Less*** button to get a function nesting analysis



For each function you get the following information:

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

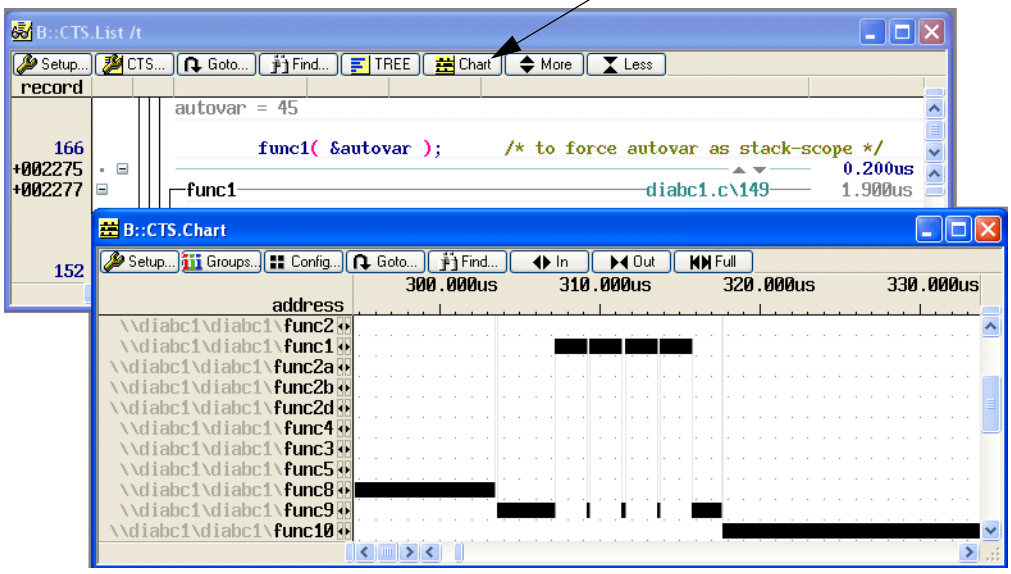
Tree Display



Click here to get a tree analysis of the function nesting

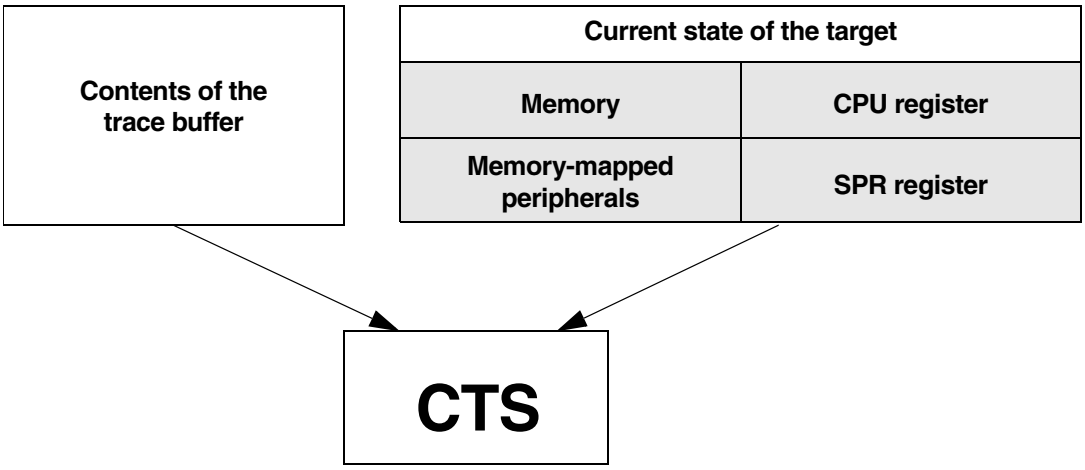
Timing Display

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

Background



CTS read and evaluated the current state of the target.

1. CTS can only perform a correct context reconstruction, if solely the processor core, for which the data flow is sampled 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 context reconstruction. These memory addresses are then displayed as unknown if CTS is used.

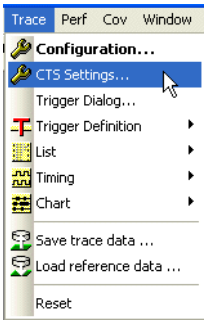
2. CTS performs memory reads while performing a context reconstruction. If read accesses to specific memory-mapped peripherals should be prevented, the addresses have to be excluded by the command **MAP.VOLATILE** from the CTS context reconstruction. These memory addresses are then displayed as unknown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

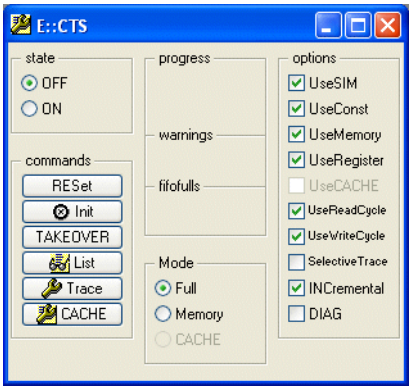
3. Under certain circumstances the reconstruction of the program flow can cause BUSERRORS on the target system. If this is the case, it is recommended to load the code to virtual memory.

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.