LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide B

Release 09.2023

General Commands Reference Guide B

Version 09-Oct-2023

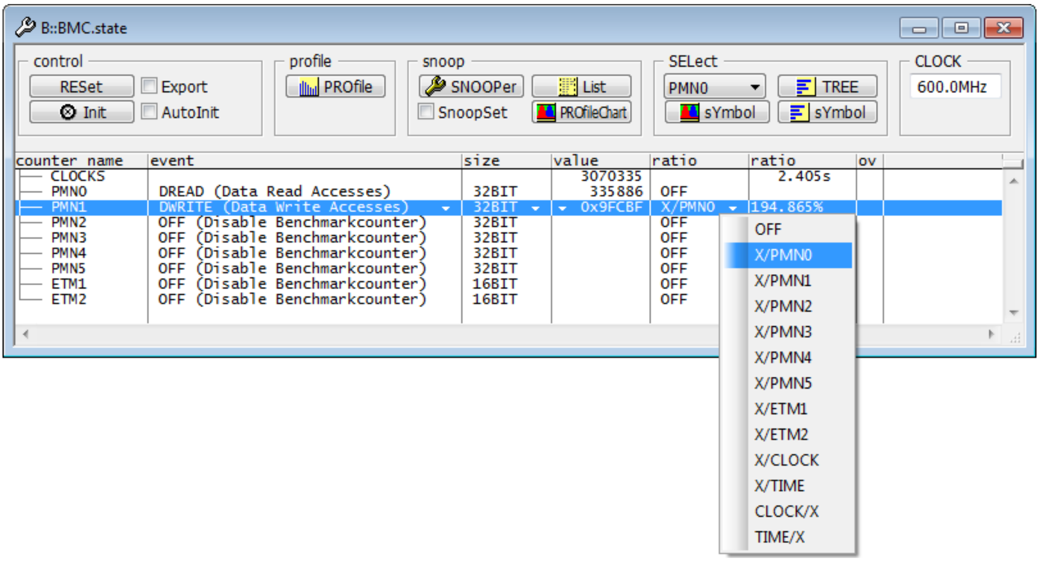
History

20-Jul-2023	New option /**OnchipDetail** for the command Break.List.
18-Jan-2023	New option /**SPOT** for the command Break.SetFunc.
07-Oct-2022	Information about task-aware real-time breakpoints for Cortex-X, Neoverse and RISC-V has been added to the description of the Break.Set command.
09-Mar-2022	New option /**DeleteHIT** for the command Break.Set.
13-Dec-2021	New command Break.CONFIG.AlwaysAlive.

BMC Benchmark counters

The **BMC** (**B**ench**M**ark **C**ounter) commands provide control and usage of the on-chip performance monitoring capabilities. Benchmark counters are on-chip counters that count specific hardware events, e.g., the number of executed instructions.

The benchmark counters can be configured via the TRACE32 command line, a PRACTICE script (*.cmm), or the **BMC.state** window. This document presents the generic functions while the architecture_specific BMC commands are in the **Processor Architecture Manual**.



counter name	event	size	value	ratio	ratio	ov
CLOCKS			3070335		2.405s	
PMN0	DREAD (Data Read Accesses)	32BIT	335886	OFF		
PMN1	DWRITE (Data Write Accesses)	32BIT	0x9FCBF	X/PMN0	194.865%	
PMN2	OFF (Disable Benchmarkcounter)	32BIT		OFF		
PMN3	OFF (Disable Benchmarkcounter)	32BIT		OFF		
PMN4	OFF (Disable Benchmarkcounter)	32BIT		OFF		
PMN5	OFF (Disable Benchmarkcounter)	32BIT		OFF		
ETM1	OFF (Disable Benchmarkcounter)	16BIT		OFF		
ETM2	OFF (Disable Benchmarkcounter)	16BIT		OFF		

See also

- BMC.<counter>
- BMC.Init
- BMC.RESet
- BMC.Attach
- BMC.PROfile
- BMC.SnoopSet
- BMC.AutoInit
- BMC.PROfileChart
- BMC.state
- BMC.CLOCK
- BMC.PROfileSTATistic
- BMC.STATistic

▲ ’BMC Functions (Benchmark Counter)’ in ’General Function Reference’
▲ ’Release Information’ in ’Legacy Release History’

BMC.<counter> Benchmark counters

See also

- BMC.<counter>.EVENT
- BMC.<counter>.FORMAT
- BMC.<counter>.RATIO
- BMC.<counter>.SIZE
- BMC
- BMC.state

BMC.<counter>.EVENT Assign event to counter

Format: **BMC.**<*counter*>**.EVENT** [<*event*> | <*event_number*>]

Assigns an event to a counter.

<*event*> Event name defined by core manufacturer.

<*event_number*> Custom event ID.

```
BMC.<counter>.EVENT ClockCycles        ; <counter> counts clock cycles

BMC.<counter> ClockCycles              ; equivalent
```

See also

- BMC.<counter>

BMC.<counter>.FORMAT Counter value format

Format: **BMC.**<*counter*>**.FORMAT** <*format*>

Sets up the display format for the for each benchmark counter.

```
BMC.<counter>.FORMAT DECimal           ; Display the counter value in
                                       ; decimal format.

BMC.<counter>.FORMAT HEXadecimal       ; Display the counter value in
                                       ; hexadecimal format.
```

See also

- BMC.<counter>

BMC.<counter>.RATIO Set two counters in relation

Format: **BMC.***<counter>***.RATIO X/***<counter _n>*

It might be useful to set two counter values in relation to each other, e.g. data cache accesses (DCACCESS) and data cache misses (DCMISS).

Example:

```
BMC.<counter>.EVENT DCMISS

BMC.<counter>.RATIO X/DCACCESS
```

See also

- BMC.<counter>

BMC.<counter>.SIZE Specify counter size

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

Specifies the width of a counter. Counters are cascaded to provide a counter of a bigger size.

Example:

```
BMC.<counter>.SIZE 32BIT
```

See also

- BMC.<counter>

BMC.Attach BMC attach

Format: **BMC.Attach**

Attaches to the BenchMark Counters without initializing the counter values to zero. This command is needed when the counters are configured by the target application.

See also

■ BMC ■ BMC.state

BMC.AutoInit Automatic initialization

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

If this command is set to ON, The **BMC.Init** command will be executed automatically, when the user program is started.

See also

■ BMC ■ BMC.state

BMC.CLOCK Provide core clock for cycle counter

Format: **BMC.CLOCK** *<clock>*

TRACE32 calculates and displays time information, if clock cycles are counted and the core clock is known.

Example:

```
BMC.<counter> ClockCylces

BMC.CLOCK 450.Mhz
```

See also

■ BMC ■ BMC.state ❑ BMC.CLOCK()

BMC.Init Initialize counters

Format: **BMC.Init**

All counters are set to their initialization values.

See also

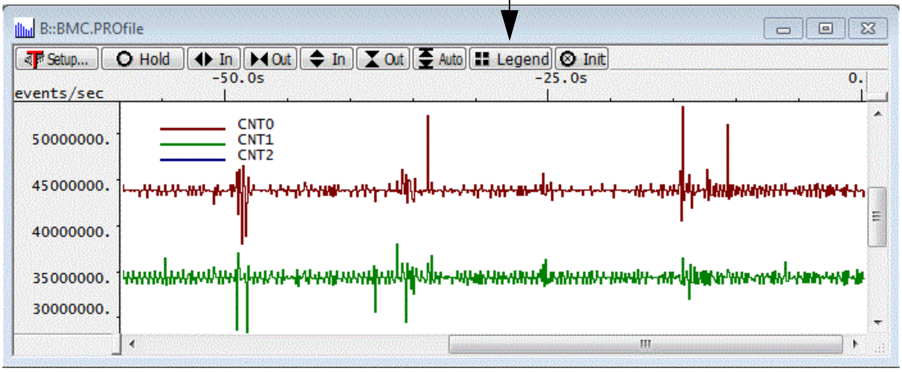
- ■ BMC
- ■ BMC.state

BMC.PROfile Display counter changes per second

Format: **BMC.PROfile** [*<y_scale>*]

If the target system allows to read the event counters while the program execution is running, TRACE32 can sample the values of up to three counters periodically. The counter changes per second are displayed graphically. The default sampling rate is 10 times per second.

Push **Legend** to get a color legend



See also

- ■ BMC
- ■ BMC.state

BMC.PROfileChart — Profile chart with benchmark counter

The **BMC.PROfileChart** command group displays distributions versus time graphically similar to **<trace>.PROfileChart**. The recorded instruction flow is synthesized with recorded benchmark counter information to display the run-time analysis.

NOTE:	Please note that the **BMC.PROfileChart** commands are only supported if the trace logic of the target processor generates BMC counter information via trace messages. Please refer to your **Processor Architecture Manual** for more information.

See also

- <trace>.PROfileChart.TASKVSINTERRUPT
- BMC.PROfileChart.DatasYmbol
- BMC.PROfileChart.GROUP
- BMC.PROfileChart.MODULE
- BMC.PROfileChart.sYmbol
- BMC.PROfileChart.TASKINFO
- BMC.PROfileChart.TASKKernel
- BMC.PROfileChart.TASKSRV
- BMC.PROfileSTATistic
- BMC.state
- <trace>.PROfileChart
- BMC.PROfileChart.AddressGROUP
- BMC.PROfileChart.DistriB
- BMC.PROfileChart.Line
- BMC.PROfileChart.PROGRAM
- BMC.PROfileChart.TASK
- BMC.PROfileChart.TASKINTR
- BMC.PROfileChart.TASKORINTERRUPT
- BMC.PROfileChart.TASKVSINTR
- BMC
- BMC.STATistic

▲ ’Release Information’ in ’Legacy Release History’

BMC.PROfileChart.AddressGROUP — Address group profile chart with BMC

Format:	**BMC.PROfileChart.AddressGROUP** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart for address **groups**. The results include groups for both program and data.

Refer to **<trace>.PROfileChart.AddressGROUP** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.AddressGROUP
- BMC.PROfileChart.GROUP

BMC.PROfileChart.DatasYmbol   Pointer profile chart with BMC

Format: **BMC.PROfileChart.DatasYmbol** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart for debug symbols with addresses corresponding to the accessed data values in the trace.

Refer to **<trace>.PROfileChart.DatasYmbol** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.DatasYmbol

BMC.PROfileChart.DistriB   Distribution display with BMC

Format: **BMC.PROfileChart.DistriB** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a graphical representation of the specified trace item as a percentage of a time slice.

Refer to **<trace>.PROfileChart.DistriB** for a description of the parameters and options.

See also

- BMC.PROfileChart

BMC.PROfileChart.GROUP   Group profile chart with BMC

Format: **BMC.PROfileChart.GROUP** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart for groups created with the **GROUP.Create** command. The results only include groups within the program range. Groups for data addresses are not included.

Refer to **<trace>.PROfileChart.GROUP** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.GROUP
- BMC.PROfileChart.AddressGROUP

BMC.PROfileChart.Line

Source code line profile chart with BMC

Format: **BMC.PROfileChart.Line** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart for high-level source code lines.

Refer to **<trace>.PROfileChart.Line** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.Line

BMC.PROfileChart.MODULE

Module profile chart with BMC

Format: **BMC.PROfileChart.MODULE** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of symbol modules. The list of loaded modules can be displayed with **sYmbol.List.Module**.

Refer to **<trace>.PROfileChart.MODULE** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.MODULE

Format: **BMC.PROfileChart.PROGRAM** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of loaded object file programs. The loaded programs can be displayed with the command **sYmbol.Browse \\***.

Refer to **<trace>.PROfileChart.PROGRAM** for a description of the parameters and options.

See also

■ BMC.PROfileChart ■ <trace>.PROfileChart.PROGRAM

BMC.PROfileChart.sYmbol Symbol profile chart with BMC

Format:	**BMC.PROfileChart.sYmbol** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink (command **Trace.METHOD**) is synthesized with recorded benchmark counter information in order to display profile chart for debug symbols.

Refer to **<trace>.PROfileChart.sYmbol** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.sYmbol

BMC.PROfileChart.TASK Task profile chart with BMC

Format:	**BMC.PROfileChart.TASK** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of OS tasks. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.PROfileChart.TASK** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASK

BMC.PROfileChart.TASKINFO Data trace via context ID with BMC

Format:	**BMC.PROfileChart.TASKINFO** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of special messages written to the Context ID register for ETM trace.

Refer to **<trace>.PROfileChart.TASKINFO** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASKINFO

BMC.PROfileChart.TASKINTR ISR2 profile chart with BMC

Format: **BMC.PROfileChart.TASKINTR** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of ORTI based ISR2. This feature can only be used if ISR2 can be traced based on the information provided by the ORTI file. Please refer to **"OS Awareness Manual OSEK/ORTI"** (rtos_orti.pdf) for more information.

Refer to **<trace>.PROfileChart.TASKINTR** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASKINTR

BMC.PROfileChart.TASKKernel Task profile chart with BMC

Format: **BMC.PROfileChart.TASKKernel** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of Tasks with kernel marker. This feature is only available if TRACE32 has been set for OS-aware debugging. Refer to **Trace.STATistic.TASKKernel** for more information.

Refer to **<trace>.PROfileChart.TASKKernel** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASKKernel

BMC.PROfileChart.TASKORINTERRUPT Task and interrupts with BMC

Format: **BMC.PROfileChart.TASKORINTERRUPT** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of OS tasks and interrupts. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.PROfileChart.TASKORINTERRUPT** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASKORINTERRUPT

BMC.PROfileChart.TASKSRV OS service routines profile chart with BMC

Format: **BMC.PROfileChart.TASKSRV** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of OS service routines.

This feature is only available if an OSEK/ORTI system is used and if the OS Awareness is configured with the **TASK.ORTI** command. Please refer to **“OS Awareness Manual OSEK/ORTI”** (rtos_orti.pdf) for more information.

Refer to **<trace>.PROfileChart.TASKSRV** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASKSRV

BMC.PROfileChart.TASKVSINTR Task related intr. profile chart with BMC

Format: **BMC.PROfileChart.TASKVSINTR** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a profile chart of task-related interrupt service routines.

This feature is only available if an OSEK/ORTI system is used and if the OS Awareness is configured with the **TASK.ORTI** command. Please refer to **“OS Awareness Manual OSEK/ORTI”** (rtos_orti.pdf) for more information.

Refer to **<trace>.PROfileChart.TASKVSINTR** for a description of the parameters and options.

See also

- BMC.PROfileChart
- <trace>.PROfileChart.TASKVSINTR

BMC.PROfileSTATistic Statistical analysis vs. time with benchmark counter

The **BMC.PROfileSTATistic** command group shows the results of numerical interval analysis in tabular format. **<trace>.PROfileSTATistic**. The recorded instruction flow is synthesized with recorded benchmark counter information to display the run-time analysis.

NOTE:	Please note that the **BMC.PROfileSTATistic** commands are only supported if the trace logic of the target processor generates BMC counter information via trace messages. Please refer to your **Processor Architecture Manual** for more information.

See also

- <trace>.PROfileSTATistic.TASKVSINTERRUPT
- BMC.PROfileSTATistic.AddressGROUP
- BMC.PROfileSTATistic.DistriB
- BMC.PROfileSTATistic.INTERRUPT
- BMC.PROfileSTATistic.MODULE
- BMC.PROfileSTATistic.RUNNABLE
- BMC.PROfileSTATistic.TASK
- BMC.PROfileSTATistic.TASKINTR
- BMC.PROfileSTATistic.TASKORINTERRUPT
- BMC.PROfileChart
- BMC.state
- BMC.PROfileSTATistic.Address
- BMC.PROfileSTATistic.DatasYmbol
- BMC.PROfileSTATistic.GROUP
- BMC.PROfileSTATistic.Line
- BMC.PROfileSTATistic.PROGRAM
- BMC.PROfileSTATistic.sYmbol
- BMC.PROfileSTATistic.TASKINFO
- BMC.PROfileSTATistic.TASKKernel
- BMC.PROfileSTATistic.TASKSRV
- BMC
- <trace>.PROfileSTATistic

BMC.PROfileSTATistic.Address Address statistical analysis with BMC

Format:	**BMC.PROfileSTATistic.Address** [*<trace_area>*] *<address1>* [*<address2>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for addresses.

Refer to **<trace>.PROfileSTATistic.Address** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.Address

BMC.PROfileSTATistic.AddressGROUP Address group statistic with BMC

Format: **BMC.PROfileSTATistic.AddressGROUP**[*<trace_area>*][**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for address **groups**. The results include groups for both program and data.

Refer to **<trace>.PROfileSTATistic.AddressGROUP** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.AddressGROUP
- BMC.PROfileSTATistic.GROUP

BMC.PROfileSTATistic.DatasYmbol Pointer profile statistic with BMC

Format: **BMC.PROfileSTATistic.DatasYmbol** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistic analysis versus time for debug symbols with addresses corresponding to the accessed data values in the trace.

Refer to **<trace>.PROfileSTATistic.DatasYmbol** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.DatasYmbol

BMC.PROfileSTATistic.DistriB Distribution statistical analysis with BMC

Format: **BMC.PROfileSTATistic.DistriB** [**%***<format>*] [*<items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistic analysis versus time for the selected *<items>*. Without *<items>* the statistic is based on the symbolic addresses.

Refer to **<trace>.PROfileSTATistic.DistriB** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.DistriB

BMC.PROfileSTATistic.GROUP   Group profile statistic with BMC

Format:	**BMC.PROfileSTATistique.GROUP** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for groups created with the **GROUP.Create** command. The results only include groups within the program range. Groups for data addresses are not included.

Refer to **<trace>.PROfileSTATistic.GROUP** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- BMC.PROfileSTATistic.AddressGROUP
- <trace>.PROfileSTATistic.GROUP

BMC.PROfileSTATistic.INTERRUPT   Interrupt profile statistic with BMC

Format:	**BMC.PROfileSTATistique.INTERRUPT** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for interrupts. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.PROfileSTATistic.INTERRUPT** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.INTERRUPT

BMC.PROfileSTATistic.Line High-level code line profile statistic with BMC

Format: **BMC.PROfileSTATistic.Line** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for high-level source code lines.

Refer to **<trace>.PROfileSTATistic.Line** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.Line

BMC.PROfileSTATistic.MODULE Module profile statistic with BMC

Format: **BMC.PROfileSTATistic.MODULE** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for symbol modules. The list of loaded modules can be displayed with **sYmbol.List.Module**.

Refer to **<trace>.PROfileSTATistic.MODULE** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.MODULE

BMC.PROfileSTATistic.PROGRAM Program profile statistic with BMC

Format: **BMC.PROfileSTATistic.PROGRAM** [<*trace_area*>] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for loaded object file programs. The loaded programs can be displayed with the command **sYmbol.Browse \\***.

Refer to **<trace>.PROfileSTATistic.PROGRAM** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.PROGRAM

BMC.PROfileSTATistic.RUNNABLE   Runnable profile statistic with BMC

Format: **BMC.PROfileSTATistic.RUNNABLE** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for AUTOSAR runnables. This feature is only available if an OSEK/ORTI system is used and if the OS Awareness is configured with the **TASK.ORTI** command. Please refer to **"OS Awareness Manual OSEK/ORTI"** (rtos_orti.pdf) for more information.

Refer to **<trace>.PROfileSTATistic.RUNNABLE** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.RUNNABLE

BMC.PROfileSTATistic.sYmbol   Symbol profile statistic with BMC

Format: **BMC.PROfileSTATistic.sYmbol** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink (command **Trace.METHOD**) is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for debug symbols.

Refer to **<trace>.PROfileSTATistic.sYmbol** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.sYmbol

BMC.PROfileSTATistic.TASK   Task profile statistic with BMC

Format: **BMC.PROfileSTATistic.TASK** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for OS tasks. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.PROfileSTATistic.TASK** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.TASK

BMC.PROfileSTATistic.TASKINFO Data trace via context ID with BMC

Format:	**BMC.PROfileSTATistic.TASKINFO** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time of special messages written to the Context ID register for ETM trace.

Refer to **<trace>.PROfileSTATistic.TASKINFO** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.TASKINFO

BMC.PROfileSTATistic.TASKINTR ISR2 profile statistic with BMC

Format:	**BMC.PROfileSTATistic.TASKINTR** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for ORTI based ISR2. This feature can only be used if ISR2 can be traced based on the information provided by the ORTI file. Please refer to **“OS Awareness Manual OSEK/ORTI”** (rtos_orti.pdf) for more information.

Refer to **<trace>.PROfileSTATistic.TASKINTR** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.TASKINTR

BMC.PROfileSTATistic.TASKKernel Task profile statistic with BMC

Format: **BMC.PROfileSTATistic.TASKKernel** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for Tasks with kernel marker. Refer to **Trace.STATistic.TASKKernel** for more information.

This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.PROfileSTATistic.TASKKernel** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.TASKKernel

BMC.PROfileSTATistic.TASKORINTERRUPT Task or interrupt with BMC

Format: **BMC.PROfileSTATistic.TASKORINTERRUPT** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for OS tasks and interrupts. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.PROfileSTATistic.TASKORINTERRUPT** for a description of the parameters and options.

See also

- BMC.PROfileSTATistic
- <trace>.PROfileSTATistic.TASKORINTERRUPT

BMC.PROfileSTATistic.TASKSRV OS service routines profile stat. with BMC

Format: **BMC.PROfileSTATistic.TASKSRV** [*<trace_area>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis versus time for OS service routines. This feature is only available if an OSEK/ORTI system is used and if the OS Awareness is configured with the **TASK.ORTI** command. Please refer to **“OS Awareness Manual OSEK/ORTI”** (rtos_orti.pdf) for more information.

Refer to **<trace>.PROfileSTATistic.TASKSRV** for a description of the parameters and options.

See also

- ■ BMC.PROfileSTATistic
- ■ <trace>.PROfileSTATistic.TASKSRV

BMC.RESet Reset benchmark counter configuration

Format: **BMC.RESet**

Resets the BenchMark Counter configuration to the default settings.

See also

- BMC
- BMC.state

BMC.SnoopSet Assign event counter to SNOOPer trace

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

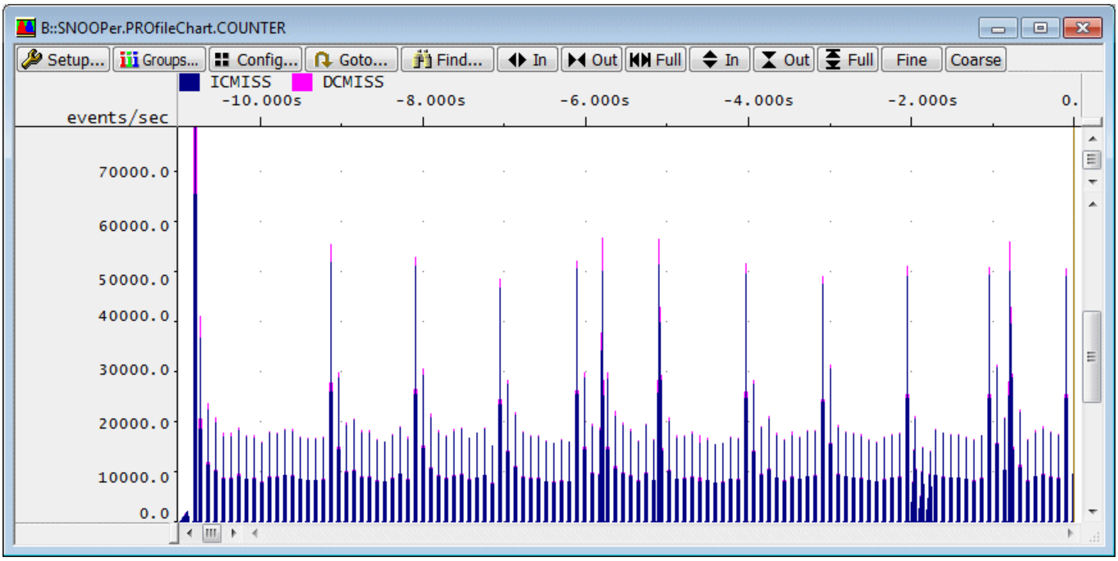
The TRACE32 SNOOPer Trace can be used to record the event counters periodically, if the target system allows to read the event counters while the program execution is running.

TRACE32 provides various ways to analyze the recorded information.

Example 1 for the pure JTAG debugger.

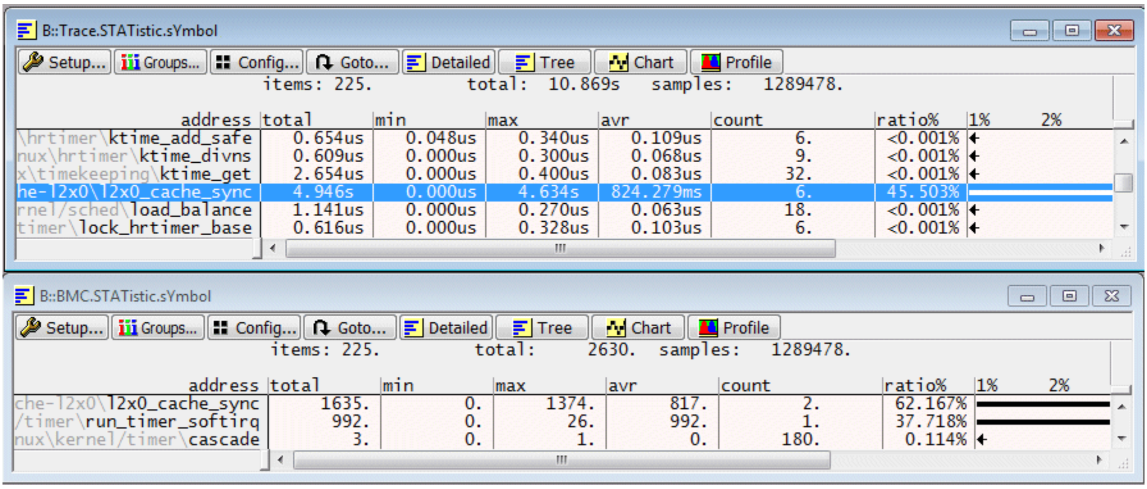
```
BMC.state                             ; display the BMC Configuration
                                      ; window

BMC.<counter1> <event1>               ; assign event of interest to
                                      ; the event counter

;BMC.<counter2> <event2>              ; several assignments possible
…

BMC.SnoopSet ON                       ; configure the TRACE32 SNOOPer
                                      ; Trace for event counter recording

SNOOPer.state                         ; display the SNOOPer Trace
                                      ; Configuration window to inspect
                                      ; the setup

Go                                    ; start the program execution to
                                      ; fill the SNOOPer trace
```

```
Break                              ; stop the program execution

SNOOPer.List                       ; display a SNOOPer trace listing

                                   ; please pay attention to the
                                   ; ti.back time, it informs you on
                                   ; the SNOOPer sampling rate

SNOOPer.PROfileChart.COUNTER       ; display a profile statistic
```



Example 2: In this script, an event counter recording is combined with an instruction flow trace recording.

```
BMC.state                          ; display the BMC Configuration
                                   ; window

BMC.<counter1> <event1>            ; assign event of interest to
                                   ; event counter

                                   ; only one event counter possible

BMC.SnoopSet ON                    ; configure the TRACE32 SNOOPer
                                   ; Trace for event counter recording

SNOOPer.state                      ; display the SNOOPer Trace
                                   ; Configuration window to inspect
                                   ; the setup

SNOOPer.SIZE 500000.               ; adjust the size of the SNOOPER
                                   ; Trace

                                   ; the SNOOPer Trace and the Trace
                                   ; recording the instruction flow
                                   ; should get full nearly at the
                                   ; same point in time
; initialize all units involved whenever the program execution is
; started, this avoids invalid combinations

Trace.AutoInit ON                    ; initialize the Trace recording
                                     ; the instruction flow

SNOOPer.AutoInit ON                  ; initialize the SNOOPER Trace

BMC.AutoInit ON                      ; initialize the event counter

Go                                   ; start the program execution to
                                     ; fill the SNOOPer trace

Break                                ; stop the program execution

SNOOPer.List                         ; display a SNOOPer trace listing

                                     ; please pay attention to the
                                     ; ti.back time, it informs you on
                                     ; the SNOOPer sampling rate

BMC.SELect <counter1>                ; select <counter1> for the
                                     ; statistic evaluation

BMC.STATistic.sYmbol                 ; assign the recorded events to the
                                     ; recorded functions/symbol ranges
```



```
B::Trace.STATistic.sYmbol
                    items: 225.         total:  10.869s   samples:    1289478.

             address total       min         max         avr         count          ratio%  1%     2%
\hrtimer\ktime_add_safe   0.654us     0.048us     0.340us     0.109us             6.    <0.001%
nux\hrtimer\ktime_divns   0.609us     0.000us     0.300us     0.068us             9.    <0.001%
x\timekeeping\ktime_get   2.654us     0.000us     0.400us     0.083us            32.    <0.001%
he-l2x0\l2x0_cache_sync   4.946s      0.000us     4.634s    824.279ms             6.    45.503%
rnel/sched\load_balance   1.141us     0.000us     0.270us     0.063us            18.    <0.001%
timer\lock_hrtimer_base   0.616us     0.000us     0.328us     0.103us             6.    <0.001%
```

```
B::BMC.STATistic.sYmbol
                    items: 225.         total:      2630.   samples:    1289478.

              address total       min         max         avr         count          ratio%  1%     2%
che-l2x0\l2x0_cache_sync      1635.          0.       1374.        817.             2.   62.167%
/timer\run_timer_softirq       992.          0.         26.        992.             1.   37.718%
nux\kernel/timer\cascade         3.          0.          1.          0.           180.    0.114%
```

See also

- BMC
- BMC.state

[Step-by-Step Procedure] [Example]

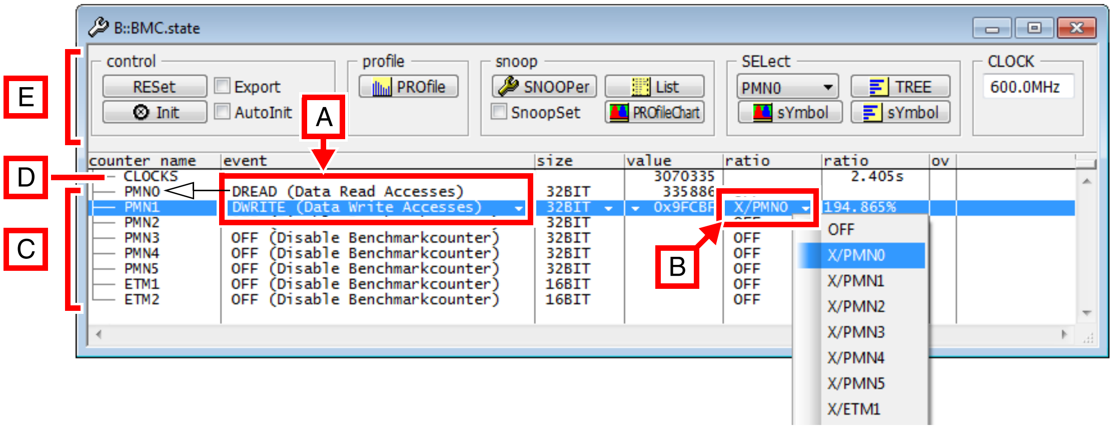
Format:	**BMC.state**

Displays the **BMC.state** window, where you can assign events to benchmark counters in order to count these events and compare one counter in relation to another counter. The benchmarking results are displayed in the **BMC.state** window.

NOTE: The layout and operating principle of the **BMC.state** window is the same for most TRACE32 debuggers, i.e. the window is architecture-*independent*.

- For a few TRACE32 debuggers, the layout of the **BMC.state** window remains architecture-specific because some chips offers only a limited benchmark counter functionality.
- Architecture-specific **BMC** commands are described in the TRACE32 processor architecture manuals.
  Choose **Help** menu > **Processor Architecture Manual**.

Description of Header and Columns: BMC.state Window (Using an OMAP4430 as an Example)



counter name	event	size	value	ratio	ratio	ov
CLOCKS			3070335		2.405s	
PMN0	DREAD (Data Read Accesses)	32BIT	335886			
PMN1	DWRITE (Data Write Accesses)	32BIT	0x9FCBF	X/PMN0	194.865%	
PMN2		32BIT				
PMN3	OFF (Disable Benchmarkcounter)	32BIT		OFF		
PMN4	OFF (Disable Benchmarkcounter)	32BIT		OFF		
PMN5	OFF (Disable Benchmarkcounter)	32BIT		OFF		
ETM1	OFF (Disable Benchmarkcounter)	16BIT		OFF		
ETM2	OFF (Disable Benchmarkcounter)	16BIT		OFF		

A The **BMC.state** window shows how two events, the **DREAD** and **DWRITE** events, can be counted by assigning them to two benchmark counters, **PMN0** and **PMN1**.

B The first **ratio** column lets you analyze one benchmark counter in relation to another benchmark counter. Here, the **PMN1** counter is analyzed in relation to the **PMN0** counter. The result is displayed in the second **ratio** column. See also **BMC.<counter>.RATIO**.

- For the CLOCKS benchmark counter, the runtime is given in seconds. This value is calculated from the clock frequency and the cycle count.
- For the other benchmark counters, the results are given in percentage, seconds, or Hertz.

C **counter name**. Performance counters from the core debug controller. The counter names are architecture specific.

D **counter name**. **CLOCKS**: The clock cycle counter is activated if at least one of the performance counters of the core debug controller is activated (not available on all cores).

E **Header**. For descriptions of the commands in the **BMC.state** window, please refer to the **BMC.*** commands in this chapter.
Example: For information about the **AutoInit** check box, see **BMC.AutoInit**.

- **event**.The drop-down list shows the name of the event together with a short description in parentheses. The available events are device-specific. See **BMC.<counter>.EVENT**.

- **size**. Displays the size of the performance counters. For architectures providing variable counter sizes, the counter size can be adjusted with the **BMC.<counter>.SIZE** command.

- **value**. Number of hardware events counted. Right-click to display the value as decimal or hex. In a PRACTICE script, you can format the value as hex or decimal using the command **BMC.<counter>.FORMAT**, see example.

- **ratio**. See [**B**].

- **ov**. Counter overflow.

To Assign Events to Benchmark Counters via the User Interface TRACE32 PowerView:

1. At the TRACE32 command line type, **BMC.state** to open the window.

2. In the **counter name** column, click the benchmark counter you want to configure.

   The selected row is highlighted in blue. Little white down-arrows indicate that you can configure the values in these columns via drop-down lists [**A**].

3. In the **event** column, right-click the white down-arrow, and then select the event to be counted [**B**].

PRACTICE Script Example for the OMAP4430:

```
BMC.state                   ;open the BMC.state window

BMC.CLOCK 600.0MHz          ;baseline for all benchmark counter
                            ;calculations

;columns 'counter name' and 'event'
BMC.PMN0.EVENT  DREAD       ;assign the DREAD event to the PMN0 counter
BMC.PMN1.EVENT  DWRITE      ;assign the DWRITE event to the PMN1 counter

;'value' column             ;for demo purposes let's format the value
BMC.PMN1.FORMAT HEXadecimal ;of PMN1 as hex

;'ratio' column
BMC.PMN1.RATIO  X/PMN0      ;analyze PMN1 in relation to PMN0

BMC.PROfile                 ;the BMC.PROfile window displays the current
                            ;number of events per second.
                            ;______ if 0 events.

Go                          ;start real-time emulation - the BMC windows
WAIT 1.s                    ;are updated while the emulation is running
Break                       ;stop emulation
```

See also

- ■ BMC
- ■ BMC.<counter>
- ■ BMC.Attach
- ■ BMC.AutoInit
- ■ BMC.CLOCK
- ■ BMC.Init
- ■ BMC.PROfile
- ■ BMC.PROfileChart
- ■ BMC.PROfileSTATistic
- ■ BMC.RESet
- ■ BMC.SnoopSet
- ■ BMC.STATistic

▲ ’Release Information’ in ’Legacy Release History’

BMC.STATistic — Statistic analysis with benchmark counter

The **BMC.STATistic** command group can be used for statistical analysis based on the information sampled to the trace buffer similar to **<trace>.STATistic**. The recorded instruction flow is additionally synthesized with recorded benchmark counter information to display the run-time analysis.

NOTE:	Please note that the **BMC.STATistic** commands are only supported if the trace logic of the target processor generates BMC counter information via trace messages. Please refer to your **Processor Architecture Manual** for more information.

See also

- <trace>.STATistic.TASKVSINTERRUPT
- BMC.STATistic.DistriB
- BMC.STATistic.GROUP
- BMC.STATistic.MODULE
- BMC.STATistic.PROGRAM
- BMC.STATistic.TASK
- BMC.STATistic.TASKINTR
- BMC.STATistic.TASKORINTERRUPT
- BMC.STATistic.TREE
- BMC.PROfileChart
- <trace>.STATistic
- BMC.STATistic.ChildTREE
- BMC.STATistic.Func
- BMC.STATistic.LINKage
- BMC.STATistic.ParentTREE
- BMC.STATistic.sYmbol
- BMC.STATistic.TASKINFO
- BMC.STATistic.TASKKernel
- BMC.STATistic.TASKSRV
- BMC
- BMC.state

BMC.STATistic.ChildTREE — Function callee context with BMC

Format:	**BMC.STATistic.ChildTREE** *<address>* [*<list_items>*] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display the call tree and run-time of all functions called by the function specified with the *<address>* parameter.

Refer to **<trace>.STATistic.ChildTREE** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.ChildTREE

BMC.STATistic.DistriB — Distribution analysis with BMC

Format: **BMC.STATistic.DistriB** [**%**<*format*>] [<*items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display the statistic distribution of the selected <*items*>. Without <*items*> the statistic is based on the symbolic addresses.

Refer to **<trace>.STATistic.DistriB** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.DistriB

BMC.STATistic.Func — Nesting function run-time with BMC

Format: **BMC.STATistic.Func** [**%**<*format*>] [<*list_items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a nesting function run-time analysis.

Refer to **<trace>.STATistic.Func** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.Func

BMC.STATistic.GROUP — Group run-time analysis with BMC

Format: **BMC.STATistic.GROUP** [**%**<*format*>] [<*list_items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a run-time analysis for groups created with the **GROUP.Create** command. The results only include groups within the program range. Groups for data addresses are not included.

Refer to **<trace>.STATistic.GROUP** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.GROUP

BMC.STATistic.LINKage Per caller function statistic with BMC

Format: **BMC.STATistic.LINKage** *<address>* [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a function run-time statistic for a single function itemized by its callers.

Refer to **<trace>.STATistic.LINKage** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.LINKage

BMC.STATistic.MODULE Module statistic with BMC

Format: **BMC.STATistic.MODULE** [**%***<format>*] [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of symbol modules. The list of loaded modules can be displayed with **sYmbol.List.Module**.

Refer to **<trace>.STATistic.MODULE** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.MODULE

BMC.STATistic.ParentTREE Statistic for call context with BMC

Format: **BMC.STATistic.ParentTREE** *<address>* [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of all callers of the specified function. The function is specified by its start *<address>*.

Refer to **<trace>.STATistic.ParentTREE** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.ParentTREE

BMC.STATistic.PROGRAM

Program statistic with BMC

Format: **BMC.STATistic.MODULE** [**%**<*format*>] [<*list_items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of loaded object file programs. The loaded programs can be displayed with the command **sYmbol.Browse \\***.

Refer to **<trace>.STATistic.PROGRAM** for a description of the parameters and options.

See also

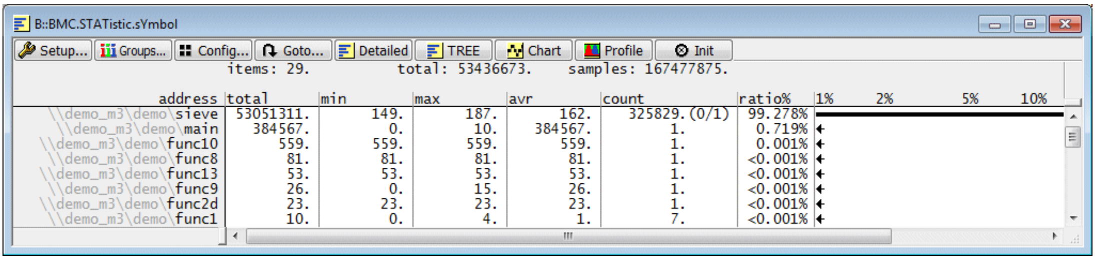
- BMC.STATistic
- <trace>.STATistic.PROGRAM

BMC.STATistic.sYmbol

Flat run-time analysis with BMC

Format: **BMC.STATistic.sYmbol** [**%**<*format*>] [<*list_items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink (command **Trace.METHOD**) is synthesized with recorded benchmark counter information in order to display a flat function run-time analysis.



```
                       items: 29.            total: 53436673.     samples: 167477875.

              address |total      |min        |max        |avr        |count         |ratio% |1%     2%       5%     10%
  \\demo_m3\demo\sieve | 53051311. |      149. |      187. |      162. |  325829.(0/1)| 99.278%
   \\demo_m3\demo\main |   384567. |        0. |       10. |   384567. |            1.|  0.719%
\\demo_m3\demo\func10 |      559. |      559. |      559. |      559. |            1.|  0.001%
  \\demo_m3\demo\func8 |       81. |       81. |       81. |       81. |            1.| <0.001%
\\demo_m3\demo\func13 |       53. |       53. |       53. |       53. |            1.| <0.001%
  \\demo_m3\demo\func9 |       26. |        0. |       15. |       26. |            1.| <0.001%
\\demo_m3\demo\func2d |       23. |       23. |       23. |       23. |            1.| <0.001%
  \\demo_m3\demo\func1 |       10. |        0. |        4. |        1. |            7.| <0.001%
```

Refer to **<trace>.STATistic.sYmbol** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.sYmbol

BMC.STATistic.TASK Statistic for tasks with BMC

Format:	**BMC.STATistic.TASK** [**%***<format>*] [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of OS tasks. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.STATistic.TASK** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TASK

BMC.STATistic.TASKINFO Statistic for context ID messages with BMC

Format:	**BMC.STATistic.TASKINFO** [**%***<format>*] [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of special messages written to the Context ID register for ETM trace. Refer to **<trace>.STATistic.TASKINFO** for more information.

Refer to **<trace>.STATistic.TASKINFO** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TASKINFO

BMC.STATistic.TASKINTR Statistic for ISR2 with BMC

Format:	**BMC.STATistic.TASKINTR** [**%***<format>*] [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of ORTI based ISR2. This feature can only be used if ISR2 can be traced based on the information provided by the ORTI file. Please refer to **“OS Awareness Manual OSEK/ORTI”** (rtos_orti.pdf) for more information.

Refer to **<trace>.STATistic.TASKINTR** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TASKINTR

BMC.STATistic.TASKKernel Statistic for tasks with BMC

Format: **BMC.STATistic.TASKKernel** [**%**<*format*>] [<*list_items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of tasks with kernel marker. Refer to **Trace.STATistic.TASKKernel** for more information. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.STATistic.TASKKernel** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TASKKernel

BMC.STATistic.TASKORINTERRUPT Tasks and interrupts with BMC

Format: **BMC.STATistic.TASKORINTERRUPT** [**%**<*format*>] [<*list_items*> …] [**/**<*option*>]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of OS tasks and interrupts. This feature is only available if TRACE32 has been set for OS-aware debugging.

Refer to **<trace>.STATistic.TASKORINTERRUPT** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TASKORINTERRUPT

BMC.STATistic.TASKSRV Statistic for OS service routines with BMC

Format: **BMC.STATistic.TASKSRV** [**%***<format>*] [*<list_items>* …] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a statistical analysis of OS service routines. This feature is only available if an OSEK/ORTI system is used and if the OS Awareness is configured with the **TASK.ORTI** command. Please refer to **"OS Awareness Manual OSEK/ORTI"** (rtos_orti.pdf) for more information.

Refer to **<trace>.STATistic.TASKSRV** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TASKSRV

BMC.STATistic.TREE Tree nesting function run-time with BMC

Format: **BMC.STATistic.TREE** [**%***<format>*] [{*<list_items>*}] [**/***<option>*]

The instruction flow recorded to the selected trace sink is synthesized with recorded benchmark counter information in order to display a graphical tree of the function nesting.

Refer to **<trace>.STATistic.TREE** for a description of the parameters and options.

See also

- BMC.STATistic
- <trace>.STATistic.TREE

BookMark — Address and trace bookmarks

See also

- BookMark.CHange
- BookMark.Create
- BookMark.Delete
- BookMark.EditRemark
- BookMark.EXPORT
- BookMark.List
- BookMark.RESet
- BookMark.Toggle
- <trace>.BookMark
- <trace>.BookMarkToggle
- <trace>.GOTO
- <trace>.TRACK
- COVerage

Overview BookMark

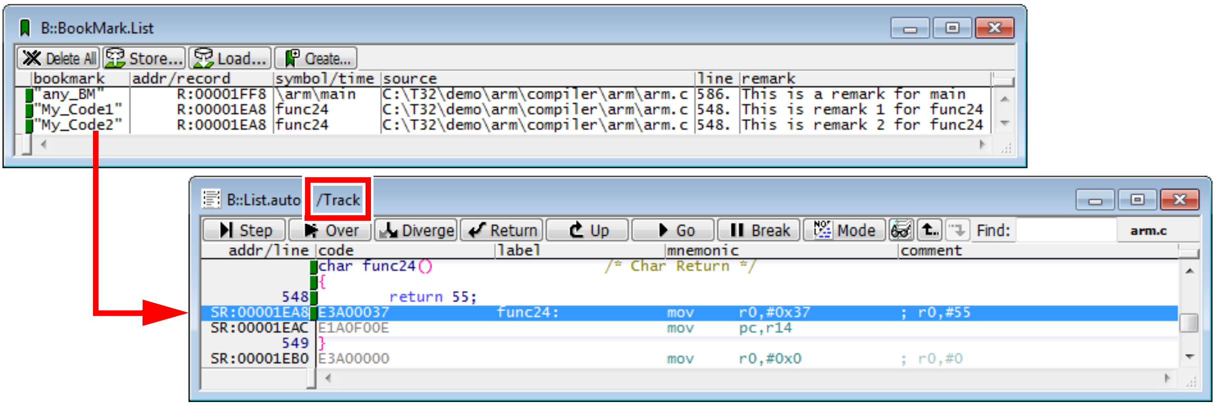
NOTE:	Bookmark names are case sensitive.

There are two types of bookmarks, which are distinguished by their color:

- **Address bookmarks** are marked with a small green rectangle.
- **Trace bookmarks** are marked with a small yellow rectangle.

Using bookmarks, you can mark, locate, and identify trace records of interest or addresses of interest. For code coverage, you can use bookmarks to add comments to not-executed code.

It is recommended that you use bookmarks together with the **/Track** option to improve navigation: Let's assume that the **List.auto /Track** window is already open. When you *single*-click any of the address bookmarks in the **BookMark.List** window, the cursor in the **List.auto /Track** window automatically points to the corresponding assembler code. See figure below.



```
 bookmark     addr/record  symbol/time source                           line remark
 "any_BM"      R:00001FF8  \arm\main   C:\T32\demo\arm\compiler\arm\arm.c 586. This is a remark for main
 "My_Code1"    R:00001EA8  func24      C:\T32\demo\arm\compiler\arm\arm.c 548. This is remark 1 for func24
 "My_Code2"    R:00001EA8  func24      C:\T32\demo\arm\compiler\arm\arm.c 548. This is remark 2 for func24
```

```
     addr/line code                     label          mnemonic                  comment
               char func24()                           /* Char Return */
               {
           548       return 55;
 SR:00001EA8   E3A00037                 func24:        mov      r0,#0x37         ; r0,#55
 SR:00001EAC   E1A0F00E                                mov      pc,r14
           549 }
 SR:00001EB0   E3A00000                                mov      r0,#0x0          ; r0,#0
```

When you *double*-click an address bookmark in the **BookMark.List** window, a new **List** window opens at the bookmarked address.

When you *double*-click a trace bookmark in the **BookMark.List** window, a new **<trace>.List** window opens at the bookmarked trace record.

BookMark.CHange

Edit the settings of a bookmark

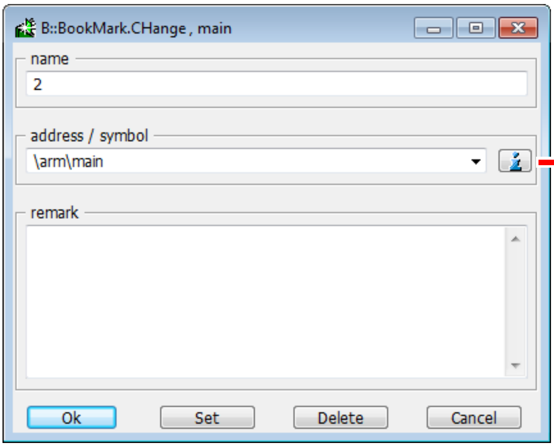
Format:	**BookMark.CHange "***<bookmark_name>***"** *<address>* | *<time>* [*<file>*] [*<line>*]

Opens a dialog where you can change the settings of a bookmark and rename the bookmark. In addition, you can use the **BookMark.CHange** command to create a new bookmark. Alternatively, you can right-click the desired bookmark in the **BookMark.List** window, and then select **Change**.

<bookmark_name>	Bookmark names are case sensitive.
<time>, <file>, <line>	The parameters *<time>*, *<file>*, and *<line>* are reserved for the scripting-mode of TRACE32, they are not needed in the dialog-mode of TRACE32.

Example:

```
;displays the settings for the bookmark "Loop"
BookMark.CHange "Loop"

;TRACE32 suggests a new bookmark name by incrementing to the next
;bookmark number
BookMark.CHange

;the bookmark name is incremented, and the new bookmark will refer
;to the symbol main (see screenshot below)
BookMark.CHange , main
```


B::BookMark.CHange, main

name: 2

address / symbol: \arm\main

Opens the Browse Symbols dialog. (**sYmbol.Browse.sYmbol**)

See also

- ■ BookMark

Format:	**BookMark.Create** "*<bookmark_name>*" *<address>* | *<time>* [*<file>*] [*<line>*]

Creates a new address bookmark. If the *<bookmark_name>* exists already, the command **BookMark.Create** will overwrite the address bookmark with the new parameters.

NOTE:	To create a *trace bookmark*, use the **<trace>.BookMark** command.

<bookmark_name>	Bookmark names are case sensitive.
<time>, <file>, <line>	The parameters *<time>*, *<file>*, and *<line>* are reserved for the scripting-mode of TRACE32, they are not needed in the dialog-mode of TRACE32.

Examples:

```
; create a new bookmark at 0x1000 and label it "start"
BookMark.Create "start" 0x1000
```

```
; create a new bookmark at the entry of func24 and name it "My_Code"
BookMark.Create "My_Code" func24
```

```
; overwrites the existing bookmark called "My_Code" with the address
; 0x2000
BookMark.Create "My_Code" 0x2000
```

See also

■ BookMark ■ <trace>.BookMark

BookMark.Delete

Delete an existing bookmark

Format:	**BookMark.Delete "***<bookmark_name>***"** [*<address>* | *<time>*] [*<file>*] [*<line>*]

Deletes an existing bookmark.

<bookmark_name> Bookmark names are case sensitive.

<time>, <file>, <line> The parameters *<time>*, *<file>*, and *<line>* are reserved for the scripting-mode of TRACE32, they are not needed in the dialog-mode of TRACE32.

Examples:

```
BookMark.Delete "start"      ; Delete the bookmark named "start"

BookMark.Delete "My_Code"    ; Delete the bookmark "My_Code"
```

See also

- ■ BookMark

Format: **BookMark.EditRemark** "*<bookmark_name>*" [*<remark>*]

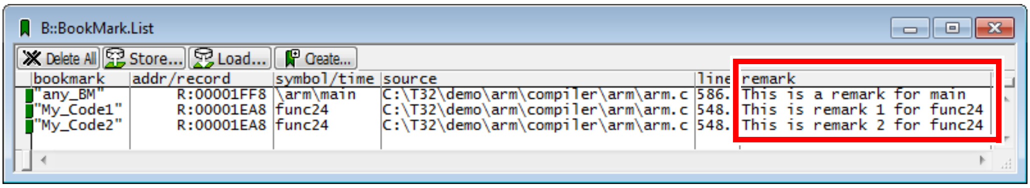
Adds a user-defined *<remark>* to a *<bookmark_name>*.

- To edit or delete a remark via the **BookMark.List** window, right-click the remark, and then select the desired option from the popup menu.
- To edit or delete a remark via the TRACE32 command line, assign the desired string or empty string to *<remark>*.

Adding another remark to the same bookmark-symbol combination overwrites the previous remark. However, you can add multiple remarks to the same symbol if you also assign multiple bookmarks to that symbol, as shown in the example below.

Example:

```
;open the Bookmark.List window
BookMark.List

;create a bookmark for symbol main and add a remark
BookMark.Create "any_BM" main
BookMark.EditRemark "any_BM" "This is a remark for main"

;create two new bookmarks at the entry of the symbol “func24”
;and name the bookmarks "My_Code1, My_Code2"
BookMark.Create "My_Code1" func24
BookMark.Create "My_Code2" func24

;for each bookmark of symbol “func24”, add one remark:
BookMark.EditRemark "My_Code1" "This is remark 1 for func24"
BookMark.EditRemark "My_Code2" "This is remark 2 for func24"
```



bookmark	addr/record	symbol/time	source	line	remark
"any_BM"	R:00001FF8	\arm\main	C:\T32\demo\arm\compiler\arm\arm.c	586.	This is a remark for main
"My_Code1"	R:00001EA8	func24	C:\T32\demo\arm\compiler\arm\arm.c	548.	This is remark 1 for func24
"My_Code2"	R:00001EA8	func24	C:\T32\demo\arm\compiler\arm\arm.c	548.	This is remark 2 for func24

See also

- BookMark
- <trace>.BookMark

See also

- BookMark.EXPORT.ADDRESS
- BookMark.EXPORT.SOURCE
- BookMark
- BookMark.EXPORT.preset
- BookMark.EXPORT.sYmbol

BookMark.EXPORT.ADDRESS Export bookmarks for specified addresses

Format: **BookMark.EXPORT.ADDRESS** *<xml_file> <address>*… [**/Append**]

Exports only those bookmarks to an XML file that have been created for the specified addresses.

<address>	Apply one or more address as filter criteria. Only bookmarks matching the specified addresses are exported.
Append	For a description and an example, see **BookMark.EXPORT**.

Example:

```
BookMark.EXPORT.ADDRESS ~~/bookmarks-addresses.xml 0x13ce 0x12aa
```

See also

- BookMark.EXPORT

BookMark.EXPORT.preset Export bookmarks to an XML file

Format: **BookMark.EXPORT.preset** *<file>* [*<range> <address>*] [**/Append**]

Exports all bookmarks to an XML file or just the bookmarks selected with *<range>* or *<address>*. The XML file is formatted by placing a transformation template (*.xsl) in the same folder as the XML file.

<range>	Range filter for exporting bookmarks that are located within a specified address range.

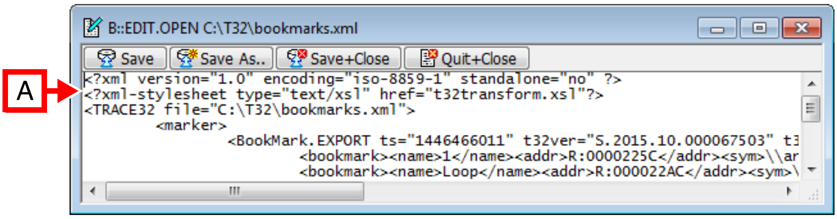
<address>	Address filter for exporting an individual bookmark located at a specified address.
Append	The bookmarks displayed in the **BookMark.List** window are appended at the end of the file.

Using the **STOre <file> BookMark** command, you can save the bookmark list as a PRACTICE script (*.cmm).

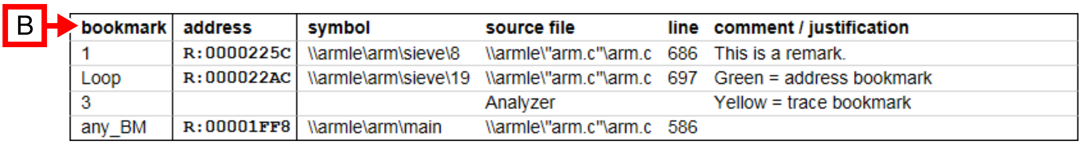
Example 1: All existing bookmarks are exported. The unformatted result is displayed in TRACE32, and the formatted result is displayed in a browser window.

```
;export all bookmarks
BookMark.EXPORT "~~/bookmarks.xml" ,

;for demo purposes: let's assume that you have added another bookmark
BookMark.Create "any_BM" R:0x1FF8    ;e.g. at this address

;append the new bookmark to the previous XML file
BookMark.EXPORT "~~/bookmarks.xml" R:0x1FF8 /Append

;for demo purposes: let's open the unformatted result in the internal
;TRACE32 editor
EDIT.OPEN "~~/bookmarks.xml"

;place the transformation template in the same folder as the XML file
COPY "~~/demo/coverage/single_file_report/t32transform.xsl" \
     "~~/t32transform.xsl"

;you can now open the formatted result in an external browser window
OS.Command start iexplore.exe "file:///C:/t32/bookmarks.xml"
```

The tildes ~~ expand to your TRACE32 system directory, by default c:\t32.



```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<?xml-stylesheet type="text/xsl" href="t32transform.xsl"?>
<TRACE32 file="C:\T32\bookmarks.xml">
        <marker>
                <BookMark.EXPORT ts="1446466011" t32ver="S.2015.10.000067503" t3
                        <bookmark><name>1</name><addr>R:0000225C</addr><sym>\\ar
                        <bookmark><name>Loop</name><addr>R:000022AC</addr><sym>\
```



bookmark	address	symbol	source file	line	comment / justification
1	R:0000225C	\\armle\arm\sieve\8	\\armle\"arm.c"\arm.c	686	This is a remark.
Loop	R:000022AC	\\armle\arm\sieve\19	\\armle\"arm.c"\arm.c	697	Green = address bookmark
3			Analyzer		Yellow = trace bookmark
any_BM	R:00001FF8	\\armle\arm\main	\\armle\"arm.c"\arm.c	586	

A Unformatted result.

B Example of a formatted result in a browser window.

Example 2: A more complex demo script is included in your TRACE32 installation. To access the script, run this command:

```
B::CD.PSTEP ~~/demo/coverage/example.cmm
```

See also

- BookMark.EXPORT

BookMark.EXPORT.SOURCE   Export bookmarks for specified source files

Format:	**BookMark.EXPORT.SOURCE** *<xml_file> <source_file>*… [**/Append**]

Exports only those bookmarks to an XML file that have been created within the specified source files.

<source_file>	Apply one or more source files as filter criteria. The wildcards ‘*’ and ‘?’ are supported. Only bookmarks matching the filter criteria are exported.
Append	For a description and an example, see **BookMark.EXPORT**.

Example:

```
BookMark.EXPORT.SOURCE ~~/bookmarks-sources.xml \\*\".\src\sieve.c"
```

See also

- BookMark.EXPORT

BookMark.EXPORT.sYmbol   Export bookmarks for specified symbols

Format:	**BookMark.EXPORT.sYmbol** *<xml_file> <symbol>*… [**/Append**]

Exports only those bookmarks to an XML file that have been created for the specified symbols.

<symbol>	Apply one or more symbol names as filter criteria. The wildcards ‘*’ and ‘?’ are supported. Only bookmarks matching the filter criteria are exported.
Append	For a description and an example, see **BookMark.EXPORT**.

Example:

```
BookMark.EXPORT.sYmbol ~~/bookmarks-symbols.xml main *eve*
```

See also

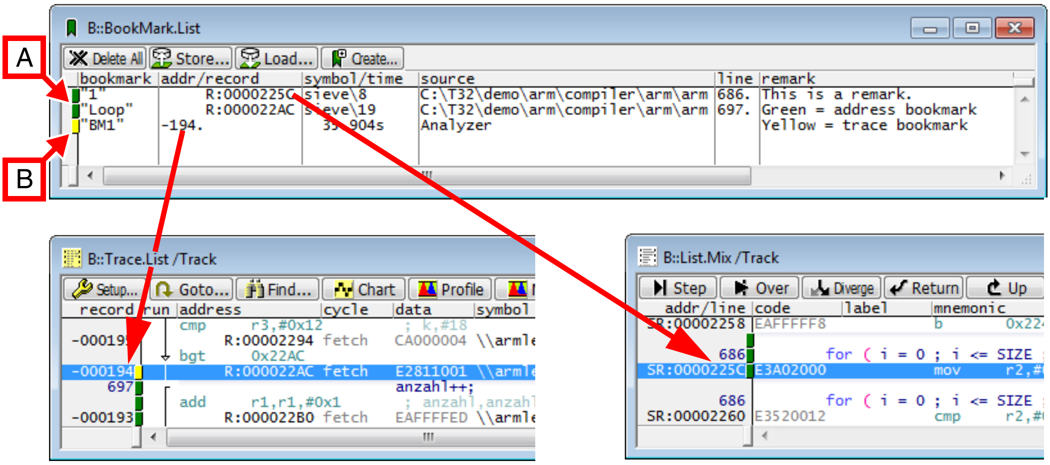
- BookMark.EXPORT

Format:	**BookMark.List**

Displays all existing bookmarks. There are two types of bookmarks, which are distinguished by their color:

- **Address bookmarks** are marked with a small green rectangle.
- **Trace bookmarks** are marked with a small yellow rectangle.

The same bookmark color codes are also used in other TRACE32 windows.



```
bookmark  addr/record   symbol/time  source                              line  remark
"1"          R:0000225C sieve\8      C:\T32\demo\arm\compiler\arm\arm   686.  This is a remark.
"Loop"       R:000022AC sieve\19     C:\T32\demo\arm\compiler\arm\arm   697.  Green = address bookmark
"BM1"     -194.           39.904s    Analyzer                                 Yellow = trace bookmark
```

```
record run address              cycle   data         symbol
           cmp      r3,#0x12            ; k,#18
-00019              R:00002294  fetch   CA000004     \\arml
           bgt      0x22AC
-000194             R:000022AC  fetch   E2811001     \\arml
   697                                  anzahl++;
           add      r1,r1,#0x1          ; anzahl,anzahl
-000193             R:000022B0  fetch   EAFFFFED     \\arml
```

```
  addr/line   code       label    mnemonic
SR:00002258   EAFFFFF8            b          0x224
        686                 for ( i = 0 ; i <= SIZE
SR:0000225C   E3A02000            mov        r2,#
        686                 for ( i = 0 ; i <= SIZE
SR:00002260   E3520012            cmp        r2,#
```

A Address bookmark.

B Trace bookmark.

Example:

```
BookMark.List                 ; display all bookmarks in a list
```

See also

■ BookMark ■ <trace>.BookMark ■ <trace>.BookMarkToggle ■ AutoSTOre
■ STOre

▲ 'Comment your Results' in 'Application Note for Trace-Based Code Coverage'

BookMark.RESet Resets all bookmarks

Format: **BookMark.RESet**

Resets the bookmarking system. Alternatively, click **Delete All** in the **BookMark.List** window.

Example:

```
BookMark.RESet     ; reset all the bookmarks in the bookmarking system
```

See also

- BookMark

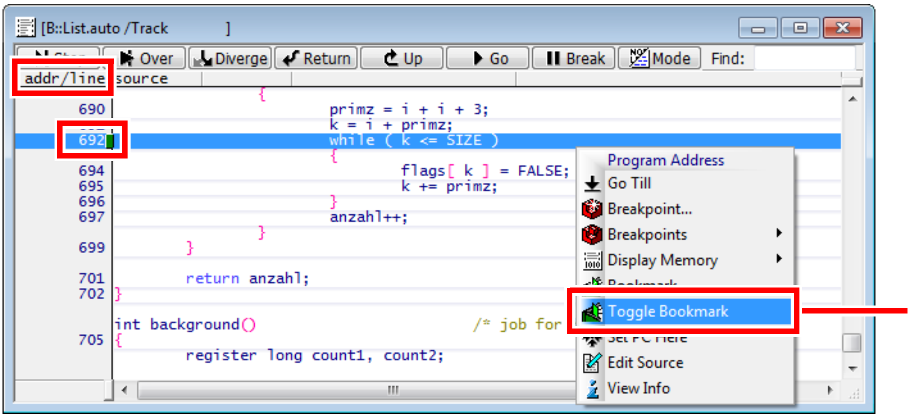
BookMark.Toggle

Toggles a single address bookmark

Format: **BookMark.Toggle** "*<bookmark_name>*" [*<address>* | *<time>*] [*<file>*] [*<line>*]

Switches a single address bookmark on or off. TRACE32 executes the same command when you right-click in a **List.auto** window, and then choose **Toggle Bookmark** (see figure below).

The resulting bookmark names are auto-incremented 1, 2, 3, etc. User-defined bookmark names can be created via the command line. A small green rectangle next to the address/line number indicates an address bookmark.



```
                    {
690                          primz = i + i + 3;
                             k = i + primz;
692                          while ( k <= SIZE )
                             {
694                                  flags[ k ] = FALSE;
695                                  k += primz;
696                          }
697                          anzahl++;
                    }
699          }
701          return anzahl;
702     }
        int background()                    /* job for
705     {
             register long count1, count2;
```

Address bookmark at line 692.

<bookmark_name> User-defined bookmark name. An auto-incremented bookmark name can be generated via the command line if a comma is entered instead of a user-defined name.

<time>, *<file>*, *<line>* The parameters *<time>*, *<file>*, and *<line>* are reserved for the scripting-mode of TRACE32, they are not needed in the dialog-mode of TRACE32.

Example:

```
List.auto /Track                ;display source listing
BookMark.List                   ;display all bookmarks in a list

BookMark.Toggle , 0x2290        ;switch on a bookmark at 0x2290 and
                                ;auto-increment the bookmark name

BookMark.Toggle "start" 0x1000  ;switch on a bookmark at 0x1000 and
                                ;label it "start"

BookMark.Toggle "start"         ;switch off the existing bookmark
```

See also

■ BookMark ■ <trace>.BookMarkToggle

Break — Stopping the program execution

The **Break** command group can be used in TRACE32 for

- Stopping the target program execution asynchronously using the command **Break.direct** or **Break.REQuest**
- Setting breakpoints
- Setting trace filters
- Programming complex triggers

See also

■ Go

▲ 'Breakpoints' in 'Training Basic Debugging'
▲ 'Breakpoints' in 'Training Basic SMP Debugging'

Breakpoints

A debugger has two methods to realize breakpoints: Software breakpoints and Onchip Breakpoints

A Software breakpoint replaces an instruction in the target memory by a special “breakpoint” instruction to stop the program and return control the debugger. The number of software breakpoints is unlimited. Breakpoints on instructions are called **Program** breakpoints by TRACE32 PowerView.

Onchip breakpoints use resources provided by the chip/core to realize a breakpoint. Onchip breakpoints are only available in a limited number. Refer to your **Processor Architecture Manual** for a detailed list of the available Onchip breakpoints. Onchip breakpoints can be set on instruction addresses (**Program** breakpoints) or can be used to stop the core at a read or write access to a memory location (**Read/Write** breakpoints).

Breakpoints can be set using the **Break.Set** command and controlled using the commands **Break.Delete**, **Break.ENable** and **Break.DISable** or from the **Break.List** window. Breakpoints set with **Break.Set** are permanent, i.e. they are not deleted when the program execution is stopped.

TRACE32 provides also so-called temporary breakpoints. Temporary breakpoints are only valid until the program execution stops the next time. They are automatically deleted by TRACE32. There are various commands that use temporary breakpoints. Just a few examples:

Break.direct *<address>* [*<breakpoint_type>*]	Set a temporary breakpoint to the specified *<address>* of the specified *<breakpoint_type>*.
Go.direct *<address>*	Set a temporary Program breakpoint to *<address>* and start the program execution.
Var.Go *<hll_expression>* [**Read** | **Write** | **ReadWrite**]	Set a temporary breakpoint to the specified *<hll_expression>* of the specified *<breakpoint_type>* and start the program execution.
Go.Return	Set a temporary Program breakpoint to the function epilog/exit and start the program execution.

The **Break.Set** command can also be used to set up trace filters as enabling or disabling the trace recording on a specific program address. These are also called in TRACE32 **Breakpoints**, which do not have however the default action “stop”.

The behavior of the different breakpoint types as well as their scope can be controlled with **Break.CONFIG** command group or from the **Break.CONFIG.state** window.

Further details and examples about the breakpoint usage are provided in **“Training Basic Debugging”** (training_debugger.pdf).

Break.Asm Stop program/set temporary breakpoint and switch to Asm mode

```
Format:          Break.Asm [<address> …[/<breaktype> …]]

<breaktype>:     Program | ReadWrite | Read | Write

                 Onchip | HARD | SOFT

                 ProgramPass | ProgramFail

                 MemoryReadWrite | MemoryRead | MemoryWrite
                 RegisterReadWrite | RegisterRead | RegisterWrite
                 VarReadWrite | VarRead | VarWrite
                 DATA[.Byte | .Word | .Long] <value> …

                 Alpha | Beta | Charly | Delta | Echo

                 WATCH | BusTrigger | BusCount
                 TraceEnable | TraceData | TraceON | TraceOFF | TraceTrigger

                 Spot
                 DISable | DISableHIT | DeleteHIT | NoMark | EXclude
                 TASK <task_magic> | <task_id> | <task_name>
                 MACHINE <machine_magic> | <machine_id> | <machine_name>
                 CORE <number>
                 COUNT <value>
                 CONDition <expression> [/AfterStep]
                 VarCONDition <hll_expression> [/AfterStep]
                 CMD <command_string>
                 RESUME

                 DIALOG | DIALOGADVANCED
```

Without an *<address>* parameter, this command stops the program execution and switches the debug mode to Asm.

With an *<address>* parameter, the command sets a temporary breakpoint at the given address. When the breakpoint is hit, TRACE32 PowerView switches the debug mode to Asm.

Refer to the description of the command **Mode.Asm** for more information about the different debug modes.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

See also

- Break.direct

Format:	**Break.CLEAR**

Resets complex triggers. This command does not reset breakpoints.

See also

- ■ Break.Delete
- ■ Break.direct
- ■ Break.Program
- ■ Break.ReProgram
- ■ Break.ViewProgram

▲ 'Introduction' in 'Application Note for Complex Trigger Language'

Break.CONFIG Configuration of breakpoint behavior and breakpoint scope

The **Break.CONFIG** command group allows the configuration of the behavior of the different breakpoint types as well as their scope.

See also

- Break.CONFIG.AlwaysAlive
- Break.CONFIG.InexactData
- Break.CONFIG.InexactTrigger
- Break.CONFIG.MatchMachine
- Break.CONFIG.METHOD
- Break.CONFIG.UseContextID
- Break.CONFIG.VarConvert
- Break.Set
- Break.CONFIG.InexactAddress
- Break.CONFIG.InexactResume
- Break.CONFIG.MatchASID
- Break.CONFIG.MatchZone
- Break.CONFIG.state
- Break.CONFIG.UseMachineID
- Break.direct

Break.CONFIG.AlwaysAlive Alive Onchip breakpoints

[build 142724 - DVD 02/2022]

Format:	**Break.CONFIG.AlwaysAlive** [**ON** | **OFF**]

Default: OFF

Allows to keep Onchip breakpoints alive in core when target is stopped.

See also

- Break.CONFIG

Break.CONFIG.InexactAddress Inexact address range breakpoint

Format:	**Break.CONFIG.InexactAddress** [**ON** | **OFF**] **TrOnchip.CONVert** [**ON** | **OFF**] (deprecated)

Default: ON

Allows to specify how TRACE32 behaves if an Onchip breakpoint is set to an address range, but the breakpoint logic of the core in use does not provide the appropriate resources **(see note below)**.

ON	TRACE32 will automatically adjust the address range to fit in the breakpoint logic. This may cause the core to stop outside the desired range. **Please note that the Break.List window still display the original address range, but the breakpoint is marked as intrusive breakpoint.** Please refer to **"Real-time vs. Intrusive Breakpoints"**, page 87 for more information.
OFF	If the breakpoint logic can not implement the address range exact, the error message "address does not fit in on-chip breakpoint resources" is returned.

Break.CONFIG.InexactAddress can be used in conjunction with **Break.CONFIG.InexactResume**. If this command is set to ON, TRACE32 will automatically resume the program execution if it detects that the stop is due to an access outside the original address range set by the user. Please note however, that in some cases, it is not possible to determine the exact address that caused the breakpoint to fire.

When stopping on an inexact breakpoint, the TRACE32 state line displays the message "**stopped at inexact breakpoint**".

NOTE:	The breakpoint logic of the core usually allows to set Onchip breakpoints for single addresses. Breakpoints for exact address ranges are however not supported by many core architectures. Some core architectures allow only single addresses (e.g. PPC740), others only fixed ranges (e.g. Intel® x86/x64 allows ranges of 2, 4 or 8 bytes) and many cores implement ranges as bit masks.

See also

■ Break.CONFIG

▲ 'Release Information' in 'Legacy Release History'

Break.CONFIG.InexactData — Inexact data value breakpoint

Format: **Break.CONFIG.InexactData** [**ON** | **OFF**]

Default: ON

The breakpoint logic of some processor architectures allows to set data value breakpoints i.e. to stop the program execution when a specific data value is written or read to/from an address. The command **Break.CONFIG.InexactData** can be used to specify how TRACE32 behaves when data value breakpoints are **not supported** by the breakpoint logic of the core.

ON	TRACE32 sets an Onchip breakpoint without data value and checks on each breakpoint hit the value which is read/written from/to the breakpoint address. The breakpoint is marked as intrusive in the **Break.List** window. Please refer to **"Real-time vs. Intrusive Breakpoints"**, page 87 for more information.
OFF	If the breakpoint logic can not implement data value Onchip breakpoints, the error message "data does not fit in on-chip breakpoint resources" is returned.

Break.CONFIG.InexactData can be used in conjunction with **Break.CONFIG.InexactResume**. If this command is set to ON, TRACE32 will automatically resume the program execution if the data value written/read to/from the breakpoint address is different from the one selected by the user.

See also

- Break.CONFIG

Break.CONFIG.InexactResume — Resuming on inexact breakpoints

```
Format:          Break.CONFIG.InexactResume [ON | OFF]
```

Default: ON

Defines how TRACE32 behaves when the execution is stopped on an inexact breakpoint. Please refer to **Break.CONFIG.InexactAddress**, **Break.CONFIG.InexactData** and **Break.CONFIG.InexactTrigger** for more information.

See also

- Break.CONFIG

Break.CONFIG.InexactTrigger — Inexact trigger breakpoints

```
Format:          Break.CONFIG.InexactTrigger [ON | OFF]
```

Default: OFF

Enables/disables inexact breakpoints for **TraceON**, **TraceOFF**, **TraceTrigger**, **BusTrigger** and **BusCount** breakpoints. Please refer to the documentation of the **Break.Set** command for more information about the different breakpoint types.

Setting **Break.CONFIG.InexactTrigger** to **ON** will automatically set **Break.CONFIG.InexactAddress ON**.

See also

- Break.CONFIG

Break.CONFIG.MatchASID — Use ASID specific breakpoints

Format: **Break.CONFIG.MatchASID** [**ON** | **OFF**]
TrOnchip.MatchASID [**ON** | **OFF**] (deprecated)
TrOnchip.ASID [ON | OFF] (deprecated)

Default: OFF

When this command is set to ON, Onchip breakpoints will be set specific to the ASID (Address Space IDentifier) relative to the used task space ID or the space ID of the current task (if supported by the target processor). Space IDs are enabled in TRACE32 with the command **SYStem.Option.MMUSPACES ON**. OS-aware debugging has additionally to be enabled in TRACE32 in order to set ASID specific breakpoints.

Example:

```
Break.CONFIG.MatchASID ON

; set an Onchip breakpoint specific to the ASID of the process with
; space ID 0x159
Break.Set 0x159:0x97D0 /Onchip

; set an Onchip breakpoint specific to the ASID of the current process
Break.Set 0x97D0 /Onchip
```

The Onchip breakpoint will only trigger if the ASID used for the breakpoint is the current one. If the ASID is not available for the target processor, **MatchASID** will be greyed out in the **Break.CONFIG.state** window and the command will be locked.

See also

- Break.CONFIG
- ▲ ’Release Information’ in ’Legacy Release History’

Break.CONFIG.MatchMachine Use machine specific breakpoints

Format: **Break.CONFIG.MatchMachine** [**ON** | **OFF**]
TrOnchip.MatchMachine [**ON** | **OFF**] (deprecated)

Default: OFF

When this command is set to ON, Onchip breakpoints will be set specific to the specified machine ID or the current machine ID if no machine is specified. The Onchip breakpoint will only trigger if the machine used for the breakpoint is the current one. Machine IDs are enabled in TRACE32 with the command **SYStem.Option.MACHINESPACES ON**. Hypervisor-aware debugging has additionally to be configured in order to set machine specific breakpoints.

Example:

```
Break.CONFIG.MatchMachine ON

; Trace only machine 2 on a 64-bit architecture
Break.Set 2:::0x0:0x0--0xffffffffffffffff /TraceEnable
```

See also

■ Break.CONFIG

▲ 'Release Information' in 'Legacy Release History'

Break.CONFIG.MatchZone Use zone specific breakpoints

Format: **Break.CONFIG.MatchZone** [**ON** | **OFF**]
TrOnchip.MatchZone [**ON** | **OFF**] (deprecated)

Default: OFF

When this command is set to ON, Onchip breakpoint are set specific to the given zone or the current zone. Zones are enabled in TRACE32 with the command **SYStem.Option.ZoneSPACES ON**.

Example:

```
Break.CONFIG.MatchZone ON

; Set an Onchip breakpoint on address 0x1000 for the Arm secure zone
Break.Set Z:0x1000 /Onchip
```

See also

■ Break.CONFIG

▲ 'Release Information' in 'Legacy Release History'

Format:	**Break.CONFIG.METHOD** [*<breaktype>* *<impl>*] **Break.METHOD** [*<breaktype>* *<impl>*] (deprecated) **Break.IMPLementation** [*<breaktype>* *<impl>*] (deprecated) **Break.SELect** (deprecated)
<breaktype>:	**Program** **Read** **Write** **Alpha** **Beta** **Charly** **Delta** **Echo**
<impl>:	**AUTO** **Onchip** **SOFT**

Defines the default implementation of breakpoints. Without any parameters, the command opens the **Break.CONFIG.state** window.

AUTO	Leave it to the debugger to use the appropriate breakpoint implementation.
SOFT	Advise TRACE32 to implement this breakpoint type as SOFTware breakpoint.
Onchip	Advise TRACE32 to implement this breakpoint type as Onchip

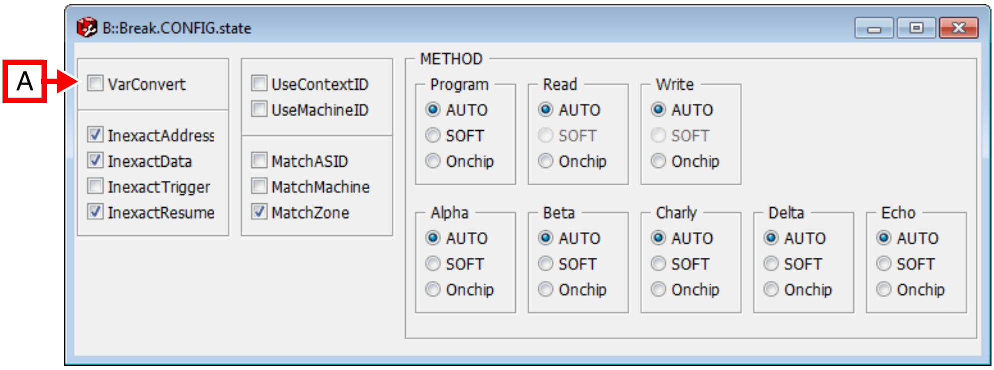
See also

- Break.CONFIG

Break.CONFIG.state — Breakpoint configuration window

Format: **Break.CONFIG.state**

Opens the breakpoint configuration window.



B::Break.CONFIG.state

□ VarConvert

☑ InexactAddress
☑ InexactData
□ InexactTrigger
☑ InexactResume

□ UseContextID
□ UseMachineID

□ MatchASID
□ MatchMachine
☑ MatchZone

METHOD

Program	Read	Write
● AUTO	● AUTO	● AUTO
○ SOFT	○ SOFT	○ SOFT
○ Onchip	○ Onchip	○ Onchip

Alpha	Beta	Charly	Delta	Echo
● AUTO	● AUTO	● AUTO	● AUTO	● AUTO
○ SOFT	○ SOFT	○ SOFT	○ SOFT	○ SOFT
○ Onchip	○ Onchip	○ Onchip	○ Onchip	○ Onchip

A For descriptions of the commands in the **Break.CONFIG.state** window, please refer to the **Break.CONFIG.*** commands in this chapter.
Example: For information about **VarConvert**, see **Break.CONFIG.VarConvert**.

See also

- Break.CONFIG

Break.CONFIG.UseContextID — Context ID specific breakpoints

Format: **Break.CONFIG.UseContextID** [**ON** | **OFF**]
TrOnchip.ContextID [**ON** | **OFF**] (deprecated)

Default: OFF

Enables/disables the usage of the ContextID comparator, if supported by the target processor architecture, for task selective Onchip breakpoints. Please note the CONTEXTIDR register has additionally to be written by the kernel on every task switch.

ON	Task-selective Onchip breakpoints will be implemented using the ContextID comparator. The breakpoint is in this case non-intrusive i.e. the execution will stop on the breakpoint only if the selected task is the current one.
OFF	Task-selective breakpoints will be implemented as intrusive breakpoints i.e. the program execution will always stop on the breakpoint. The execution will be automatically resumed by the debugger if the selected task for the breakpoint is not the current one.

If the ContextID comparator is not available for the target processor architecture, **UseContextID** will be greyed out in the **Break.CONFIG.state** window and the command will be locked.

See also

■ Break.CONFIG

▲ 'Release Information' in 'Legacy Release History'

Break.CONFIG.UseMachineID   Machine ID specific breakpoints

Format: **Break.CONFIG.UseMachineID** [**ON** | **OFF**]
TrOnchip.MachineID [**ON** | **OFF**] (deprecated)

Default: OFF

Enables/disables the usage of the VMID comparator to set machine specific breakpoints, if supported by the target processor architecture. Please note the VMID has additionally to be written by the kernel on every machine switch.

ON Machine-selective Onchip breakpoints will be implemented using the VMID comparator. The breakpoint is in this case non-intrusive i.e. the execution will stop on the breakpoint only if the selected machine is the current one.

OFF Machine-selective breakpoints will be implemented as intrusive breakpoints i.e. the program execution will always stop on the breakpoint. The execution will be automatically resumed by the debugger if the selected machine for the breakpoint is not the current one.

See also

■ Break.CONFIG

▲ ’Release Information’ in ’Legacy Release History’

Format:	**Break.CONFIG.VarConvert** [**ON** | **OFF**] TrOnchip.VarCONVert [ON | OFF] (deprecated)

Default: OFF

Defines the debugger behavior when setting a breakpoint to a scalar variable (int, float, double).

ON	The breakpoint is set to the start address of the variable. This setting consumes the least amount of core breakpoint resources.
OFF	The breakpoint is set to all the memory address range that holds the variable value. This setting requires more core breakpoint resources, but also triggers on partial accesses to the variable (e.g. only one byte of the 32 bit variable). Use this setting when searching for a variable being partially overwritten (e.g. by an out-of bounds access to an array located nearby).

See also

■ Break.CONFIG

▲ 'Release Information' in 'Legacy Release History'

Break.Delete   Delete breakpoints

Format:	**Break.Delete** [[*<address>* | *<addressrange>*] [**/***<breaktype>* …]]
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write** **Onchip** | **HARD** | **SOFT** **ProgramPass** | **ProgramFail** **Alpha** | **Beta** | **Charly** | **Delta** | **Echo** **WATCH** | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **Spot**

Deletes all breakpoints if used without a parameter.

<address>, *<addressrange>*	Specifying an *<address>* or an *<addressrange>* allows to delete only the specified breakpoint.
<breaktype>	Specifying a *<breaktype>* allow to delete all breakpoints of this type. For a description of the breakpoint types and breakpoint options, see **Break.Set**.

Examples:

```
Break.Delete                     ; delete all breakpoints

Break.Delete 0x1000--0x1fff      ; delete all breakpoints
                                 ; in the address range of 0x1000 to 0x1fff

Break.Delete func9               ; delete the breakpoint at the entry
                                 ; to the function func9

Break.Delete mstatic1 /Read      ; delete read breakpoints on integer
                                 ; variable mstatic1

; delete write breakpoint on array flags
Var.Break.Delete flags /Write
```

See also

- ■ Break.CLEAR ■ Break.direct ■ Var.Break.Delete

▲ 'Breakpoint Handling' in 'Training Basic Debugging'
▲ 'Breakpoint Handling' in 'Training Basic SMP Debugging'

Break.DeletePATtern — Delete breakpoints allowing wildcards

Format:	**Break.DeletePATtern** *<symbol_pattern>* [**/***<type>*]

Delete breakpoints allowing the wildcards ? and *. For details on deleting breakpoints, refer to the **Break.Delete** command.

<type>	Specifying a *<type>* allow to delete all breakpoints of this type. For a description of the breakpoint types and breakpoint options, see **Break.Set**.

Example:

```
Break.DeletePATtern *memory* /Program      ; delete program breakpoints
                                           ; from all debug symbols that
                                           ; contain the string "memory".
```

See also

- ■ Break.direct
- ▲ 'Release Information' in 'Legacy Release History'

Format:	**Break.direct** [*<address>* …[**/***<breaktype>* …]]
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write** **Onchip** | **HARD** | **SOFT** **ProgramPass** | **ProgramFail** **MemoryReadWrite** | **MemoryRead** | **MemoryWrite** **RegisterReadWrite** | **RegisterRead** | **RegisterWrite** **VarReadWrite** | **VarRead** | **VarWrite** **DATA**[**.Byte** | **.Word** | **.Long**] *<value>* … **Alpha** | **Beta** | **Charly** | **Delta** | **Echo** **WATCH** | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger** **Spot** **DISable** | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **CORE** *<number>* **COUNT** *<value>* **CONDition** *<expression>* [**/AfterStep**] **VarCONDition** *<hll_expression>* [**/AfterStep**] **CMD** *<command_string>* **RESUME** **DIALOG** | **DIALOGANVANCED**

Break.direct stops the program execution, if no address parameter is specified

If address parameters are provided, **Break.direct** sets so-called *temporary* breakpoints at the specified addresses. A temporary breakpoint is valid until the program stops the next time. Once the program stops, all temporary breakpoints are deleted by the debugger. One application is to set temporary breakpoints on multiple alternative execution paths, if it is not known which one will be taken.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

NOTE:	Please note that **break** and **b** are abbreviations of the **Break.direct** command and not of **Break.Set**. Also note the convention used in TRACE32 manuals to spell commands with all mandatory letters capitalized.

Examples:

```
Go                             ; start program execution
Break                          ; stop program execution

Break 0x1000                   ; set a temporary Program breakpoint at
                               ; address 0x1000
Go                             ; start the program execution

Break main /Program            ; set a temporary breakpoint of the type
                               ; Program to the entry of the function
                               ; main

Break \main\100                ; set a temporary breakpoint to line 100
                               ; of module "main"

Break func1 func9              ; set temporary breakpoints to the entries
                               ; of the functions func1 and func9
Go                             ; start the program execution

Go func1 func9                 ; or identical


Var.Break ast /Read            ; set a temporary Read breakpoint to
                               ; the variable ast
```

See also

- Break.Asm
- Break.CLEAR
- Break.CONFIG
- Break.Delete
- Break.DeletePATtern
- Break.DISable
- Break.ENable
- Break.Hll
- Break.Init
- Break.List
- Break.Mix
- Break.MONitor
- Break.PASS
- Break.PATtern
- Break.Program
- Break.ReProgram
- Break.REQuest
- Break.RESet
- Break.Set
- Break.SetFunc
- Break.SetLine
- Break.SetMONitor
- Break.SetPATtern
- Break.SetTask
- Break.ViewProgram
- Go.direct
- Var.Break.direct

Break.DISable — Disable breakpoints

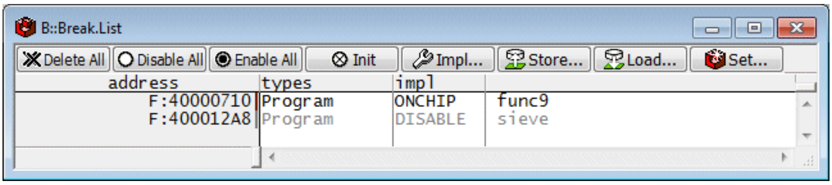
Format:	**Break.DISable** [[*<address>* | *<addressrange>*] [**/***<breaktype>* …]]
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write** **Onchip** | **HARD** | **SOFT** **ProgramPass** | **ProgramFail** **Alpha** | **Beta** | **Charly** | **Delta** | **Echo** **WATCH** | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **CORE** *<number>*

Disables a breakpoint. The breakpoint remains set but is not active.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

Examples:

```
Break.DISable                 ; disable all breakpoints

Break.DISable sieve           ; disable the breakpoint at address sieve
```



```
       address  types      impl
    F:40000710  Program    ONCHIP    func9
    F:400012A8  Program    DISABLE   sieve
```

See also

- ■ Break.direct

▲ 'Breakpoint Handling' in 'Training Basic Debugging'
▲ 'Breakpoint Handling' in 'Training Basic SMP Debugging'

Format: **Break.ENable** [[*<address>* | *<addressrange>*] [**/***<breaktype>* …]]

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**
Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

TASK *<task_magic>* | *<task_id>* | *<task_name>*

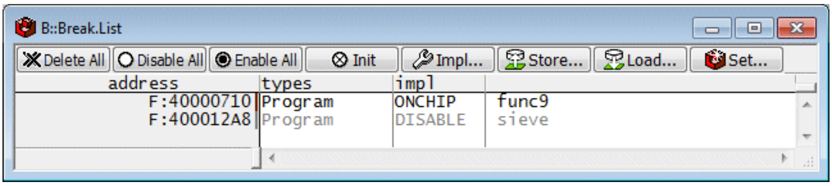
CORE *<number>*

Enables a breakpoint. The breakpoint becomes active again.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

Examples:

```
Break.DISable sieve          ; disable the breakpoint at address sieve
```



```
      address   types     impl
   F:40000710   Program   ONCHIP    func9
   F:400012A8   Program   DISABLE   sieve
```

```
Break.ENable sieve           ; enable the breakpoint at address sieve
```

See also

■ Break.direct

▲ 'Breakpoint Handling' in 'Training Basic Debugging'
▲ 'Breakpoint Handling' in 'Training Basic SMP Debugging'

Break.Hll Stop program/set temporary breakpoint and switch to HLL mode

Format:	**Break.Hll** [*<address>* …[**/***<breaktype>* …]]
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write** **Onchip** | **HARD** | **SOFT** **ProgramPass** | **ProgramFail** **MemoryReadWrite** | **MemoryRead** | **MemoryWrite** **RegisterReadWrite** | **RegisterRead** | **RegisterWrite** **VarReadWrite** | **VarRead** | **VarWrite** **DATA**[**.Byte** | **.Word** | **.Long**] *<value>* … **Alpha** | **Beta** | **Charly** | **Delta** | **Echo** **WATCH** | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger** **Spot** **DISable** | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **CORE** *<number>* **COUNT** *<value>* **CONDition** *<expression>* [**/AfterStep**] **VarCONDition** *<hll_expression>* [**/AfterStep**] **CMD** *<command_string>* **RESUME** **DIALOG | DIALOGADVANCED**

Stops the program execution or sets a temporary breakpoint and switches the debug mode to Hll. Please refer to the description of the **Mode.Hll** command for more information.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

See also

■ Break.direct

Break.Init   Initialize breakpoints

Format:	**Break.Init**

Break.Init deletes all temporary breakpoints, sets all permanent breakpoint again and resets the breakpoint counters.

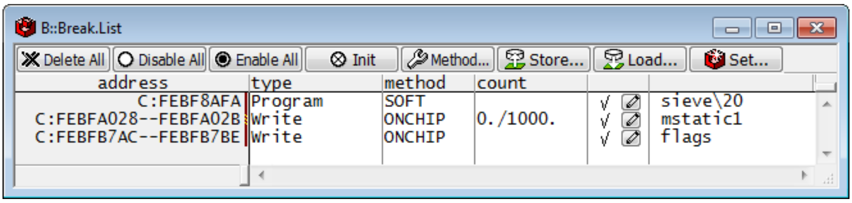
See also

- ■ Break.direct
- ▲ 'Release Information' in 'Legacy Release History'

Break.List   Display list of breakpoints

Format:	**Break.List** [**/**<*option*>]
<*option*>:	**Onchip** | **CTL** | **Summary** | **HARD**

Displays a list of all breakpoints.



```
             address type     method  count
          C:FEBF8AFA Program  SOFT                 √   sieve\20
C:FEBFA028--FEBFA02B Write    ONCHIP  0./1000.     √   mstatic1
C:FEBFB7AC--FEBFB7BE Write    ONCHIP               √   flags
```

The following options are mainly used for diagnosis:

Onchip	Display details on Onchip breakpoints.
OnchipDetail	Display details about the usage of the available address comparators for the individual Onchip breakpoints. [build 116363 - DVD 02/2020]
CTL	Display details on CTL breakpoints.
Summary Physical (deprecated)	Summarizes the details about all breakpoints.
HARD	Display details on HARDware breakpoints.

See also

- ■ Break.direct
- ▲ 'Release Information' in 'Legacy Release History'

- ▲ ’Breakpoint Handling’ in ’Training Basic Debugging’
- ▲ ’Breakpoint Handling’ in ’Training Basic SMP Debugging’

Break.Mix Stop program/set temporary breakpoint and switch to MIX mode

Format:	**Break.Mix** [*<address>* …[**/***<breaktype>* …]]
<breaktype>:	**Program** | **ReadWrite** | **Read** | **Write** **Onchip** | **HARD** | **SOFT** **ProgramPass** | **ProgramFail** **MemoryReadWrite** | **MemoryRead** | **MemoryWrite** **RegisterReadWrite** | **RegisterRead** | **RegisterWrite** **VarReadWrite** | **VarRead** | **VarWrite** **DATA**[**.Byte** | **.Word** | **.Long**] *<value>* … **Alpha** | **Beta** | **Charly** | **Delta** | **Echo** **WATCH** | **BusTrigger** | **BusCount** **TraceEnable** | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger** **Spot** **DISable** | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude** **TASK** *<task_magic>* | *<task_id>* | *<task_name>* **MACHINE** *<machine_magic>* | *<machine_id>* | *<machine_name>* **CORE** *<number>* **COUNT** *<value>* **CONDition** *<expression>* [**/AfterStep**] **VarCONDition** *<hll_expression>* [**/AfterStep**] **CMD** *<command_string>* **RESUME** **DIALOG** | **DIALOGADVANCED**

Stops program execution or sets a temporary breakpoint and switches the debug mode to Mix. Refer to **Mode.Mix** for more information,

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.

See also

- Break.direct

Break.MONitor   Switch back to stop mode debugging

Format: **Break.MONitor**

This command is used in Run Mode debugging to switch back to Stop Mode and stop the program execution.

The command **Go.MONitor** is used to switch from Stop Mode to Run Mode debugging.

See also

- Break.direct
- Break.SetMONitor
- Go.MONitor

Break.PASS   Define pass condition for breakpoint

Format: **Break.PASS** [*<boolean_expression>*]

When the program execution is stopped by a breakpoint, and the boolean expression is true, the program execution is automatically restarted. The feature can be cleared by entering the command without arguments.

Examples:

```
Break.PASS Register(A7)>0x1000    ; automatically restart the program
                                  ; execution at a breakpoint hit, if
                                  ; the register A7 is larger than
                                  ; 0x1000
Break.Set 0x100                   ; set a breakpoint
Break.Set sieve+34                ; set a second breakpoint
Go                                ; start the program execution
…
Break.PASS                        ; remove the pass condition
```

The following commands shows how a condition can be directly assigned to a single breakpoint.

```
Break.Set sieve+34 /Program /CONDition Register(R9)==0

Go

Break.Delete sieve+34
```

See also

- Break.direct
- Break.ReProgram
- Break.ViewProgram

Break.PATtern — Set temporary breakpoints allowing wildcards

Format: **Break.PATtern** *<symbol_pattern>* [**/***<type>*]

Sets a temporary breakpoint allowing the wildcards ? and *. For details on temporary breakpoints, refer to the **Break.direct** command.

Example:

```
Break.PATtern *memory* /Program    ; set temporary program breakpoints to
                                   ; all debug symbols that contain the
                                   ; string "memory".
```

See also

■ Break.direct

▲ 'Release Information' in 'Legacy Release History'

Break.Program — CTL interactive programming

Format: **Break.Program** [*<file>*]

Opens the **Break.Program** editor window, where you can create Complex Trigger Language (CTL) scripts. The editor provides syntax highlighting, configurable auto-indentation and an online syntax check. The input is guided by softkeys.

See also

■ Break.CLEAR ■ Break.direct

▲ 'Introduction' in 'Application Note for Complex Trigger Language'

Break.ReProgram   Activate existing CTL program file

Format:	**Break.ReProgram** [<*file*>]

Activates an existing Complex Trigger Language (CTL) file.

See also

■ Break.CLEAR ■ Break.direct ■ Break.PASS

▲ 'Introduction' in 'Application Note for Complex Trigger Language'

Break.REQuest   Request a program break

Format:	**Break.REQuest**

This command requests a program break but does not wait until the program execution is stopped.

See also

■ Break.direct

Break.RESet   Delete all breakpoints and reset the TRACE32 break system

Format:	**Break.RESet**

Deletes all breakpoints and resets the TRACE32 break system.

See also

■ Break.direct

Break.Set Set breakpoints

[Breakpoint Types] [Breakpoint Options]

Format: **Break.Set** [*<address>*|*<range>*] [**/***<breaktype>* …]] [**/***<impl>*]

<impl>: **SOFT** | **Onchip**

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME
DIALOG | **DIALOGADVANCED**

The **Break.Set** command sets breakpoints via the TRACE32 command line. Without parameters, the command opens the **Break.Set** dialog window for setting breakpoints.

NOTE: You can configure the breakpoint behavior with the **Break.CONFIG** command group.

A detailed introduction into the breakpoint usage can be found in **"Training Basic Debugging"** (training_debugger.pdf).

NOTE:	Do not erroneously abbreviate the command **Break.Set** *<address>* as **Break** *<address>* The command **Break.Set** *<address>* sets a permanent breakpoint, whereas the command **Break** *<address>* sets a breakpoint that is automatically deleted when the program execution is stopped the next time (temporary breakpoint).

The following breakpoint implementations are available:

SOFT	The code at the breakpoint location is patched with a break instruction. A software breakpoint usually requires RAM at the breakpoint location. If you want to set software breakpoints to instructions in FLASH refer to command **FLASH.Auto**.
Onchip	The resources for the breakpoints are provided by the chip.

[Back to Top]

Refer to your **Processor Architecture Manual** for a detailed list of the available Onchip breakpoints.

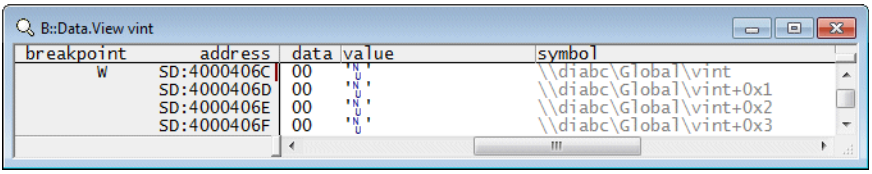
For some processor architectures Onchip breakpoints can only mark **single addresses** (e.g Cortex-A9). Most processor architectures, however, allow to mark **address ranges** with Onchip breakpoints. It is very common that one Onchip breakpoint marks the start address of the address range while the second Onchip breakpoint marks the end address (e.g. MPC57xx).

The command **Break.CONFIG.VarConvert** (TrOnchip.VarConvert in older software versions) allows to control how range breakpoints are set for scalars (int, float, double).

Break.CONFIG.VarConvert ON	If a breakpoint is set to a scalar variable (int, float, double) the breakpoint is set to the start address of the variable. + Requires only one single address breakpoint. - Program will not stop on unintentional accesses to the variable’s address space.
Break.CONFIG.VarConvert OFF	If a breakpoint is set to a scalar variable (int, float, double) breakpoints are set to all memory addresses that store the variable value. + The program execution stops also on any unintentional accesses to the variable’s address space. - Requires two onchip breakpoints since a range breakpoint is used.

The current setting can be inspected and changed from the **Break.CONFIG** window.

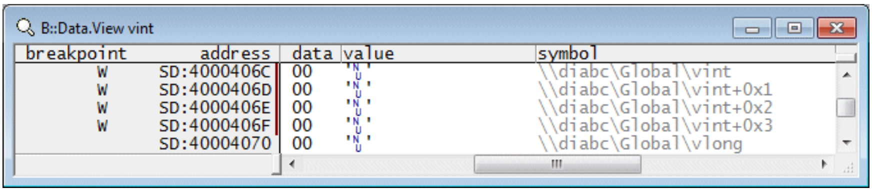
Example: the red line in the **Data.View** window shows the range of the Onchip breakpoint.



```
breakpoint      address  data value          symbol
         W  SD:4000406C  00   'N U'          \\diabc\Global\vint
            SD:4000406D  00   'N U'          \\diabc\Global\vint+0x1
            SD:4000406E  00   'N U'          \\diabc\Global\vint+0x2
            SD:4000406F  00   'N U'          \\diabc\Global\vint+0x3
```

```
; Set an Onchip breakpoint to the start address of the variable vint
Break.CONFIG.VarConvert ON
Var.Break.Set vint /Write
Data.View vint
```

```
; Set an Onchip breakpoint to the whole memory range address of the
; variable vint
Break.CONFIG.VarConvert OFF
Var.Break.Set vint /Write
Data.View vin
```



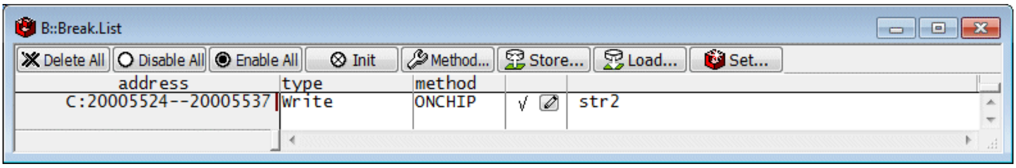
```
breakpoint      address   data value      symbol
         W  SD:4000406C   00   '␀'        \\diabc\Global\vint
         W  SD:4000406D   00   '␀'        \\diabc\Global\vint+0x1
         W  SD:4000406E   00   '␀'        \\diabc\Global\vint+0x2
         W  SD:4000406F   00   '␀'        \\diabc\Global\vint+0x3
            SD:40004070   00   '␀'        \\diabc\Global\vlong
```

A number of processor architectures provide only **bit masks** or **fixed range sizes** to mark an address range with Onchip breakpoints. In this case the address range is always enlarged to the **smallest bit mask/next allowed range** that includes the address range.

It is recommended to control which addresses are actually marked with breakpoints by using the **Break.List /Onchip** command:

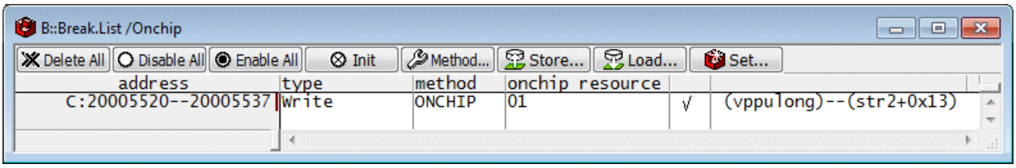
Breakpoint setting:

```
Var.Break.Set str2

Break.List
```



```
             address  type      method
C:20005524--20005537  Write     ONCHIP   √    str2
```

```
Break.List /Onchip
```



```
             address  type      method   onchip resource
C:20005520--20005537  Write     ONCHIP   01                √   (vppulong)--(str2+0x13)
```

Breakpoint Types

[Back to Top]

The following breakpoint types are available:

Program	The program execution is stopped before the instruction marked with the breakpoint is executed (for most processor architectures). Default implementation for a Program breakpoint is SOFT for nearly all processor architectures.
ReadWrite	The program executions is stopped at a read or write access to the specified address. Default implementation for a read/write breakpoint is Onchip.
Read	The program executions is stopped at a read access to the specified address. Default implementation for a read breakpoint is Onchip.
Write	The program executions is stopped at a write access to the specified address. Default implementation for a write breakpoint is Onchip.

There are two flavours of breakpoints:

- **Break after make**

  The program execution is stopped after the read/write access was performed respectively after the instruction marked with the breakpoint was executed.

- **Break before make**

  The program execution is stopped before the instruction marked with the breakpoint was executed respectively before the read/write access was performed.

Real-time vs. Intrusive Breakpoints

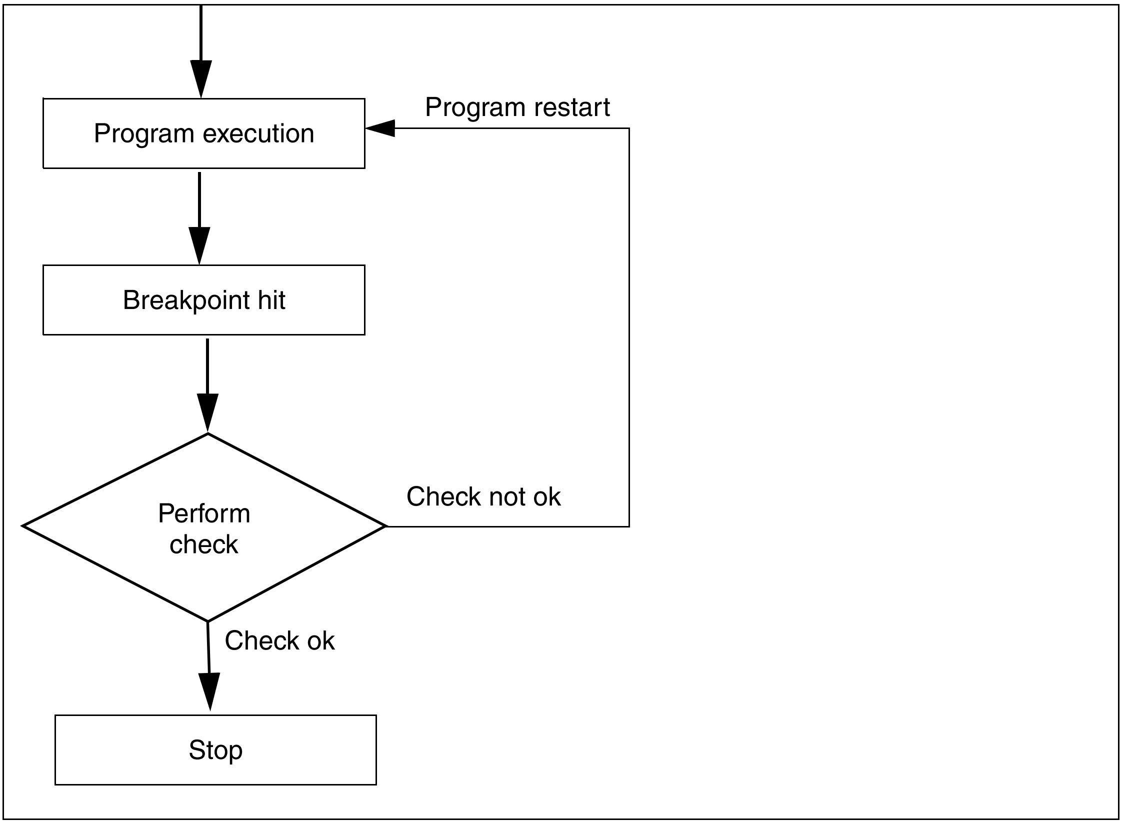
[Back to Top]

Real-time breakpoints

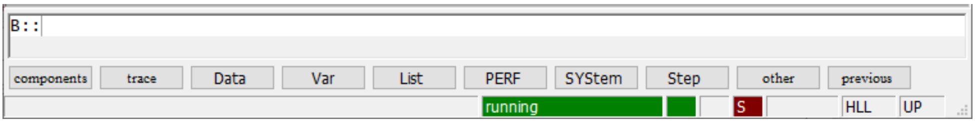
The usage of a breakpoint does not influence the real-time behavior of the application program.

Intrusive breakpoints

The usage of the breakpoint influences the real-time behavior. Intrusive breakpoints perform as follows:



Each stop to perform the check suspends the program execution for at least 1 ms.



The (short-time) display of a red S in the state line indicates that an intrusive breakpoint was hit.

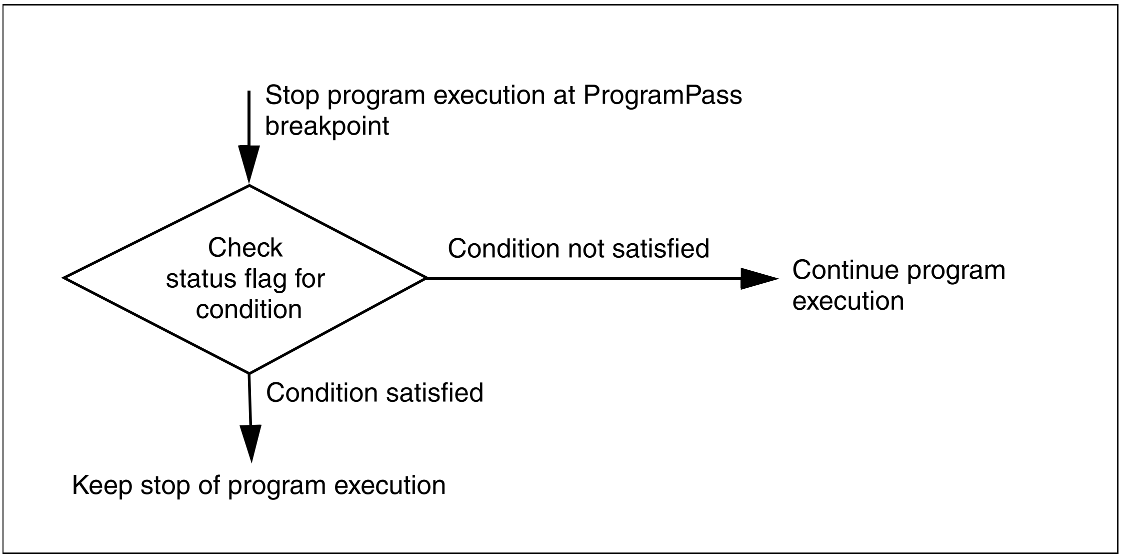
TRACE32 implements real-time breakpoints whenever possible.

Intrusive breakpoints are marked with a special breakpoint indicator:

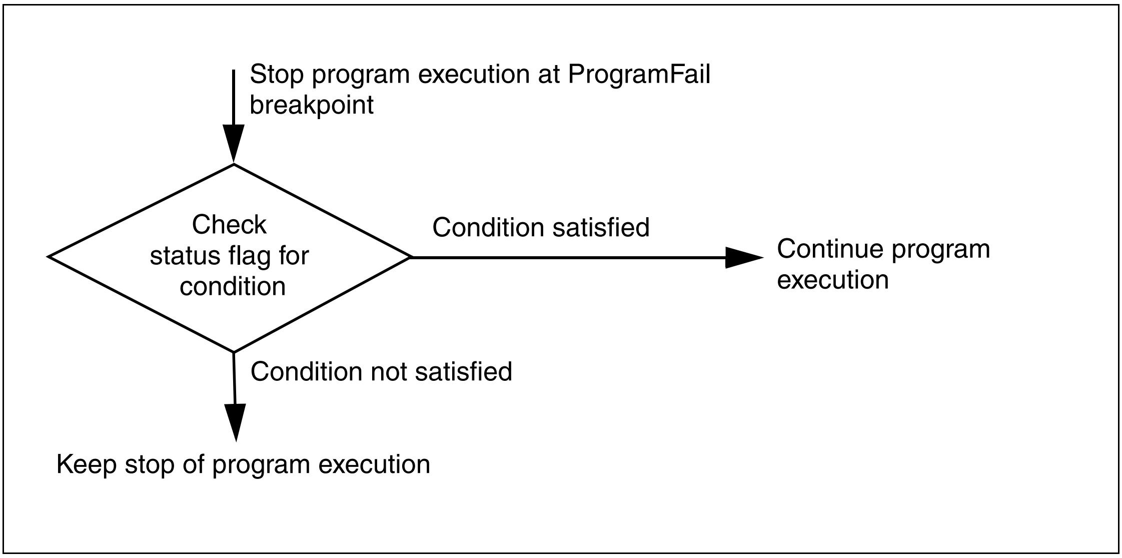
[Back to Top]

If an instruction is conditionally executed (e.g. BGT - Branch Greater Then, LDREQB - Load Byte if Equal), TRACE32 stops shortly to check the status flags in order to find out if the condition is satisfied.

- **ProgramPass (intrusive breakpoint)**


Stop program execution at ProgramPass breakpoint
Check status flag for condition
Condition not satisfied
Continue program execution
Condition satisfied
Keep stop of program execution

- **ProgramFail (intrusive breakpoint)**


Stop program execution at ProgramFail breakpoint
Check status flag for condition
Condition satisfied
Continue program execution
Condition not satisfied
Keep stop of program execution

NOTE: The following options are **not** available for all processor architectures!

The following options can be used, if the on-chip debug unit of your processor makes it possible to stop the program execution when a read or write access to an address is performed by a specific code section. If this feature is not supported by your processor, these options are deactivated.

MemoryReadWrite	Set a MemoryReadWrite breakpoint.
MemoryRead	Set a MemoryRead breakpoint.
MemoryWrite	Set a MemoryWrite breakpoint.
VarReadWrite	Set a MemoryReadWrite breakpoint to a static variable.
VarRead	Set a MemoryRead breakpoint to a static variable.
VarWrite	Set a MemoryWrite breakpoint to a static variable.

Examples:

```
; Stop the program execution when an instruction of the code range
; 0xA100--0xA32D writes to the address 0x400
Break.Set 0xA100--0xA32D /MemoryWrite 0x400
```

```
; Stop the program execution when an instruction of the function sieve
; writes to the variable flags
Var.Break.Set sieve /VarWrite flags
```

The following options can be used, if the on-chip debug unit of you processor makes it possible to stop the program execution when a read or write access to a core register is performed by a specific code section. If this feature is not supported by your processor, these options are deactivated.

RegisterReadWrite	Set a breakpoint which stops the cpu on a core register access.
RegisterRead	Set a breakpoint which stops the cpu on a core register read.
RegisterWrite	Set a breakpoint which stops the cpu on a core register write.
VarReadWrite	Set a RegisterReadWrite breakpoint to a register variable.
VarRead	Set a RegisterRead breakpoint to a register variable.
VarWrite	Set a RegisterWrite breakpoint to a register variable.

Examples:

```
; Stop the program execution when an instruction of the code range
; 0xA100--0xA32D writes to register R1

Break.Set 0xA100--0xA32D /RegisterWrite R1
```

```
; Stop the program execution when an instruction of the function sieve
; writes to the register variable i

Var.Break.Set sieve /VarWrite i
```

The following options are only used together with the on-chip trigger unit of the processor. Please refer to the **TrOnchip** commands.

Alpha	Set an Alpha breakpoint.
Beta	Set an Beta breakpoint.
Charly	Set an Charly breakpoint.
Delta	Set an Delta breakpoint.
Echo	Set an Echo breakpoint.

```
                              ; Example for MPC500/800
                              ; Generate a pulse on the processor pin IWP0
                              ; if the function func1 is entered

Break.Set func1 Alpha         ; Set an Alpha breakpoint to the entry of
                              ; func1
TrOnchip.IW0 Ibus Alpha       ; The addresses marked with Alpha
                              ; breakpoints define the Ibus address
TrOnchip.IW0 WATCH ON         ; Generate a pulse on IWP0 when IW0 is hit
```

If the option **Spot** is selected, the program execution is only stopped shortly to update the TRACE32 screen when the breakpoint is hit. As soon as the screen is updated, the program execution continues. Each stop at a breakpoint with the option **Spot** takes approximately 50 … 100 ms.

Spot	Set the option Spot for a breakpoint.

```
Break.Set func7 /Program /Spot            ; When the program breakpoint
                                          ; at the entry of function
                                          ; func7 is hit update the
                                          ; TRACE32 screen.

Break.Set data /Write /Spot               ; Update the TRACE32 screen
                                          ; when a write access to the
                                          ; address data occurred.

Var.Break.Set flags[3] /Write /Spot       ; Update the TRACE32 screen
                                          ; when a write access to the
                                          ; variable flags[3] occurred.
```

The following options can be used, if they are supported by the used processor, they are deactivated otherwise.

WATCH	If the option WATCH is set, the program execution is not stopped at a breakpoint hit, the WATCH facility of the processor is activated instead. **Examples for the WATCH facility are:** Watchpoint Hit Messages with NEXUS; a short pulse on a watchpoint pin for the MPC5xx family etc.
BusTrigger	If the option **BusTrigger** is set, the program execution is not stopped at a breakpoint hit, a pulse for the internal trigger bus of the TRACE32 development tool is generated instead. For information about the internal trigger bus refer to the **TrBus** command.
BusCount	If the option **BusCount** is set, the program execution is not stopped at a breakpoint hit, the breakpoint hits are counted by the TRACE32 counter system instead. For more information about the TRACE32 counter system refer to the **Counter** command.

Examples:

```
Break.Set sieve /Program /Watch           ; Activate the WATCH facility of
                                          ; your processor when the
                                          ; function sieve is entered.
```

```
Break.Set sieve /Program /BusTrigger      ; Generate a 100 ns pulse for
                                          ; the TRACE32 internal trigger
                                          ; bus when the function sieve
                                          ; is entered
TrBus.RESet                               ; Configure the TRACE32 internal
                                          ; trigger bus
TrBus.Connect Out                         ; The TRIGGER connector of the
                                          ; TRACE32 development tool works
                                          ; as output
TrBus.Mode Low                            ; A 100 ns low pulse is
                                          ; generated on TRIGGER
```

```
Break.Set sieve /Program /BusCount        ; Count the entries to the
                                          ; function sieve
Count.RESet
Count.Mode EventHigh
```

The following options are available if a trace is used and trace control features are provided either by the used processor or by the TRACE32 hardware. These options are deactivated otherwise.

TraceEnable	Enable the trace on the specified event.
TraceData	Sample the complete program flow and the specified data event.
TraceON	Switch the sampling to the trace ON on the specified event.
TraceOFF	Switch the sampling to the trace OFF on the specified event.
TraceTrigger	Stop the sampling to the trace on the specified event. A trigger delay is possible.

Examples:

```
; Sample only the function entries to func5 to the trace buffer
Break.Set func5 /Program /TraceEnable

; Sample only write accesses to the variable vint into the trace buffer
Var.Break.Set vint /Write /TraceEnable
```

```
; Sample the complete program flow plus all write accesses to the
; variable vlong into the trace buffer
Var.Break.Set vlong /Write /TraceData
```

```
; Start the sampling to the trace buffer, when the function func7 is
; entered and stop the sampling to the trace buffer after the variable
; WriteBuffer was read
Break.Set func7 /Program /TraceON
Var.Break.Set WriteBuffer /Read /TraceOFF
```

```
; Sample another 2000. records to the trace buffer after the function
; func23 was entered
Break.Set func23 /Program /TraceTrigger
Trace.TDelay 2000.
```

DISable	Set the specified breakpoint, but disable it.
DISableHit	Disable the breakpoint after it was hit.
DeleteHIT	Delete the breakpoint when it is hit.
NoMark	Don't display a breakpoint indicator on the TRACE32 screen.

EXclude	The breakpoint is inverted:
	• by the inverting logic of the on-chip trigger unit
	• by setting the specified breakpoint to the following 2 address ranges
	`0x0--(start_of_breakpoint_range -1)`
	`(end_of_breakpoint_range+1)--end_of_memory`
	
	The **EXclude** option only applies to the implementation Onchip or Hardware.
	If the implementation is Onchip and the Onchip trigger unit does not provide an inverting logic, the processor has to provide the facility to set the specified breakpoint type on 2 address ranges.

Examples:

```
; Set a Write breakpoint to the address data but disable it
Break.Set data /Write /DISable
```

```
; Set a Program breakpoint to the entry of the function sieve. Disable
; the breakpoint after it was hit.
Break.Set sieve /Program /DISableHit
```

```
; Set a Program breakpoint to the entry of the function sieve.
; delete the breakpoint when it is hit.
Break.Set sieve /Program /DeleteHIT
```

```
; Set a Write breakpoint to the code range 0x3F000--0x3FAFF to make sure
; that no write access happens to your code range, but suppress the
; display of a break indicator
Break.Set 0x3F000--0x3FAFF /Write /NoMark
```

```
; Stop the program execution when a instruction outside of the function
; sieve accesses the variable flags
Var.Break.Set sieve; /VarReadWrite flags; /EXclude
```

The following options allow to stop the program execution when a specific data value is read or written.

DATA.Byte *<value>*	Define the data value for a byte access.
DATA.Word *<value>*	Define the data value for a word access.
DATA.Long *<value>*	Define the data value for a long access.
DATA.Quad *<value>*	Define the data value for a quad access.
DATA.TByte *<value>*	Define the data value for a triple-byte access.
DATA.HByte *<value>*	Define the data value for a hexabyte access.
DATA.auto *<value>*	Define the data value for an HLL variable. The access width is taken from the HLL information. If there is no HLL information available, the architecture width is taken.

Examples:

```
; Stop the program execution when 0x33 is written to the address buffer
; via a byte write
Break.Set buffer /Write /DATA.Byte 0x33
```

```
; Stop the program execution when 0xf00023aa is read from the address
; long_value via a long read
Break.Set long_value /Read /DATA.Long 0xf00023aa
```

```
; Stop the program execution when 0x0 is written to the variable
; flags[3]
Var.Break.Set flags[12] /Write /DATA 0x0
```

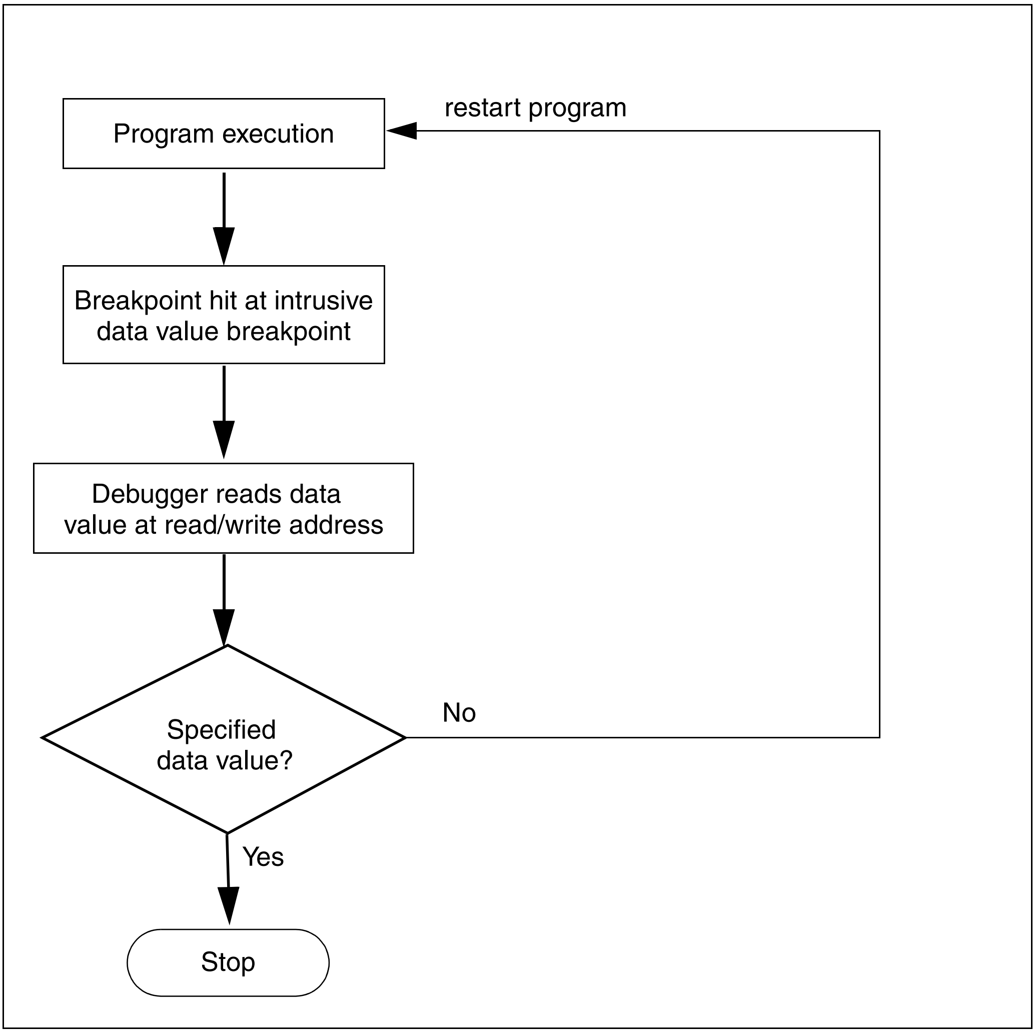
```
Break.Set word_value /Write /DATA.Word 0yxxxxxxxxxxxxxxx1
```

- Not all data widths are supported for all architectures. Quad will normally not be available for most 8-, 16- or 32-bit architectures. TByte and HByte are only available for specific DSP architectures.

- If the processor provides data value breakpoints (see **“On-chip Breakpoints”**, page 84) a **real-time data value breakpoint** is possible.

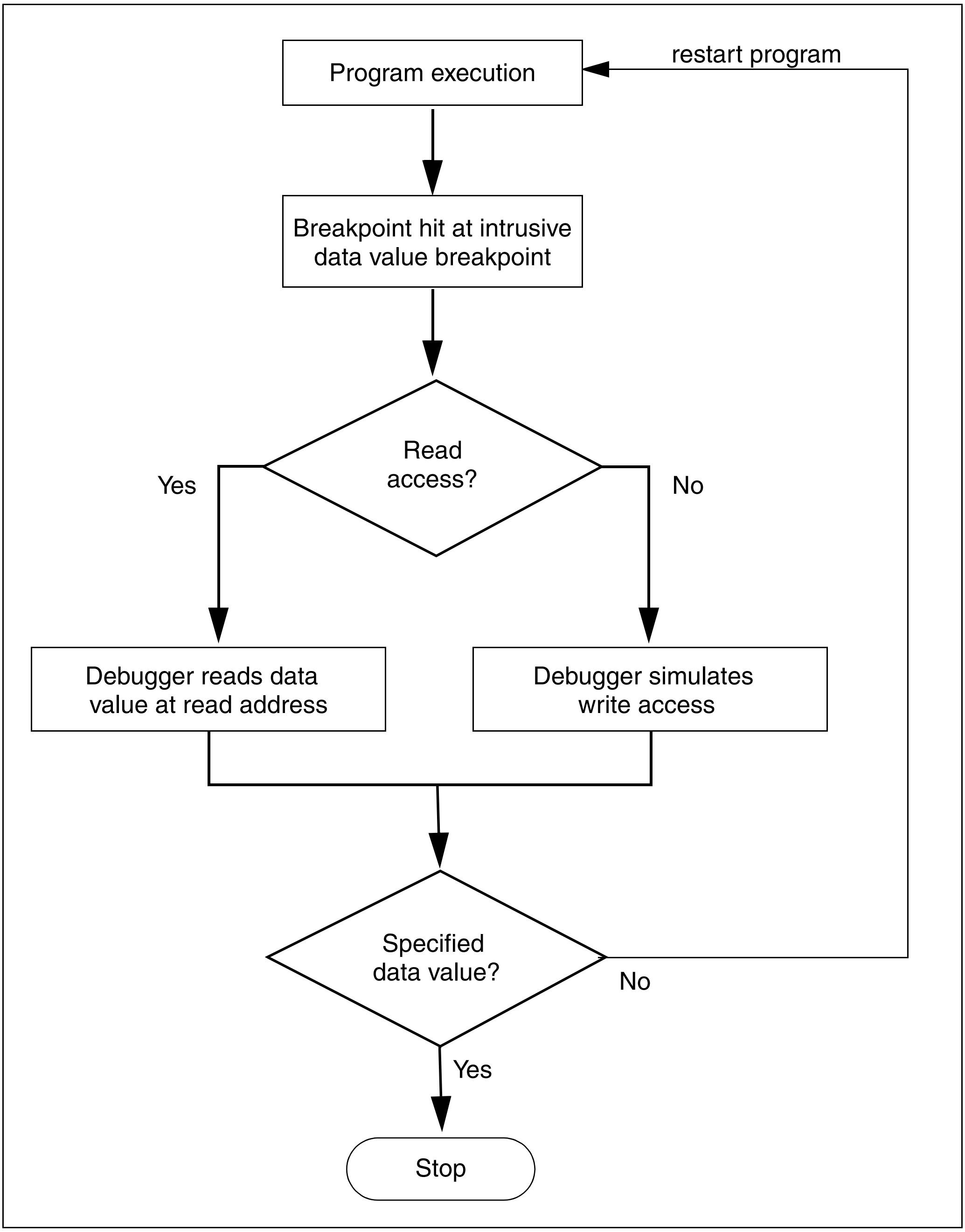
- TRACE32 provides **an intrusive data value breakpoint**, if the processor does not provide data value breakpoints.

An intrusive data value breakpoint for “**break after make**” processors is implemented as follows:



Program execution
restart program
Breakpoint hit at intrusive data value breakpoint
Debugger reads data value at read/write address
Specified data value?
No
Yes
Stop

An intrusive data value breakpoint for “**break before make**” processors is implemented as follows:


Program execution
restart program
Breakpoint hit at intrusive data value breakpoint
Read access?
Yes
No
Debugger reads data value at read address
Debugger simulates write access
Specified data value?
No
Yes
Stop

⚠	A intrusive data value breakpoint on a memory-mapped I/O register can result in a failing read or destructive write access.

TASK *<task_magic>*, etc.	If OS-aware debugging is configured, TASK-aware breakpoints allow to stop the program execution at a breakpoint only if the specified task/process is running. TASK-aware breakpoints are implemented on most cores as intrusive breakpoints. A few cores support real-time TASK-aware breakpoints (e.g ARM/Cortex). See also **“What to know about the Task Parameters”** (general_ref_t.pdf).
MACHINE *<machine_id>*, etc.	Specify the machine where you want to set the breakpoint. The breakpoint action, such as **stop**, takes effect only if the program is executed on the specified machine. The breakpoint is a real-time breakpoint if the processor architecture provides a machine ID register. Otherwise the breakpoint is an intrusive breakpoint. See also **“What to know about the Machine Parameters”** (general_ref_t.pdf).
CORE *<number>*	Specify the core where you want to set the breakpoint. The breakpoint action, such as **stop**, takes effect only if the program is executed on the specified core.

If a hex number is entered to identify the **TASK**, it is interpreted as **task magic number**.

```
; Stop the program execution at the entry to func12 only if the task
; with the magic 0xC2034000 is running
Break.Set func12 /Program /TASK 0xC2034000
```

If a decimal number is entered to identify the **TASK**, it is interpreted as **task ID**. If the OS does not assign a task ID, the decimal number is interpreted as magic instead.

```
; Stop the program execution at the entry to func9 only if the task
; with the ID 14. is running
Break.Set func9 /Program /TASK 14.
; if the RTOS doesn’t assign IDs, the ID is interpreted as magic
```

```
; Stop the program execution at the entry to func7 only if the task with
; the name task5 is running
Break.Set func7 /Program /TASK "task5"
```

Task-specific real-time breakpoints are available for:

ARM7/ARM9	By chaining the 2 on-chip breakpoints.
ARM11	Via the Context ID register.
Cortex-A/-R/-X	Via the Context ID register.
ColdFire	Via ASID (Address Space Identifier) for V4 architecture.
MMDSP8820	Via thread_ref register.
Neoverse	Via the Context ID register.
RISC-V	Via textr debug register.

Examples:

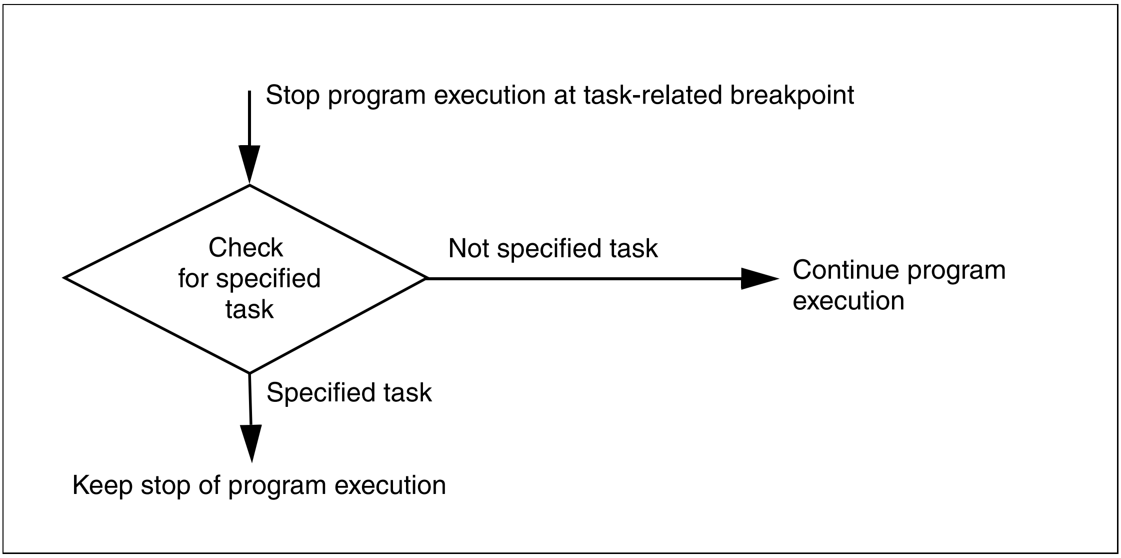
```
; example ARM7/ARM9
; disable all on-chip breakpoints
Break.DISable /Onchip
; set task-specific real-time breakpoint
Break.Set buzzer_high /Program /Onchip /TASK smxKillTask
```

```
; example ARM11/Cortex/Neoversetextra
; inform the debugger that your OS serves the Context ID register
Break.CONFIG.UseContextID ON
Break.Set DPhysicalDevice::Info /Program /Onchip /TASK EKern.exe:Thread1
```

```
; example for MMDSP8820
; if the OS serves the thread_ID register
Break.Set buzzer_high /Program /Onchip /TASK smxTask
```

```
; example for RISC-V
; if the OS serves the scontext register
Break.Set buzzer_high /Program /Onchip /TASK smxTask
```

If no task-specific real-time breakpoints are available, task-specific breakpoints are implemented as intrusive breakpoints.


Stop program execution at task-related breakpoint
Check for specified task
Not specified task
Continue program execution
Specified task
Keep stop of program execution

COUNT *<value>*	Stop the program execution after *<value>* breakpoint hits. **Implementation:** If the on-chip trigger unit provides a counter and the breakpoint is implemented as Onchip, this counter is used. Otherwise the program execution is stopped shortly at each breakpoint hit, the counter is incremented and the program execution is restarted if the current counter value is smaller then *<value>*. The current counter value is displayed in the **Break.List** window. Use the **Break.Init** command to reset the counter.

The on-chip debug units for the following processor architectures provide on-chip counters:

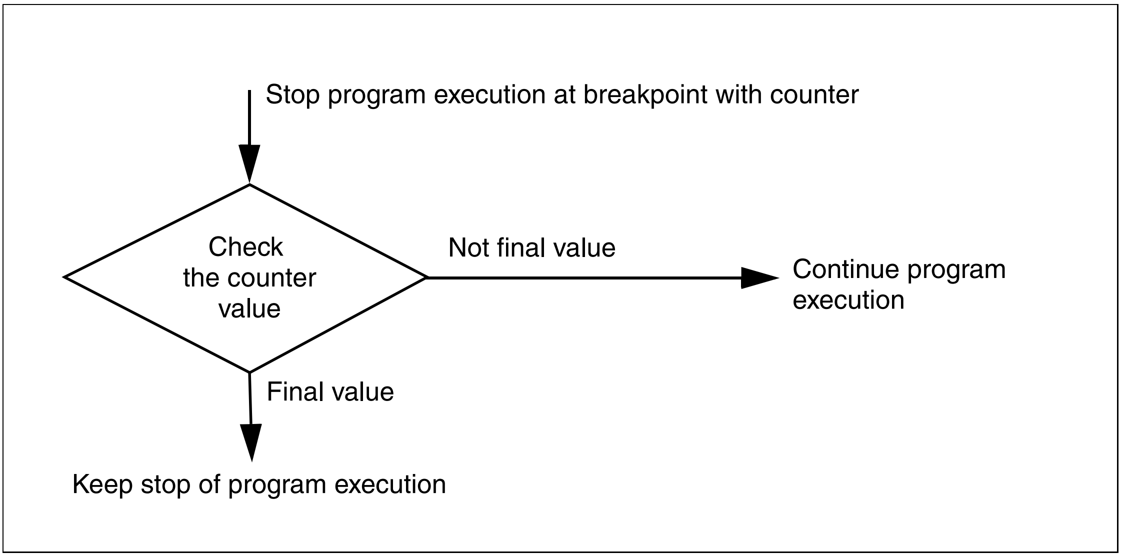
Architecture	**On-chip Counters**
MMDSP	1 x 16-bit counter
MPC500/800	2 x 16-bit counter for instructions 2 x 16-bit counter for data
MPC5500	2 x 16-bit counter (not MPC551x)
SH2A	1 x 12-bit counter
SH4	2 x 32-bit counter
StarCore	1 x 30-bit counter
Super10	1 x 16-bit counter

On-chip counter allow to count the event of interest in real-time. TRACE32 uses the on-chip counters only if the implementation **/Onchip** is used:

Example:

```
; Stop the program execution after 5 entries to func25

Break.Set func25 /Program /Onchip /COUNT 5.
```

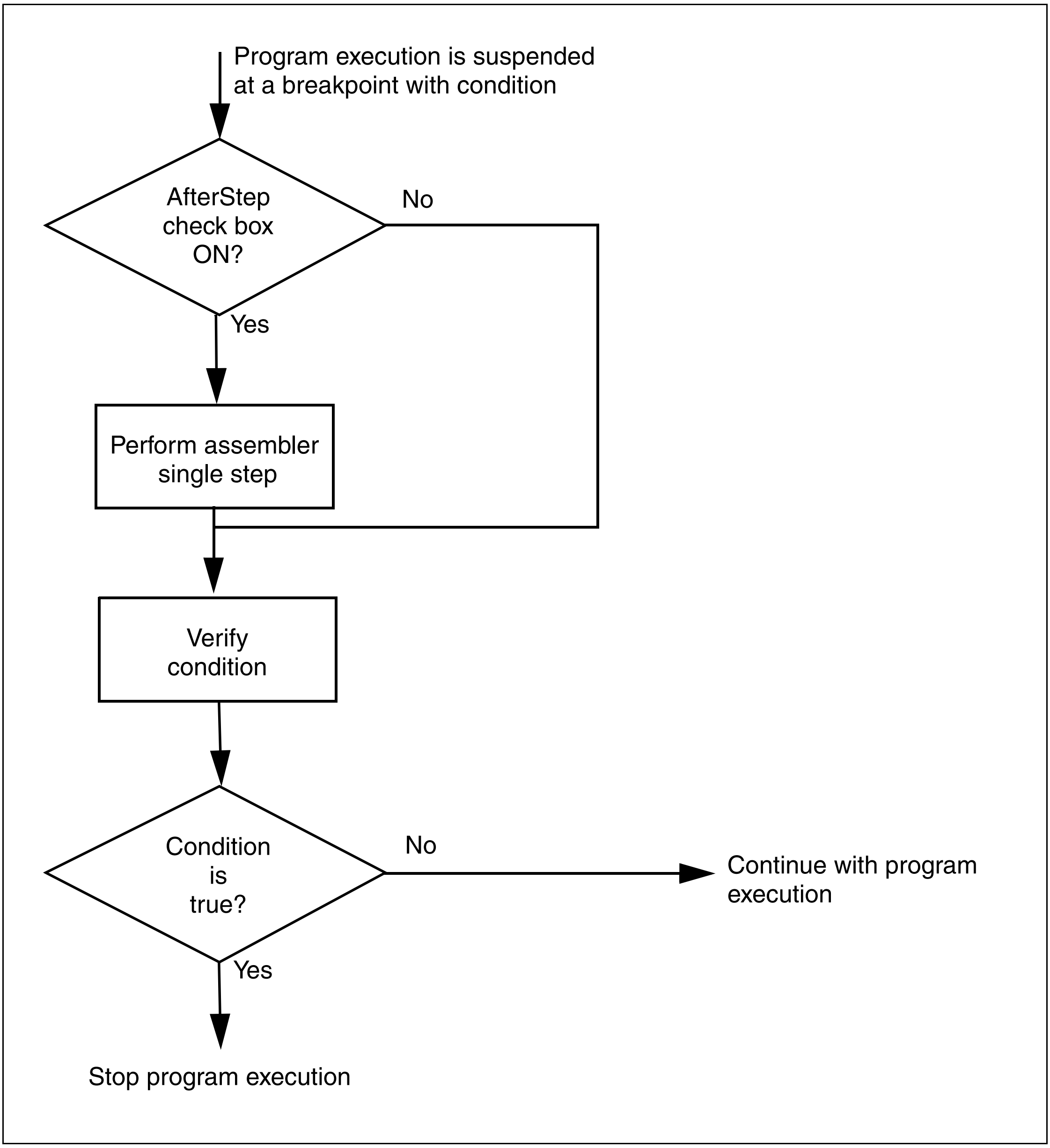
If no on-chip counter is provided by the on-chip debug unit or if the implementing **/SOFT** is used for a Program breakpoint, an intrusive breakpoint is used to count the event of interest.


Stop program execution at breakpoint with counter
Check the counter value
Not final value
Continue program execution
Final value
Keep stop of program execution

Example:

```
; Stop the program execution after 5 entries to func25

Break.Set func25 /Program /SOFT /COUNT 5.
```

CONDition *<expression>*	The program execution is only stopped at the breakpoint if the specified condition is true. The condition has to be defined in the TRACE32 syntax **(intrusive breakpoint)**.
VarCONDition *<hll_expression>*	The program execution is only stopped at the breakpoint if the specified HLL condition is true. The condition has to be defined in the syntax of your programming language **(intrusive breakpoint)**.
AfterStep	AfterStep forces TRACE32 to perform an assembler step before the specified condition is verified. This option might be useful: • If a Program breakpoint with condition is set to a register-indirect call instruction • If a Read/Write breakpoint with condition is set and the processor architecture under debug stops before the read/write access occurred.


Program execution is suspended at a breakpoint with condition
AfterStep check box ON?
No
Yes
Perform assembler single step
Verify condition
Condition is true?
No
Continue with program execution
Yes
Stop program execution

Examples:

```
; Stop the program execution at the instruction address 0x2228 only if
; the contents of Register R7 is greater 5.
Break.Set 0x2228 /Program /CONDition Register(R7)>5
```

```
; Stop the program execution at the register-indirect call at 0x2228
; only if the contents of Register R7 is greater 5, perform the register-
; indirect call before the condition is verified
Break.Set 0x2228 /Program /CONDition Register(R7)>5 /AfterStep
```

```
; Stop the program execution at a write access to vint only if flags[12]
; is equal to 0
Var.Break.Set vint /Write /VarCONDition (flags[12]==0)
```

```
; Stop the program execution at a write access to vint only if flags[12]
; is equal to 0 and vint is greater 10
; perform an assembler single step because the processor architecture
; stops before the write access occurs (break-before make breakpoint)
Var.Break.Set vint /Write /VarCOND (flags[12]==0)&&(vint>10.) /AfterStep
```

```
; Stop the program execution at the instruction address 0x2228 only if
; the contents of address 0x1234 has value of 0x55.
Break.Set 0x2228 /Program /CONDition Data.Word(D:0x1234)==0x55
```

CMD *<string>*	Execute one or more TRACE32 commands when the breakpoint is hit.
RESUME [**ON** | **OFF**]	**ON:** Restart the program execution after the commands are executed. Please be aware that the execution of a single TRACE32 commands takes at least 200 ms. **OFF:** The program execution is not resumed.

It is recommended to set RESUME to OFF, if CMD

- starts a PRACTICE script with the command **DO**
- commands are used that open processing windows like **Trace.STATistic.Func**, **Trace.Chart.sYmbol** or **CTS.List**

because the program execution is restarted before these commands are finished.

Example:

```
; Save the contents to register R12 to the file outreg1.lst whenever
; the breakpoint is hit.

Open #1 outreg1.lst /Create
Break.Set sieve\17 /Program /CMD "write #1 ""R12="" register(r12)"
/RESUME
Close #1
```

DIALOG	Open a standard **Break.Set** dialog for the breakpoint configuration.
DIALOGADVANCED	Open a full **Break.Set** dialog for the breakpoint configuration (**advanced** features).

See also

- ■ Break.CONFIG
- ■ Break.direct
- ■ Var.Break.Set

- ▲ 'Release Information' in 'Legacy Release History'
- ▲ 'Breakpoint Handling' in 'Training Basic Debugging'
- ▲ 'Breakpoint Handling' in 'Training Basic SMP Debugging'

Format:	**Break.SetFunc** [*<range>* | *<module>*] [**/***<option>*]
<option>:	**TAGS** **ALLRET** **ALLBX** **ODD** **ONLYAB** **SIMPLE** **PATCH** **INTR | NOINTR** **Program** **TraceONOFF** **BreakReturn** **SPOT**

Without parameter, the entry point of all HLL functions is marked with an **Alpha** breakpoint and the exit point with a **Beta** breakpoint. Otherwise, only the specified function(s)/range(s) is/are marked with **Alpha** and **Beta**. The breakpoints can then be used for statistic analysis (see **Analyzer.STATistic**) or for function runtime and nesting displays (see **Analyzer.List**).

ALLBX (only ARM)	Tags any BX instruction found in the function.
ALLRET	Tags any return instruction found in the function. This option is useful when the compiler produces more than one exit point for a function, but doesn't inform the debugger about it.
INTR	Marks the beginning of the function by Alpha and Charly, as required for interrupt programs.
ONLYAB	Uses only **Alpha** and **Beta** breakpoints. When the **INTR** option is also set then the combination of both will be used to mark interrupt functions. The option is required when the **Charly** is not available (e.g. when using ROM breakpoints on the C167 Bondout).
PATCH	Uses debug patch information.
SIMPLE	Tags the last instruction of a function even when the default strategy for determining the end of a function would be different.
TAGS	Processors with cache or prefetch can cause serious problems for statistic analysis. The best workaround is to make a data access base analysis. For this purpose extra code is added at each function entry and exit. This code writes to two variable to tag the entry or exit of a function. With the option **TAGS** all symbols beginning with '_r_' are marked with Alpha and Beta. These symbols can be generated by Microtec compilers to support performance analysis.

TraceONOFF	Sets TraceON/TraceOFF breakpoints.
BreakReturn	Sets stopping breakpoints at function returns.
SPOT	For a description, see **Break.Set Spot**.

Examples:

```
Break.SetFunc                      ; marks all functions

Break.SetFunc 0x1000--0x2fff       ; marks functions in address range

Break.SetFunc \mcc                 ; marks all functions in one module
```

See also

■ Break.direct

▲ 'Release Information' in 'Legacy Release History'

Break.SetLine   Mark HLL lines

Format: **Break.SetLine** [*<range>* | *<module>* | *<function>*] [**/***<option>*]

<option>: **Alpha** | **Beta** | **Charly**

The HLL lines are marked with **Alpha** breakpoints. The breakpoints are set short after the first instruction of the line, to prevent the access of the breakpoint by a prefetch of the CPU. The breakpoints can be used either for HLL line sampling or for performance analysis on HLL line (see **Analyzer.STATistic.Line**).

Examples:

```
Break.SetLine                      ; marks all lines

Break.SetLine 0x1000--0x2fff       ; marks lines in address range

Break.SetLine \mcc                 ; marks lines in one module

Break.SetLine main                 ; marks lines in function 'main'
```

See also

■ Break.direct

Break.SetMONitor Switch to run mode debugging at the next “Go”

Format: **Break.SetMONitor** [**ON** | **OFF**]

Switches to run mode debugging at the next **Go**.

See also

- ■ Break.direct
- ■ Break.MONitor
- ■ Go.MONitor

Break.SetPATtern Set breakpoints allowing wildcards

Format: **Break.SetPATtern** *<symbol_pattern>* [**/***<type>*]

Sets breakpoints allowing the wildcards ? and *. For details on setting breakpoints, refer to the **Break.Set** command.

Example:

```
Break.SetPATtern *memory* /Program         ; set program breakpoints to
                                           ; all debug symbols that
                                           ; contain the string "memory".
```

See also

- ■ Break.direct
- ▲ ’Release Information’ in ’Legacy Release History’

Break.SetTask Stop the program execution when task is scheduled

Format: **Break.SetTask** *<task_magic>* | *<task_id>* | *<task_name>*

Sets a breakpoint to stop as soon as the task is scheduled. This function is only available, if the debugger is configured with the appropriate OS Awareness.

Depending on the capabilities of the OS and the OS Awareness, this command may set a conditional breakpoint onto the OS variable that holds the current task, or a breakpoint to stop as soon as the saved PC of this task is read. The program execution will be stopped inside the kernel scheduler. You can then step up to the calling task manually.

<task_magic>, etc.	See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

Examples:

```
Break.SetTask 7.             ; set a breakpoint to the next entry of the
                             ; task with the ID 7

Break.SetTask "module1"      ; set a breakpoint to the next entry of the
                             ; task module1
```

See also

- Break.direct

Break.ViewProgram Show state of the CTL trigger unit

Format:	**Break.ViewProgram**

Opens a windows that shows the state of the Complex Trigger Language (CTL) trigger unit.

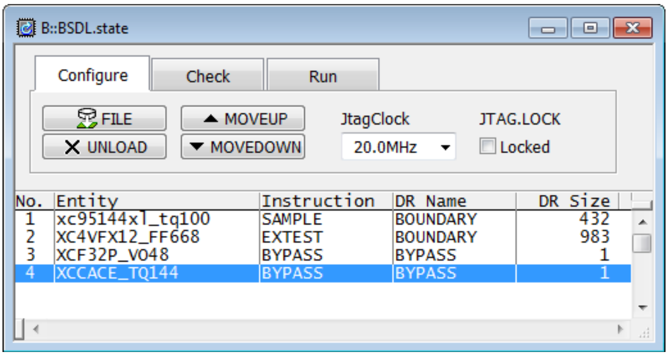
See also

- Break.CLEAR
- Break.direct
- Break.PASS

BSDL Boundary scan description language

The **BSDL** commands are used for reading boundary scan description language (IEE1149-1) files, performing boundary scan tests and program external flash memories via the boundary scan chain. For more information and step-by-step procedures, refer to **“Boundary Scan User´s Guide”** (boundary_scan.pdf).

For configuration, use the TRACE32 command line, a PRACTICE script (*.cmm), or the **BSDL.state** window.



JtagClock 20.0MHz

JTAG.LOCK □ Locked

```
No. Entity           Instruction DR Name   DR Size
 1  xc95144xl_tq100  SAMPLE      BOUNDARY      432
 2  XC4VFX12_FF668   EXTEST      BOUNDARY      983
 3  XCF32P_VO48      BYPASS      BYPASS          1
 4  XCCACE_TQ144     BYPASS      BYPASS          1
```

The following TRACE32 ■ commands and ❑ functions() are available to configure the boundary scan chain.

See also

- ■ BSDL.BYPASSall
- ■ BSDL.CHECK
- ■ BSDL.FILE
- ■ BSDL.FLASH
- ■ BSDL.HARDRESET
- ■ BSDL.IDCODEall
- ■ BSDL.LINKAGE
- ■ BSDL.LoadDR
- ■ BSDL.MOVEDOWN
- ■ BSDL.MOVEUP
- ■ BSDL.ParkState
- ■ BSDL.RESet
- ■ BSDL.RUN
- ■ BSDL.RUNTCK
- ■ BSDL.SAMPLEall
- ■ BSDL.SELect
- ■ BSDL.SET
- ■ BSDL.SetAndRun
- ■ BSDL.SOFTRESET
- ■ BSDL.state
- ■ BSDL.StepPauseDR
- ■ BSDL.SToreDR
- ■ BSDL.TwoStepDR
- ■ BSDL.UNLOAD
- ❑ BSDL.GetDRBit()
- ❑ BSDL.GetPortLevel()

- ▲ ’What to know about Boundary Scan’ in ’Boundary Scan User’s Guide’
- ▲ ’FLASH Programming via Boundary Scan’ in ’eMMC FLASH Programming User’s Guide’
- ▲ ’Boundary Scan Description Language (BSDL) Functions’ in ’General Function Reference’
- ▲ ’FLASH Programming via Boundary Scan’ in ’Serial FLASH Programming User’s Guide’

BSDL.BYPASSall Check bypass mode

Format: **BSDL.BYPASSall**

Sets all chips in the boundary scan chain in BYPASS mode and shifts a 32-bit random number through it. If this test fails, an error will be reported.

See also

■ BSDL ■ BSDL.state ❏ BSDL.CHECK.BYPASS()

BSDL.CHECK Enable test result checking

Format: **BSDL.CHECK ON | OFF**

Enables or disables the test result checking for boundary scan. When enabled all data register bits with expect high or low are checked after a **BSDL.RUN** / **BSDL.RUN DR** command. If a test fails, an error message is printed.

See also

■ BSDL ■ BSDL.state

BSDL.FILE Load a BSDL file

Format: **BSDL.FILE** *<file>*

Loads a BSDL file and places its entity on the current position in the boundary scan chain.

See also

■ BSDL ■ BSDL.LINKAGE ■ BSDL.state

BSDL.FLASH command group is used for programming non-volatile memories via boundary scan. The following protocols are supported:

- Common flash interface (NOR flash memory)
- I2C
- SPI
- eMMC

With the **BSDL.FLASH** commands the boundary scan chain is prepared for flash programming, the flash programming itself is done with either the **FLASH** or **FLASHFILE** commands.

See also

■ BSDL.FLASH.IFCheck ■ BSDL.FLASH.IFDefine ■ BSDL.FLASH.IFMap ■ BSDL.FLASH.INIT
■ BSDL ■ FLASH ■ FLASHFILE

▲ 'FLASH' in 'General Commands Reference Guide F'
▲ 'FLASHFILE' in 'General Commands Reference Guide F'

BSDL.FLASH.IFCheck Check flash interface definition

Format: **BSDL.FLASH.IFCheck**

Checks if flash definition is valid and all required flash ports are mapped to a device port. The check results are displayed in the area window.

NOR flash:

- Required ports (will cause an error, if not mapped):

  OE (output enable)

  WE (write enable)

  A0 - An (address ports, number n of address ports is defined with BSDL.FLASH.IFDefine)

  DQ0-DQm (data ports, number m of data ports is defined with BSDL.FLASH.IFDefine)

- Optional ports (will cause a warning, if not mapped):

  CE (chip enable)

  RB (ready/busy)

  BYTE (data bus width selection)

  RESET (flash hardware reset)

  WP (write protection/acceleration input)

SPI flash:

- Required ports (will cause an error, if not mapped):

  CE (chip enable / chip select)

  SCK (serial data clock)

  SI (serial data input)

  SO (serial data output)

I2C flash:

- Required ports (will cause an error, if not mapped):

  SDA (serial data)

  SCL (serial clock)

MMC flash:

- Required ports (will cause an error, if not mapped):

  CLK (Clock)

  CMD (Command)

  DAT0 - DATn (Data I/O)

See also

■ BSDL.FLASH ❏ BSDL.CHECK.FLASHCONF()

Format:	**BSDL.FLASH.IFDefine RESet** *<nor_param>* | *<spi_param>* | *<i2c_param>* | *<mmc_param>*
<nor_param>:	**NOR** *<chip_number> <address_size> <data_size>*
<spi_param>:	**SPI** *<chip_number>*
<i2c_param>:	**I2C** *<chip_number>*
<mmc_param>:	**MMC** *<chip_number> <data_size>*

Defines the flash memory configuration:

RESet Resets the BSDL flash configuration.

NOR Selects NOR flash memory type.

SPI Selects SPI flash memory type.

I2C Selects I2C flash memory type.

MMC Selects MMC flash memory type.

<chip_number> Number of the chip in the boundary scan chain to which the flash memory is connected.

<address_size> Number of address ports of the flash memory

<data_size> Number of data ports of the flash memory (max. 32 bit for NOR flash; 1, 4, or 8 bit for MMC flash)

```
BSDL.FLASH.IFDefine DELete                  ; deletes all BSDL flash
                                            ; configurations
BSDL.FLASH.IFDefine NOR 2. 23. 16.          ; defines a NOR flash on chip
                                            ; 2 of the boundary scan chain
                                            ; with 23 address ports (A0-
                                            ; A22)and 16 data ports (DQ0-
                                            ; DQ15)
```

See also

- BSDL.FLASH
- FLASH.BSDLaccess

Format: **BSDL.FLASH.IFMap** *<flash_port> <device_port>*

Maps the generic flash ports to the device ports.

<flash_port>	Generic flash port names **NOR flash**: • CE (chip enable), OE (output enable), WE (write enable), RB (ready busy), BYTE, RESET, WP (write protection) • CE2, OE2, WE2, RB2, BYTE2, RESET2, WP2 • CE3, OE3, WE3, RB3, BYTE3, RESET3, WP3 • CE4, OE4, WE4, RB4, BYTE4, RESET4, WP4 • A* (address), DQ* (data input/output) **SPI flash**: • CE (chip enable), SCK (serial clock), SI (Master output, slave input), SO (Master input, slave output) **I2C** (FLASH EEPROM): • SCL (serial clock), SDA (serial data) **MMC flash**: • CLK (Clock), CMD (Command), DAT0 - DAT7 (Data I/O)
<device_port>	Device port name (from the corresponding BSDL file, case insensitive)

Examples:

```
BSDL.FLASH.IFMap CE   PR7C            ; Maps the generic NOR flash port CE
                                      ; to the device port PR7C

BSDL.FLASH.IFMap DQ15 PR12A           ; Maps the generic NOR flash port
                                      ; DQ15 to the device port PR12A
```

See also

- BSDL.FLASH
- FLASH.BSDLaccess

BSDL.FLASH.INIT — Initialize flash interface

Format:	**BSDL.FLASH.INIT SAFE | SAMPLE | ZERO | ONE | NONE**

Initializes the boundary scan chain for flash programming. The boundary scan register of the device to which the flash memory is connected, will be initialized to the parameter value, the flash control ports will be set in the inactive state (all control ports set to ’1’, data output driver disabled, address ports set to ’0’).

The chip, which is connected to the flash memory is set to EXTEST mode, all other chips are set to BYPASS mode.

SAFE	The boundary scan register is initialized to the SAFE (defined in the corresponding BSDL file).
SAMPLE	A SAMPLE run is executed and the sampled data are taken for initialization.
ZERO	The boundary scan register is initialized to all zero.
ONE	The boundary scan register is initialized to all one.
NONE	The boundary scan register is not initialized, it must be initialized before with **BSDL.SET**, otherwise its state will be undefined.

See also

- BSDL.FLASH

BSDL.HARDRESET — TAP reset via TRST

Format:	**BSDL.HARDRESET**

TRST port is toggled and the TAP controllers are set to the “Select-DR-SCAN” state.

See also

- BSDL
- BSDL.state

BSDL.IDCODEall Check ID codes

Format:	**BSDL.IDCODEall**

Sets all chips in the boundary scan chain in IDCODE mode and checks the resulting ID codes. Chips, without an ID code register will be set in BYPASS mode. If this test fails, an error will be reported.

See also

■ BSDL ■ BSDL.state ❏ BSDL.CHECK.IDCODE()

BSDL.LINKAGE Create a bypass device

Format:	**BSDL.LINKAGE** *<IR size>*

Creates a bypass device with instruction size <IR size> and places its entity on the current position in the boundary scan chain.

See also

■ BSDL ■ BSDL.FILE ■ BSDL.state

BSDL.LoadDR Load data register from file

Format: **BSDL.LoadDR** *<chip_number> <register_name> <file>* [**/***<option>*]

<option>: **ASCII**
BINary

Loads the content of *<file>* into data register *<register_name>* of IC *<chip_number>*.

- If the *<file>* contains more date than data register *<register_name>*, the redundant data from the *<file>* will be ignored.
- If the *<file>* contains less data than data register *<register_name>* only the least significant bits of data register *<register_name>* will be loaded.

<chip_number> Number of IC in the boundary scan chain, if the boundary scan chain has only one IC, this parameter can be omitted.

<register_name> Data register name (must be defined in BSDL file).

<file> File with register data

ASCII File format is ASCII.

BINary File format is binary (default).

The **BINary** format is byte wise, the first byte will be the first 8 bit of the data register *<register_name>*.

The **ASCII** format is 1 bit per line. Line comments starts with “//”:

```
// IC001 = CPU_TEST
// DR    = USER_DATA[56]
1
0
0
0
0
1
0
0 // 21
1
```

See also

■ BSDL ■ BSDL.SToreDR

BSDL.MOVEDOWN

Move selected chip downwards

Format:	**BSDL.MOVEDOWN**

Moves the selected chip down by one position (i.e. increase chip number by one).

Chip is either selected by the command **BSDL.SELect** or in the **BSDL.state** window.

See also

■ BSDL ■ BSDL.state

BSDL.MOVEUP — Move selected chip upwards

Format: **BSDL.MOVEUP**

Moves the selected chip up by one position (i.e. decrease chip number by one).

Chip is either selected by the command **BSDL.SELect** or in the **BSDL.state** window.

See also

- BSDL
- BSDL.state

BSDL.ParkState — Select JTAG parking state

Format: **BSDL.ParkState Run-Test/Idle | Select-DR-Scan**

Selects the parking state for the JTAG state machine. The parking state is the state where the JTAG state machine will stop after a **BSDL.HARDRESET**, **BSDL.SOFTRESET** or a **BSDL.RUN** command. The default parking state after a **BSDL.RESet** is Run-Test/Idle.

Run-Test/Idle Selects Run-Test/Idle as parking state for the JTAG state machine

Select-DR-Scan Selects Select-DR-Scan as parking state for the JTAG state machine

If the parking states of the debug and the boundary scan functions are different, unintended side effects may occur. See **“Boundary Scan User´s Guide”** for details.

See also

- BSDL

BSDL.RESet — Reset boundary scan configuration

Format: **BSDL.RESet**

Deletes the boundary scan configuration and set all boundary scan options to their default values

See also

- BSDL

BSDL.RUN — Run JTAG sequence

Format: **BSDL.RUN** [**IR** | **DR**]

The **BSDL.RUN** command will apply (i.e. shift out) the instruction and data register settings to the boundary scan chain. Without any option, the instruction register settings are applied first and the data register settings are applied second.

IR With the option **IR** only the instruction register settings are applied,

DR With the option **DR** only the data register settings are applied.

When a DR shift is executed, the result data can be viewed in the settings/result window (opens with **BSDL.SET** *<chip_number>* or double click on the corresponding entry in the **BSDL.state** entity list).

See also

- BSDL
- BSDL.state

BSDL.RUNTCK — Toggle TCK

Format: **BSDL.RUNTCK** *<count>*

Toggles TCK for *<count>* clocks.

See also

- BSDL

BSDL.SAMPLEall

Sample all signals

Format:	**BSDL.SAMPLEall**

Sets all chips in the boundary scan chain in SAMPLE mode and runs a sample test. The results can be viewed in the result window (see **BSDL.SET**).

See also

■ BSDL ■ BSDL.state

BSDL.SELect

Select a chip

Format:	**BSDL.SELect** [*<chip_number>*]

Selects *<chip_number>* for the commands **BSDL.MOVEUP**, **BSDL.MOVEDOWN**, and **BSDL.FILE**.

- **BSDL.MOVEUP**, **BSDL.MOVEDOWN**: The selected chip is moved.
- **BSDL.FILE**: The loaded entity is placed after the selected chip

See also

■ BSDL ■ BSDL.state

Format: **BSDL.SET** [*<chip_number>*] [*<set_selection>*]

<set_selection>: *<ir_conf>* | *<dr_conf>* | *<bsr_conf>* | *<port_conf>* | *<pinmap_conf>* | *<options>*

<ir_conf>: **IR** *<instr_name>* | *<opcode>*

<dr_conf>: **DR** *<bit_slice>* **ZERO** | **ONE** | **ExpectH** | **ExpectL** | **ExpectX** | *<opcode>*

<bsr_conf>: **BSR** *<bit_slice>* **ZERO** | **ONE** | **SAFE** | **SAMPLE** | **DISable** | **ENable** | **Drive0** | **Drive1** | **ExpectH** | **ExpectL** | **ExpectX** | *<opcode>*

<port_conf>: **PORT** *<port_name>* **1** | **0** | **Z** | **H** | **L** | **X**

<pinmap_conf>: **PINMAP** *<pinmap_name>*

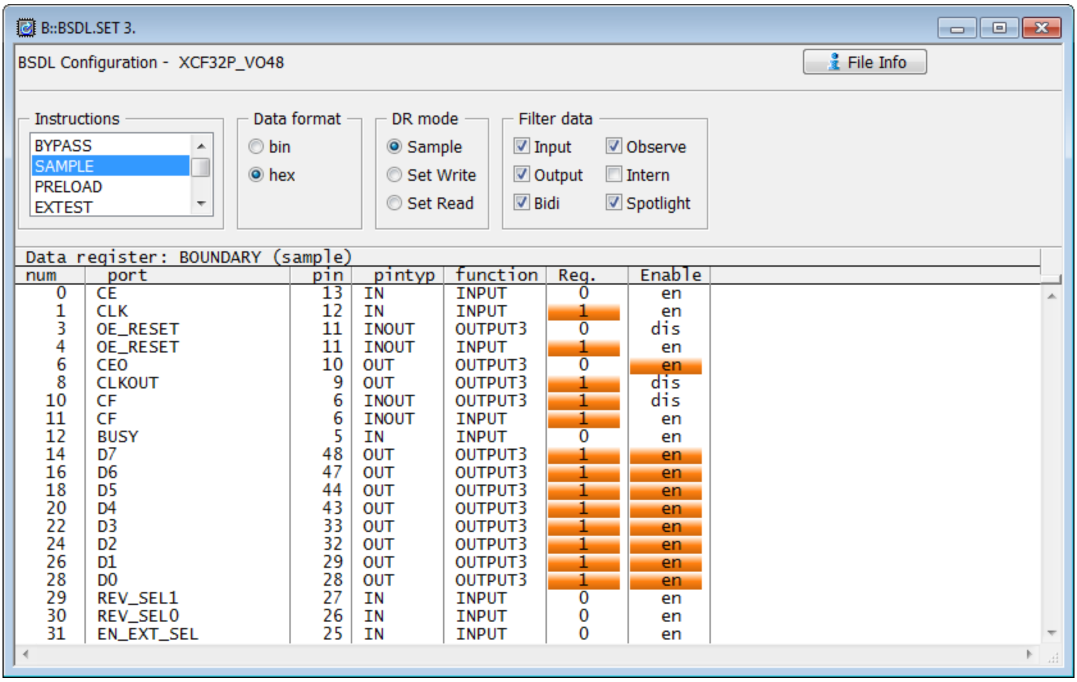
<options>: **OPTION IN** | **OUT** | **BIDI** | **OBSERVE** | **INTERN** | **ALL** | **SPOTLIGHT** | **MARKLINES** | **BSRHISTORY ON** | **OFF**

The command **BSDL.SET** modifies the instruction and data register settings for a chip in the boundary scan chain. The settings are applied to the system with **BSDL.RUN** command.

If the boundary scan chain has only chip, the *<chip_number>* can be omitted.

With *<chip_number>* as the only parameter **BSDL.SET** will open the settings/result window for *<chip_number>*.

```
BSDL.SET 4.              ; opens the settings/result window for chip 4
```



BSDL Configuration - XCF32P_VO48

Instructions: BYPASS, SAMPLE, PRELOAD, EXTEST (SAMPLE selected)

Data format: ○ bin ● hex

DR mode: ● Sample ○ Set Write ○ Set Read

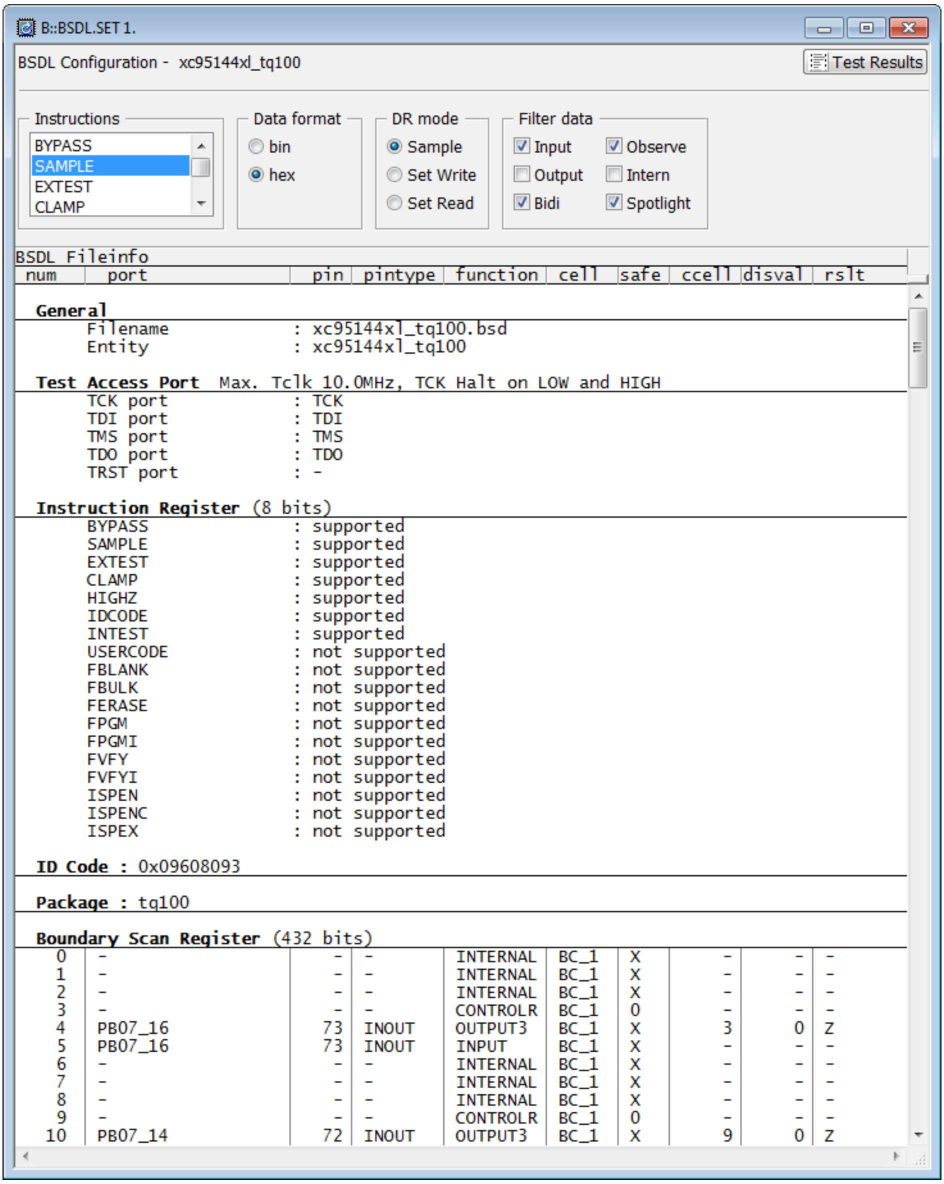
Filter data: ☑ Input ☑ Observe ☑ Output □ Intern ☑ Bidi ☑ Spotlight

Data register: BOUNDARY (sample)

num	port	pin	pintyp	function	Reg.	Enable
0	CE	13	IN	INPUT	0	en
1	CLK	12	IN	INPUT	1	en
3	OE_RESET	11	INOUT	OUTPUT3	0	dis
4	OE_RESET	11	INOUT	INPUT	1	en
6	CEO	10	OUT	OUTPUT3	0	en
8	CLKOUT	9	OUT	OUTPUT3	1	dis
10	CF	6	INOUT	OUTPUT3	1	dis
11	CF	6	INOUT	INPUT	1	en
12	BUSY	5	IN	INPUT	0	en
14	D7	48	OUT	OUTPUT3	1	en
16	D6	47	OUT	OUTPUT3	1	en
18	D5	44	OUT	OUTPUT3	1	en
20	D4	43	OUT	OUTPUT3	1	en
22	D3	33	OUT	OUTPUT3	1	en
24	D2	32	OUT	OUTPUT3	1	en
26	D1	29	OUT	OUTPUT3	1	en
28	D0	28	OUT	OUTPUT3	1	en
29	REV_SEL1	27	IN	INPUT	0	en
30	REV_SEL0	26	IN	INPUT	0	en
31	EN_EXT_SEL	25	IN	INPUT	0	en

Depending on the selected instruction, the data area of the settings/result window shows the results of the last DR scan operation. The instruction and the view options for the chip can be modified.

The information from the BSDL file can be viewed by toggling the data area to the “File info” view. It shows the provided instructions, compliance pattern, boundary scan register, TAP parameters, etc.



Instructions: BYPASS, SAMPLE, EXTEST, CLAMP (SAMPLE selected)

Data format: ○ bin ● hex

DR mode: ● Sample ○ Set Write ○ Set Read

Filter data: ☑ Input ☑ Observe ☐ Output ☐ Intern ☑ Bidi ☑ Spotlight

```
BSDL Fileinfo
 num      port          pin  pintype  function   cell   safe  ccell disval  rslt

  General
       Filename            : xc95144xl_tq100.bsd
       Entity              : xc95144xl_tq100

  Test Access Port  Max. Tclk 10.0MHz, TCK Halt on LOW and HIGH
       TCK port            : TCK
       TDI port            : TDI
       TMS port            : TMS
       TDO port            : TDO
       TRST port           : -

  Instruction Register (8 bits)
       BYPASS              : supported
       SAMPLE              : supported
       EXTEST              : supported
       CLAMP               : supported
       HIGHZ               : supported
       IDCODE              : supported
       INTEST              : supported
       USERCODE            : not supported
       FBLANK              : not supported
       FBULK               : not supported
       FERASE              : not supported
       FPGM                : not supported
       FPGMI               : not supported
       FVFY                : not supported
       FVFYI               : not supported
       ISPEN               : not supported
       ISPENC              : not supported
       ISPEX               : not supported

  ID Code : 0x09608093

  Package : tq100

  Boundary Scan Register (432 bits)
   0 | -                  -  | -      | INTERNAL | BC_1 | X    |      - |    - | -
   1 | -                  -  | -      | INTERNAL | BC_1 | X    |      - |    - | -
   2 | -                  -  | -      | INTERNAL | BC_1 | X    |      - |    - | -
   3 | -                  -  | -      | CONTROLR | BC_1 | 0    |      - |    - | -
   4 | PB07_16           73  | INOUT  | OUTPUT3  | BC_1 | X    |      3 |    0 | Z
   5 | PB07_16           73  | INOUT  | INPUT    | BC_1 | X    |      - |    - | -
   6 | -                  -  | -      | INTERNAL | BC_1 | X    |      - |    - | -
   7 | -                  -  | -      | INTERNAL | BC_1 | X    |      - |    - | -
   8 | -                  -  | -      | INTERNAL | BC_1 | X    |      - |    - | -
   9 | -                  -  | -      | CONTROLR | BC_1 | 0    |      - |    - | -
  10 | PB07_14           72  | INOUT  | OUTPUT3  | BC_1 | X    |      9 |    0 | Z
```

Instruction register settings

IR Selects the instruction register for the **BSDL.SET** command.

<instr_name> Instruction names of the selected chip: SAMPLE/PRELOAD, BYPASS, EXTEST
Depending on the chip more instructions may be available.

<opcode> One or more 64 bit integer values, only n(=instruction register size) bits are used other bits will be ignored.

```
BSDL.SET 4. IR SAMPLE       ; sets chip 4 in SAMPLE mode

BSDL.SET 4. IR 0x023        ; sets the instruction register of chip 4 to
                            ; 0x023 (bits > instruction size will be
                            ; ignored
```

Data register settings

DR

Selects the data register for the **BSDL.SET** command. The currently selected instruction determines the data register size, the upper index of the bit slice will be cut, if it exceed the data register size

<bit_slice>

Bit slice can be:
i--k: the bits from i to k will be modified
i : bit i will be modified
* : all bits will be modified

ZERO

The selected bit slice will be set to zero.

ONE

The selected bit slice will be set to one.

ExpectH

The selected bit slice will be set to “expect high” (for read register)

ExpectL

The selected bit slice will be set to “expect low” (for read register)

ExpectX

The selected bit slice will be set to “ignore” (for read register)

<opcode>

One or more 64 bit integer values, only n (=bit slice size) bits are used other bits will be ignored.

```
BSDL.SET 4. DR 3.--16. ONE     ; sets the bits 3..16 of chip 4 to one

BSDL.SET 4. DR 0 ZERO          ; sets the bit 0 of chip 4 to zero

BSDL.SET 1. DR * 0x1234        ; sets data register of chip 1 to 0x1234
                               ; all bits > 15 will be set to zero
```

Boundary scan register settings

BSR

Selects the boundary scan register for the **BSDL.SET** command. Register size is equal to the boundary scan register size, the upper index of the bit slice will cut, if it exceed the register size

<bit_slice>

Bit slice can be:
i--k : the bits from i to k will be modified
i : bit i will be modified
* : all bits will be modified

ZERO The selected bit slice will be set to zero.

ONE The selected bit slice will be set to one.

SAFE The selected bit slice will be set to the SAFE state (according the BSDL file).

SAMPLE The selected bit slice will be set to previously sampled data.

DISable All ports in the selected bit slice will be disabled (if a control cell is defined)

ENable All ports in the selected bit slice will be enabled (if a control cell is defined)

Drive0 All ports in the selected bit slice will drive ’0’, if the port is an output or bidi. Output drivers will be enabled, if required.

Drive1 All ports in the selected bit slice will drive ’1’, if the port is an output or bidi. Output drivers will be enabled, if required.

ExpectH The selected bit slice will be set to “expect high” (for read register)

ExpectL The selected bit slice will be set to “expect low” (for read register)

ExpectX The selected bit slice will be set to “ignore” (for read register)

<opcode> One or more 64 bit integer values, only n (=bit slice size) bits are used other bits will be ignored.

The settings for the boundary scan register are only meaningful in PRELOAD, EXTEST or INTEST mode.

```
BSDL.SET 4. BSR * SAMPLE       ; initializes the boundary scan register
                               ; of chip 4 with a previous sample run
BSDL.SET 1. BSR * SAFE         ; initializes the boundary scan register
                               ; of chip 1 to SAFE values
BSDL.SET 1. BSR 2--7 Drive0    ; drive 0 to the ports which are control-
                               ; led by the register bits 2..7 of chip 1
```

Port settings

PORT Selects the port settings for the **BSDL.SET** command. The boundary scan register is modified for this port (drive/expect value, enable/disable output)

<port_name> Name of the port, which should be modified. The port name must be listed in the definition of the boundary register in the BSDL file.

1 The selected port is set to drive 1 (only for output/bidir ports).

0 The selected port is set to drive 0 (only for output/bidir ports).

Z The selected port is set to drive 'Z' (only for output/bidir ports).

H The selected port is set to expect high (only for input/bidir ports).

L The selected port is set to expect low (only for input/bidir ports).

X The selected port is set to ignore result (only for input/bidir ports).

```
BSDL.SET 4. PORT PL7A 1         ; set port PL7A of IC4 to "drive 1"
BSDL.SET 3. PORT PS1  H         ; set port PS1  of IC3 to "expect high"
```

Pin map settings

PINMAP Selects the pin map settings for the **BSDL.SET** command.

<pinmap_name> Name of the pin map, which should be selected. It must be a valid pin map from the BSDL file.

This command can be used, if no default pin map is defined in the BSDL file or if it has multiple pin maps. It has only an effect on the data output shown in the **BSDL.SET** window (boundary register view, fileinfo view).

```
BSDL.SET 3. PINMAP TQFP_48      ; select pin map TQFP_48 for IC3
```

Option settings

The options can be turned on or off.

OPTION Selects the options menu for the command **BSDL.SET**.

IN Show/hide inputs in result window.

OUT Show/hide outputs in result window.

BIDI Show/hide bidi ports in result window.

OBSERVE Show/hide observer cells in result window.

INTERN Show/hide internal cells in result window.

ALL Show/hide all cells in result window.

SPOTLIGHT Enable/disable the spotlight function in result window (Sample mode).

MARKLINES Enable/disable alternating line colors in result window.

BSRHISTORY Enable/disable graphical history view for boundary scan register in result window (Sample mode).

```
BSDL.SET 4. OPTION IN ON          ; show inputs in the settings/result
                                  ; window for chip 4
BSDL.SET 4. OPTION INTERN OFF     ; hide internal registers in the
                                  ; settings/result window for chip 4
```

See also

- BSDL
- BSDL.state

BSDL.SetAndRun Immediate data register takeover

Format: **BSDL.SetAndRun ON | OFF**

Enables or disables the set and run feature. If enabled, a modification of a data register bit or bitslice will cause an immediate **BSDL.RUN**, i.e. the modified settings are applied immediately to the boundary scan register chain.

See also

- BSDL
- BSDL.state

Format:	**BSDL.SOFTRESET**

A TMS reset (5 TCK cycles with TMS=’1’) are executed and the TAP controllers are set to the “Select-DR-SCAN” state.

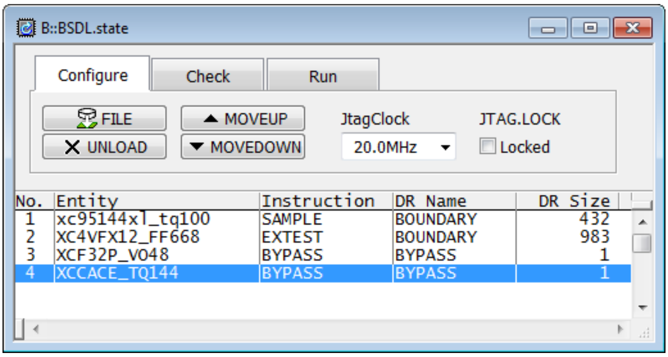
See also

- ■ BSDL
- ■ BSDL.state

Format: **BSDL.state**

The command **BSDL.state** opens the boundary scan chain configuration dialog. The entity, which is closest to the TDO has the number one in the list, the entity with the highest number is connected to the TDI.

A double-click on a list entry will open the settings/result window for this entry.



20.0MHz

□ Locked

No.	Entity	Instruction	DR Name	DR Size
1	xc95144xl_tq100	SAMPLE	BOUNDARY	432
2	XC4VFX12_FF668	EXTEST	BOUNDARY	983
3	XCF32P_VO48	BYPASS	BYPASS	1
4	XCCACE_TQ144	BYPASS	BYPASS	1

The list shows the entity name (taken from the corresponding BSDL file), the current instruction and the corresponding data register name and size for each entity in the boundary scan chain. If an instruction is changed, its name and the corresponding data register will change its color. As soon as the changes are applied to boundary scan chain (**BSDL.RUN IR** / **BSDL.RUN DR**), they will change their color to normal.

Configure (Chain configuration):

- **FILE**: Load a BSDL file and place it on the current position in the boundary scan chain
- **MOVEUP**, **MOVEDOWN**: move the selected entity up or down in the boundary scan chain
- **UNLOAD**: remove the selected entity from the boundary scan chain

Run:

- **RUN IR**: the instruction register settings will be applied to the boundary scan chain.
- **RUN DR**: the data register setting will be applied to the boundary scan chain. The read data can be viewed for each entity by opening the set/result window (double click on list entry)
- **RUN**: a instruction and data register shift will be executed (same as “RUN IR” + “RUN DR”)

Checks:

- **BYPASSall**: BYPASS mode for all entities will be set and tested, the result is shown right to this button
- **IDCODEall**: IDCODE mode is set for all entities (if defined in the BSDL file) and tested, the result is shown right to this button
- **SAMPLEall**: SAMPLE mode for all entities will be set and tested, the results for each entity can be viewed in the set/result window (double click on list entry)

See also

■ BSDL
■ BSDL.BYPASSall
■ BSDL.CHECK
■ BSDL.FILE
■ BSDL.HARDRESET
■ BSDL.IDCODEall
■ BSDL.LINKAGE
■ BSDL.MOVEDOWN
■ BSDL.MOVEUP
■ BSDL.RUN
■ BSDL.SAMPLEall
■ BSDL.SELect
■ BSDL.SET
■ BSDL.SetAndRun
■ BSDL.SOFTRESET
■ BSDL.TwoStepDR
■ BSDL.UNLOAD
❏ BSDL.GetDRBit()
❏ BSDL.GetPortLevel()

▲ 'Configuration of the Boundary Scan Chain' in 'Boundary Scan User's Guide'
▲ 'Boundary Scan Description Language (BSDL) Functions' in 'General Function Reference'

BSDL.StepPauseDR Special DR shift

Format: **BSDL.StepPauseDR** [**ON** | **OFF**]

Default: OFF.

Enables or disables the step through PauseDR for the boundary scan chain. If enabled, each DR-SCAN will step through PauseDR and Exit2DR state.

See also

■ BSDL

BSDL.SToreDR

Store data register to file

Format:	**BSDL.SToreDR** *<chip_number> <register_name> <file>* [**/***<option>*]
<option>:	**ASCII** **BINary**

Stores the data register *<register_name>* to *<file>*.

<chip_number>	Number of IC in the boundary scan chain, if the boundary scan chain has only one IC, this parameter can be omitted.
<register_name>	Data register name (must be defined in BSDL file).
<file>	File for register data
ASCII	File format is ASCII.
BINary (default)	File format is binary.

See also

- ■ BSDL
- ■ BSDL.LoadDR

BSDL.TwoStepDR — Single/double data register shift

Format: **BSDL.TwoStepDR ON | OFF**

Enables or disables double data register shift execution. When enabled, each **BSDL.RUN DR** command will execute 2 data register shifts (**BSDL.RUN** will execute 1 instruction register shift and 2 data register shifts).

This option is useful in interactive connection test, when 1 device acts as a signal driver and another as a signal receiver.

ON	With **TwoStepDR** enabled, the modified data register of the driver is shifted twice and the effect on the receiver could be observed immediately.
OFF	Without **TwoStepDR** mode, the modified data register of the driver would be shifted in and the data register from the previous cycle would be shifted out. To see the modified signal from the driver on the receiver, a second **BSDL.RUN DR** is required.

See also

- ■ BSDL
- ■ BSDL.state

BSDL.UNLOAD — Unload a chip from chain

Format: **BSDL.UNLOAD** *<chip_number>* | **ALL**

Removes one or all chips from the boundary scan chain configuration.

<chip_number>	The *<chip_number>* is removed from the configuration.
ALL	All chips are removed from the configuration.

See also

- ■ BSDL
- ■ BSDL.state

BTrace   Script-controlled trace sink

BTrace allows to add trace information to TRACE32 PowerView using PRACTICE commands. This trace information can then be displayed using the **BTrace.*** windows. The trace memory is reserved on the host running TRACE32 PowerView.

The chapter **“BTrace-specific Trace Commands”**, page 136 describes the BTrace-specific commands. While the chapter **“Generic BTrace Trace Commands”**, page 140 lists the BTrace analysis and display commands, which are generic for all TRACE32 trace methods.

BTrace-specific Trace Commands

BTrace.<specific_cmds> Overview of BTrace-specific commands

BTrace.Mode Set the trace operation mode

Format:	*<trace>*.**Mode** [*<mode>*]
<mode>:	**Fifo** | **Stack**

Selects the trace operation mode.

Fifo	If the trace is full, new records will overwrite older records. The trace records always the last cycles before the break.
Stack	If the trace is full recording will be stopped. The trace always records the first cycles after starting the trace.

BTrace.PUSH Push trace data

Format:	**BTrace.PUSH** *<data> <data>*
<cycle> *<data>:*	 **Read** *<address> <value> <time>* **Write** *<address> <value> <time>* **EXECUTE** *<address> <value> <time>* **STATistic** *<address> <count> <time> <mintime> <maxtime>* **STATisticROOT** *<time>*

Adds trace records to **BTrace**.

Definition of the *<cycle>* Parameter

Read	Memory read access with data value.
Write	Memory write access with data value.
EXECUTE	Program execution.
STATistic	Run-time statistic.
STATisticROOT	Total execution time (root).

Definition of the *<data>* Parameters

<address>: Address of the added trace record.

<value>: Represents the data access value for cycles **Read** and **Write**.
Represents the opcode for cycle **EXECUTE**.

<count>: Number of calls. This value is displayed under **count** in the **BTrace.STATistic** windows.
Cycles **STATistic** and **STATisticROOT** only.

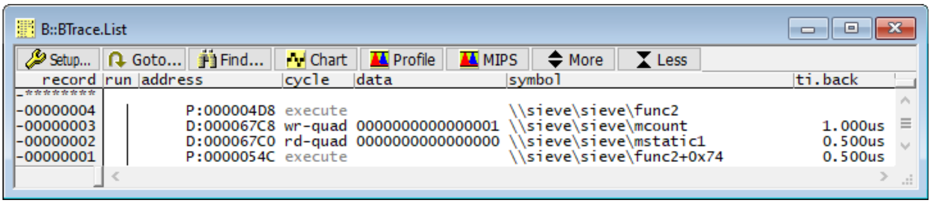
<time>: Timestamp of the trace record for cycles **Read**, **Write** and **EXECUTE**.
Total time for cycles **STATistic** and **STATisticROOT**.

<mintime>: Shortest execution time. This value is displayed under **min** in the **BTrace.STATistic** windows.
Cycles **STATistic** and **STATisticROOT** only.

<maxtime>: Longest execution time. This value is displayed under **max** in the **BTrace.STATistic** windows.
Cycles **STATistic** and **STATisticROOT** only.

Example 1: Read, Write and EXECUTE cycles

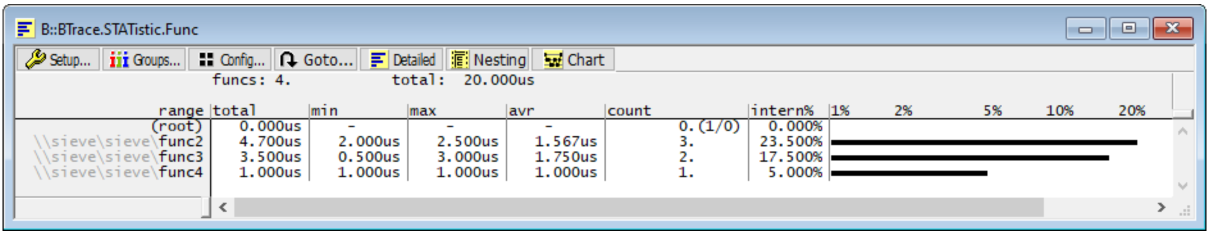
```
BTrace.RESet
BTrace.SIZE 1000.
BTrace.Arm
BTrace.PUSH EXECUTE func2 0xB590 1us
BTrace.PUSH Write mcount 1 2.us
BTrace.PUSH Read mstatic1 0 2.5us
BTrace.PUSH EXECUTE sYmbol.EXIT(func2) 0x4700 3us
BTrace.OFF
BTrace.List
```



```
   record run address        cycle   data             symbol                   ti.back
-********
-00000004        P:000004D8 execute                  \\sieve\sieve\func2
-00000003        D:000067C8 wr-quad 0000000000000001 \\sieve\sieve\mcount          1.000us
-00000002        D:000067C0 rd-quad 0000000000000000 \\sieve\sieve\mstatic1        0.500us
-00000001        P:0000054C execute                  \\sieve\sieve\func2+0x74      0.500us
```

Example 2: STATIStic and STATisticROOT cycles

```
BTrace.RESet
BTrace.SIZE 1000.
BTrace.OFF
BTrace.PUSH STATistic func2 3. 4.7us 2.us 2.5us
BTrace.PUSH STATistic func3 2. 3.5us 0.5us 3us
BTrace.PUSH STATistic func4 1. 1.us 1us 1us
BTrace.PUSH STATisticROOT 20.us
BTrace.STATistic.Func
```

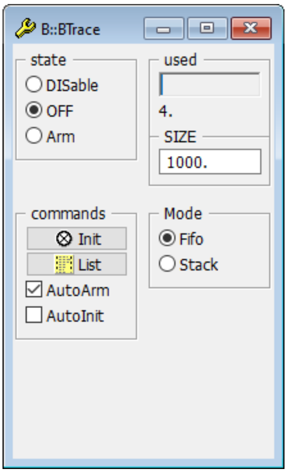


```
                       funcs: 4.          total:  20.000us

                 range total      min        max        avr        count          intern% 1%   2%    5%   10%   20%
                (root)   0.000us      -          -          -            0.(1/0)   0.000%
 \\sieve\sieve\func2     4.700us    2.000us    2.500us    1.567us       3.         23.500%
 \\sieve\sieve\func3     3.500us    0.500us    3.000us    1.750us       2.         17.500%
 \\sieve\sieve\func4     1.000us    1.000us    1.000us    1.000us       1.          5.000%
```

An example for RH850 using the BTrace and BenchMark Counters (BMC) can be found in TRACE32 system directory under `~~/demo/rh850/etc/runtime_measurement/runtime.cmm`

Format: **BTrace.state**

Displays the **BTrace.state** window, where you can configure the **BTrace**.



state
○ DISable
● OFF
○ Arm

used
4.

SIZE
1000.

Mode
● Fifo
○ Stack

☑ AutoArm
□ AutoInit

Generic BTrace Trace Commands

BTrace.Arm — Arm the trace

See command **<trace>.Arm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 133).

BTrace.AutoArm — Arm automatically

See command **<trace>.AutoArm** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 134).

BTrace.AutoInit — Automatic initialization

See command **<trace>.AutoInit** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 139).

BTrace.BookMark — Set a bookmark in trace listing

See command **<trace>.BookMark** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 140).

BTrace.Chart — Display trace contents graphically

See command **<trace>.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 143).

BTrace.ComPare — Compare trace contents

See command **<trace>.ComPare** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 191).

BTrace.DISable — Disable the trace

See command **<trace>.DISable** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 196).

BTrace.DRAW — Plot trace data against time

See command **<trace>.DRAW** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 200).

BTrace.EXPORT — Export trace data for processing in other applications

See command **<trace>.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 211).

BTrace.FILE — Load a file into the file trace buffer

See command **<trace>.FILE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 232).

BTrace.Find — Find specified entry in trace

See command **<trace>.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 234).

BTrace.FindAll — Find all specified entries in trace

See command **<trace>.FindAll** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 236).

BTrace.FindChange — Search for changes in trace flow

See command **<trace>.FindChange** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 237).

BTrace.GOTO — Move cursor to specified trace record

See command **<trace>.GOTO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 243).

BTrace.Init Initialize trace

See command **<trace>.Init** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 245).

BTrace.List List trace contents

See command **<trace>.List** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 247).

BTrace.ListNesting Analyze function nesting

See command **<trace>.ListNesting** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 262).

BTrace.LOAD Load trace file for offline processing

See command **<trace>.LOAD** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 269).

BTrace.OFF Switch off

See command **<trace>.OFF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 277).

BTrace.PROfileChart Profile charts

See command **<trace>.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 283).

BTrace.PROTOcol Protocol analysis

See command **<trace>.PROTOcol** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

BTrace.PROTOcol.Chart Graphic display for user-defined protocol

See command **<trace>.PROTOcol.Chart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 339).

BTrace.PROTOcol.Draw Graphic display for user-defined protocol

See command **<trace>.PROTOcol.Draw** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 341).

BTrace.PROTOcol.EXPORT Export trace buffer for user-defined protocol

See command **<trace>.PROTOcol.EXPORT** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 342).

BTrace.PROTOcol.Find Find in trace buffer for user-defined protocol

See command **<trace>.PROTOcol.Find** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 343).

BTrace.PROTOcol.List Display trace buffer for user-defined protocol

See command **<trace>.PROTOcol.List** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 344).

BTrace.PROTOcol.PROfileChart Profile chart for user-defined protocol

See command **<trace>.PROTOcol.PROfileChart** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 347).

BTrace.PROTOcol.PROfileSTATistic Profile chart for user-defined protocol

See command **<trace>.PROTOcol.PROfileSTATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 348).

BTrace.PROTOcol.STATistic Display statistics for user-defined protocol

See command **<trace>.PROTOcol.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 350).

BTrace.REF Set reference point for time measurement

See command **<trace>.REF** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

BTrace.RESet Reset command

See command **<trace>.RESet** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 357).

BTrace.SAVE Save trace for postprocessing in TRACE32

See command **<trace>.SAVE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 358).

BTrace.SIZE Define buffer size

See command **<trace>.SIZE** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 372).

BTrace.STATistic Statistic analysis

See command **<trace>.STATistic** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 377).

BTrace.Timing Waveform of trace buffer

See command **<trace>.Timing** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 496).

BTrace.TRACK Set tracking record

See command **<trace>.TRACK** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 499).

BTrace.View Display single record

See command **<trace>.View** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 501).

BTrace.ZERO Align timestamps of trace and timing analyzers

See command **<trace>.ZERO** in 'General Commands Reference Guide T' (general_ref_t.pdf, page 502).