

# Training - Debugger Basics

Release 09.2022

**MANUAL**

# Training - Debugger Basics

---

**TRACE32 Online Help**

**TRACE32 Directory**

**TRACE32 Index**

<b>TRACE32 Training</b> .....		
<b>Debugger Training</b> .....		
<b>Training - Debugger Basics</b> .....		<b>1</b>
<b>System Concept</b> .....		<b>6</b>
On-chip Debug Interface		7
Debug Features		7
TRACE32 Tools		8
On-chip Debug Interface plus On-chip Trace Buffer		10
On-chip Debug Interface plus Trace Port		12
NEXUS Interface		13
<b>Starting a TRACE32 PowerView Instance</b> .....		<b>14</b>
Basic TRACE32 PowerView Parameters		14
Configuration File		14
Standard Parameters		15
Examples for Configuration Files		16
Additional Parameters		20
Application Properties (Windows only)		21
Configuration via T32Start (Windows only)		22
About TRACE32		23
Version Information		23
Prepare Full Information for a Support Email		24
<b>Establish your Debug Session</b> .....		<b>25</b>
<b>TRACE32 PowerView</b> .....		<b>26</b>
TRACE32 PowerView Components		26
Main Menu Bar and Accelerators		27
Main Tool Bar		29
Window Area		31
Command Line		34
Message Line		37
Softkeys		38
State Line		39
<b>Registers</b> .....		<b>40</b>
Core Registers		40

Display the Core Registers	40
Colored Display of Changed Registers	41
Modify the Contents of a Core Register	42
Special Function Register	43
Display the Special Function Registers	43
Details about a Single Special Function Register	46
Modify a Special Function Register	47
The PER Definition File	48
<b>Memory Display and Modification</b>	<b>49</b>
The Data.dump Window	50
Display the Memory Contents	50
Modify the Memory Contents	55
Run-time Memory Access	56
Colored Display of Changed Memory Contents	66
The List Window	67
Displays the Source Listing Around the PC	67
Displays the Source Listing of a Selected Function	68
<b>Breakpoints</b>	<b>70</b>
Breakpoint Implementations	70
Software Breakpoints in RAM	70
Software Breakpoints in FLASH	72
Onchip Breakpoints in NOR Flash	73
Onchip Breakpoints on Read/Write Accesses	76
Onchip Breakpoints by Processor Architecture	77
ETM Breakpoints for ARM or Cortex-A/-R	78
Breakpoint Types	81
Program Breakpoints	82
Read/Write Breakpoints	84
<b>Breakpoint Handling</b>	<b>86</b>
Breakpoint Setting at Run-time	86
Real-time Breakpoints vs. Intrusive Breakpoints	87
Break.Set Dialog Box	89
The HLL Check Box - Function Name	90
The HLL Check Box - Program Line Number	93
The HLL Check Box - Variable	94
The HLL Check Box - HLL Expression	96
Implementations	99
Actions	100
Options	104
DATA Breakpoints	108
Advanced Breakpoints	112
TASK-aware Breakpoints	113

Intrusive TASK-aware Breakpoint	113
Real-time TASK-aware Breakpoint	116
COUNTer	117
Software Counter	117
On-chip Counter	120
CONDITION	121
CMD	129
memory/register/var	132
Display a List of all Set Breakpoints	137
Delete Breakpoints	138
Enable/Disable Breakpoints	138
Store Breakpoint Settings	139
<b>Debugging .....</b>	<b>140</b>
Debugging of Optimized Code	140
Basic Debug Control	143
<b>Sample-based Profiling .....</b>	<b>155</b>
Program Counter Sampling	155
Standard Procedure	156
Details	160
TASK Sampling	162

# Training - Debugger Basics

---

Version 26-Oct-2022

# System Concept

---

A single-core processor/multi-core chip can provide:

- An on-chip debug interface
- An on-chip debug interface plus an on-chip trace buffer
- An on-chip debug interface plus an off-chip trace port
- A NEXUS interface including an on-chip debug interface

Depending on the debug resources different debug features can be provided and different TRACE32 tools are offered.

# On-chip Debug Interface

---

The TRACE32 debugger allows you to test your embedded hardware and software by using the on-chip debug interface. The most common on-chip debug interface is JTAG.

A single on-chip debug interface can be used to debug all cores of a multi-core chip.

## Debug Features

---

Depending on the processor architecture different debug features are available.

### **Debug features provided by all processor architectures:**

- Read/write access to registers
- Read/write access to memories
- Start/stop of program execution

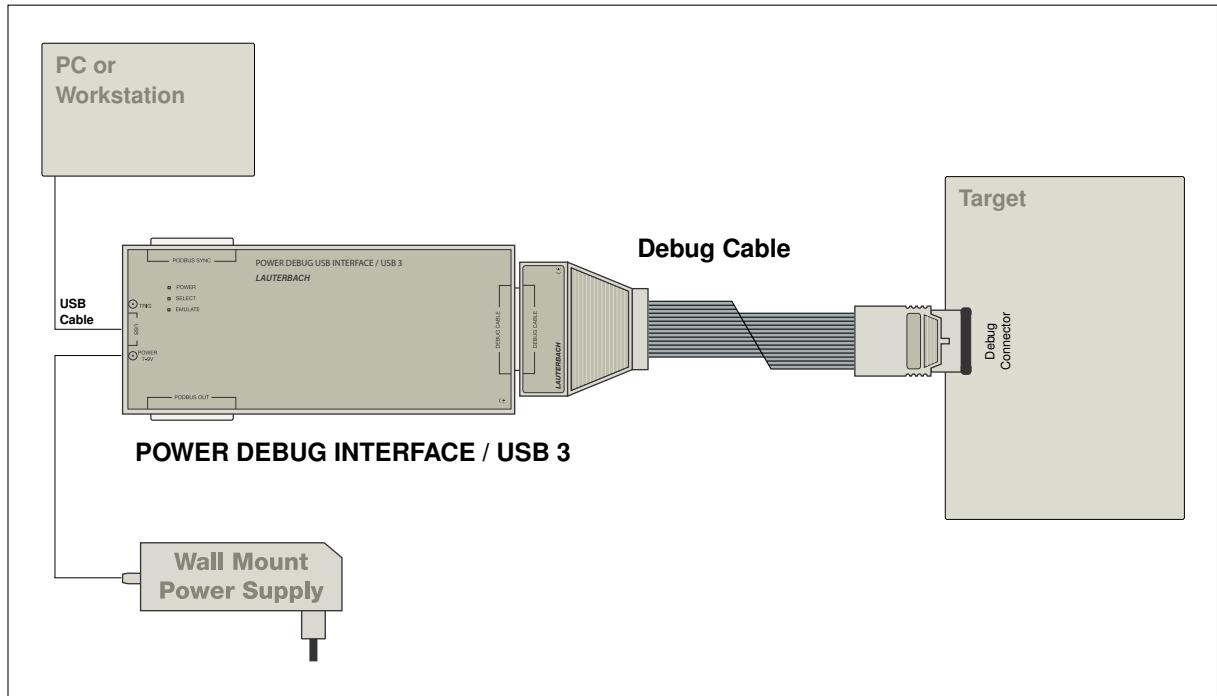
### **Debug features specific for a processor architecture:**

- Number of on-chip breakpoints
- Read/write access to memory while the program execution is running
- Additional features as benchmark counters, triggers etc.

The TRACE32 debugger hardware always consists of:

- Universal debugger hardware
- Debug cable specific to the processor architecture

## Debug Only Modules

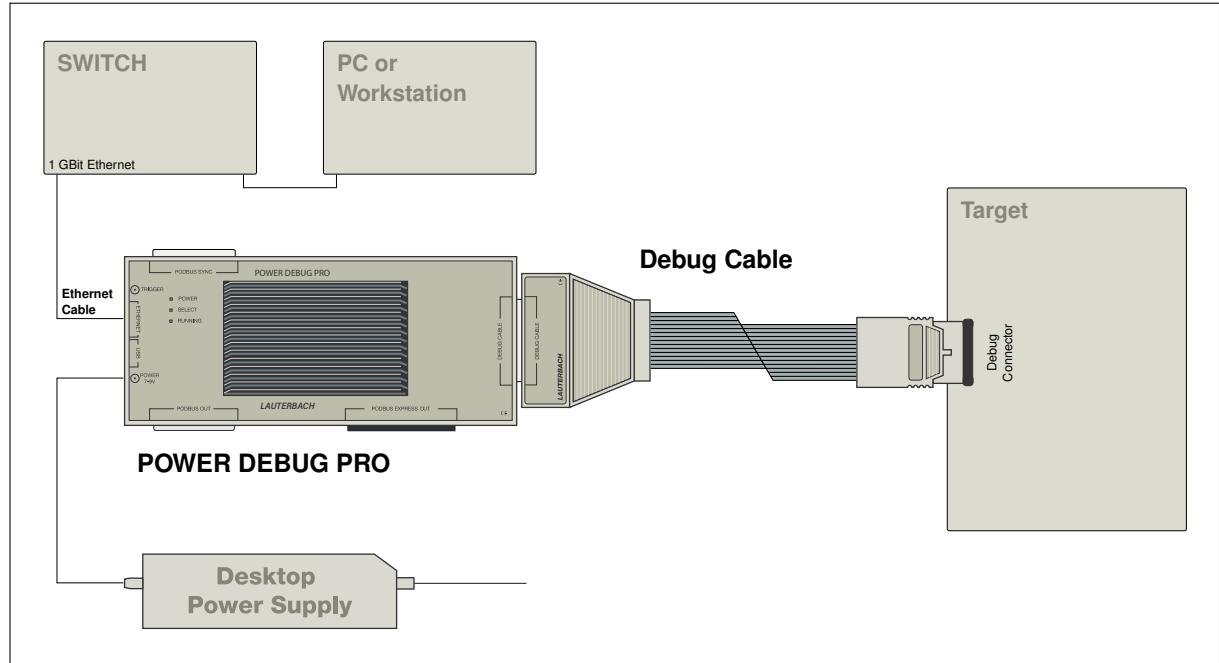


Current module:

- POWER DEBUG INTERFACE / USB 3

Deprecated module:

- POWER DEBUG INTERFACE / USB 2



Current module:

- POWER DEBUG PRO (USB 3 and 1 GBit Ethernet)

Deprecated modules:

- POWER DEBUG II (USB 2 and 1 GBit Ethernet)
- POWER DEBUG / ETHERNET (USB 2 and 100 MBit Ethernet)

# On-chip Debug Interface plus On-chip Trace Buffer

A number of single-core processors/multi-core chips offer in addition to the on-chip debug interface an on-chip trace buffer.

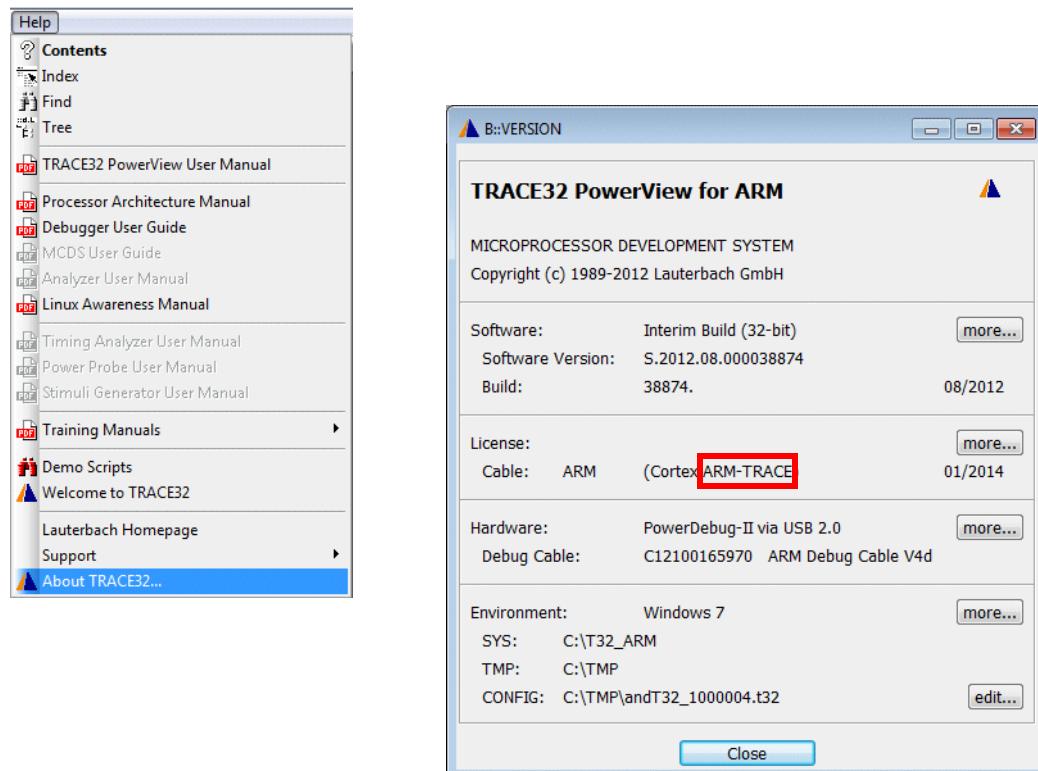
## On-chip Trace Features

The on-chip trace buffer can store information:

- On the executed instructions.
- On task/process switches.
- On load/store operations if supported by the on-chip trace generation hardware.

In order to analyze and display the trace information the debug cable needs to provide a **Trace License**. The Trace Licenses use the following name convention:

- <core>-TRACE e.g. ARM-TRACE
- or <core>-MCDS) e.g. TriCore-MCDS



The display and the evaluation of the trace information is described in the following training manuals:

- “[Training Arm ETM](#)” (training\_arm\_etm.pdf).
- “[Training Cortex-M Trace](#)” (training\_cortexm\_etm.pdf).
- “[Training AURIX Trace](#)” (training\_aurix\_trace.pdf).
- “[Training Hexagon-ETM](#)” (training\_hexagon\_etm.pdf).
- “[Training Nexus](#)” (training\_nexus.pdf).

# On-chip Debug Interface plus Trace Port

---

A number of single-core processors/multi-core chips offer in addition to the on-chip debug interface a so-called trace port. The most common trace port is the TPIU for the ARM/Cortex architecture.

## Off-chip Trace Features

---

The trace port exports in real-time trace information:

- On the executed instructions.
- On task/process switches.
- On load/store operations if supported by the on-chip trace generation logic.

The display and the evaluation of the trace information is described in the following training manuals:

- [“Training Arm ETM”](#) (training\_arm\_etm.pdf)
- [“Training Cortex-M Trace”](#) (training\_cortexm\_etm.pdf)
- [“Training AURIX Trace”](#) (training\_aurix\_trace.pdf)
- [“Training Hexagon-ETM”](#) (training\_hexagon\_etm.pdf)

NEXUS is a standardized interface for on-chip debugging and real-time trace especially for the automotive industry.

## NEXUS Features

---

### **Debug features provided by all single-core processors/multi-core chips:**

- Read/write access to the registers
- Read/write access to all memories
- Start/stop of program execution
- Read/write access to memory while the program execution is running

### **Debug features specific for single-core processor/multi-core chip:**

- Number of on-chip breakpoints
- Benchmark counters, triggers etc.

### **Trace features provided by all single-core processors/multi-core chips:**

- Information on the executed instructions.
- Information on task/process switches.

### **Trace features specific for the single-core processor/multi-core chip:**

- Information on load/store operations if supported by the trace generation logic.

The display and the evaluation of the trace information is described in “[Training Nexus](#)” (training\_nexus.pdf).

# Starting a TRACE32 PowerView Instance

---

## Basic TRACE32 PowerView Parameters

---

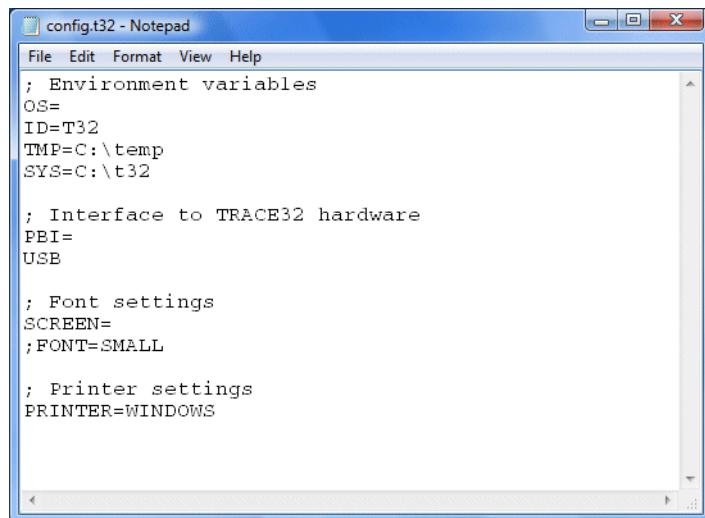
This chapter describes the basic parameters required to start a TRACE32 PowerView instance.

The parameters are defined in the configuration file. By default the configuration file is named **config.t32**. It is located in the TRACE32 system directory (parameter **SYS**).

## Configuration File

---

Open the file **config.t32** from the system directory (default c :\T32\config.t32) with any ASCII editor.



A screenshot of a Windows Notepad window titled "config.t32 - Notepad". The window contains the following configuration file text:

```
File Edit Format View Help
; Environment variables
OS=
ID=T32
TMP=C:\temp
SYS=C:\t32

; Interface to TRACE32 hardware
PBI=
USB

; Font settings
SCREEN=
;FONT=SMALL

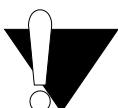
; Printer settings
PRINTER=WINDOWS
```

The following rules apply to the configuration file:

- Parameters are defined paragraph by paragraph.
- The first line/headline defines the parameter type.
- Each parameter definition ends with an empty line.
- If no parameter is defined, the default parameter will be used.

## Standard Parameters

Parameter	Syntax	Description
Host interface	PBI= <i>&lt;host_interface&gt;</i>  PBI=ICD <i>&lt;host_interface&gt;</i>	Host interface type of TRACE32 tool hardware (USB or ethernet)  Full parameter syntax which is not in use.
Environment variables	OS= ID= <i>&lt;identifier&gt;</i> TMP= <i>&lt;temp_directory&gt;</i> SYS= <i>&lt;system_directory&gt;</i> HELP= <i>&lt;help_directory&gt;</i>	(ID) Prefix for all files which are saved by the TRACE32 PowerView instance into the TMP directory  (TMP) Temporary directory used by the TRACE32 PowerView instance (*)  (SYS) System directory for all TRACE32 files  (HELP) Directory for the TRACE32 help PDFs (**)
Printer definition	PRINTER=WINDOWS	All standard Windows printer can be used from TRACE32 PowerView
License file	LICENSE= <i>&lt;license_directory&gt;</i>	Directory for the TRACE32 license file (not required for new tools)



(\*) In order to display source code information TRACE32 PowerView creates a copy of all loaded source files and saves them into the TMP directory.

(\*\*) The TRACE32 online help is PDF-based.

# Examples for Configuration Files

---

## Configuration File for USB

---

Single debugger hardware module connected via USB:

```
; Host interface
PBI=
USB

; Environment variables
OS=
ID=T32
TMP=C:\temp ; temporary directory for TRACE32
SYS=C:\t32   ; system directory for TRACE32
HELP=C:\t32\pdf ; help directory for TRACE32

; Printer settings
PRINTER=WINDOWS ; all standard windows printer can be
                 ; used from the TRACE32 user interface
```

Multiple debugger hardware modules connected via USB:

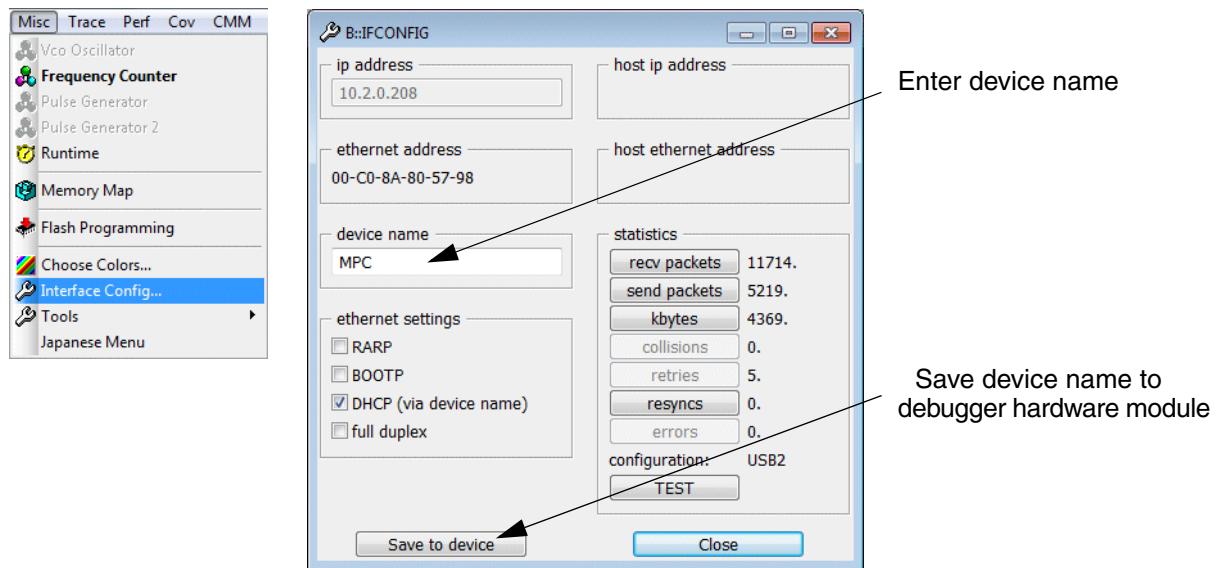
```
; Host interface
PBI=
USB
NODE=training1 ; NODE name of TRACE32

; Environment variables
OS=
ID=T32_training1
TMP=C:\temp ; temporary directory for TRACE32
SYS=C:\t32   ; system directory for TRACE32
HELP=C:\t32\pdf ; help directory for TRACE32

; Printer settings
PRINTER=WINDOWS ; all standard windows printer can be
                 ; used from TRACE32 PowerView
```

Use the IFCONFIG command to assign a device name (NODE=) to a debugger hardware module. The manufacturing default device name is the serial number of the debugger hardware module:

- e.g. E18110012345 for a debugger hardware module with ethernet interface, such as PowerDebug PRO.
- e.g. C18110045678 for a debugger hardware module with USB interface only, such as PowerDebug USB 3.



## IFCONFIG

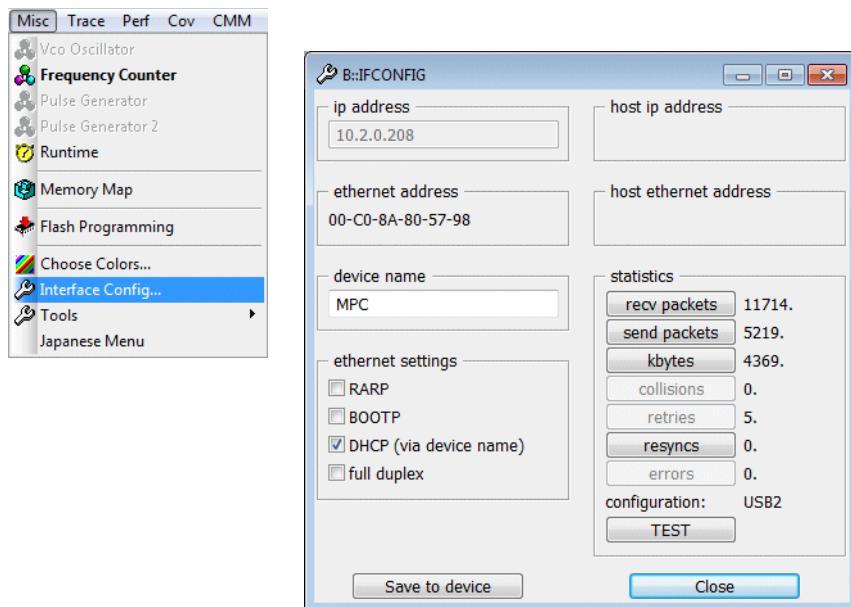
Dialog to assign USB device name

Please be aware that USB device names are case-sensitive

TRACE32 allows to communicate with a POWER DEBUG INTERFACE USB from a remote PC. For an example, see “[Example: Remote Control for POWER DEBUG INTERFACE / USB](#)” in TRACE32 Installation Guide, page 56 (installation.pdf).

```
; Host interface  
PBI=  
NET  
NODE=training1  
  
; Environment variables  
OS=  
ID=T32 ; temp directory for TRACE32  
SYS=C:\t32 ; system directory for TRACE32  
HELP=C:\t32\pdf ; help directory for TRACE32  
  
; Printer settings  
PRINTER=WINDOWS ; all standard windows printer can be  
; used from the TRACE32 user interface
```

## Ethernet Configuration and Operation Profile



### IFCONFIG

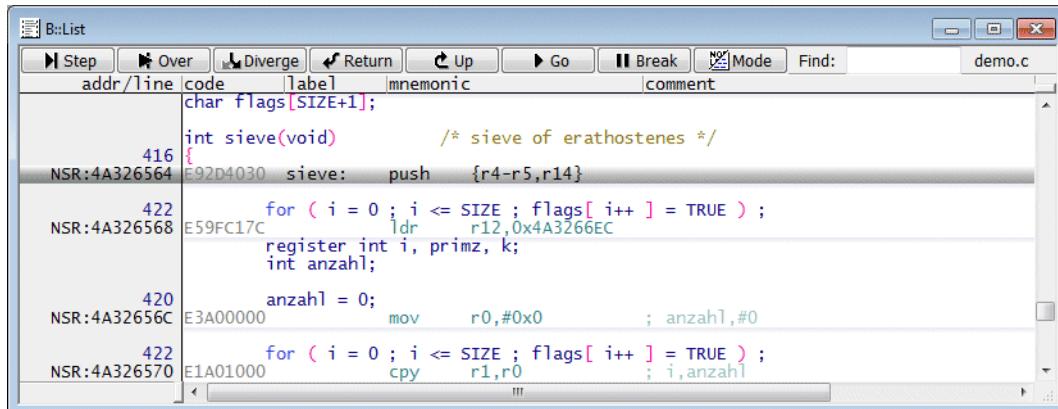
Dialog to display and change information for the Ethernet interface

## Additional Parameters

Changing the font size can be helpful for a more comfortable display of TRACE32 windows.

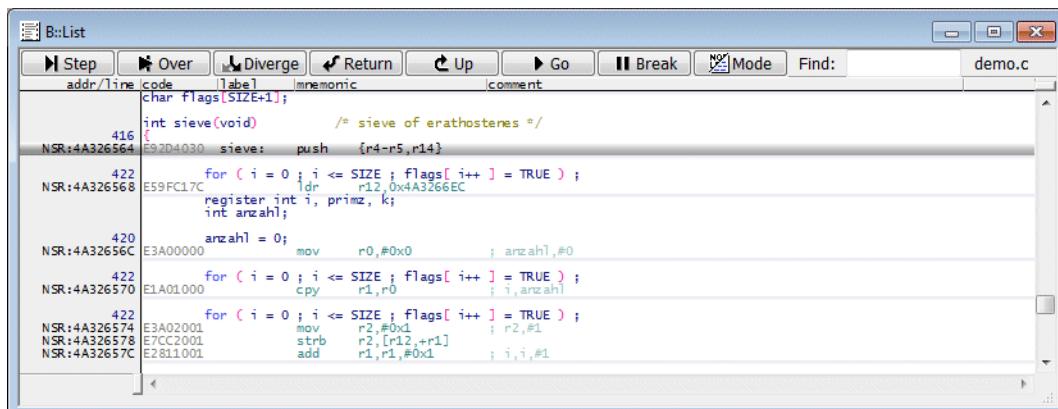
```
; Screen settings
SCREEN=
FONT=SMALL ; Use small fonts
```

### Display with normal font:



The screenshot shows the TRACE32 B::List window displaying assembly code for a sieve function. The font size is standard, making the code readable but not particularly compact. The assembly code includes instructions like `push {r4-r5,r14}` and loops with conditions like `i <= SIZE`.

### Display with small font:

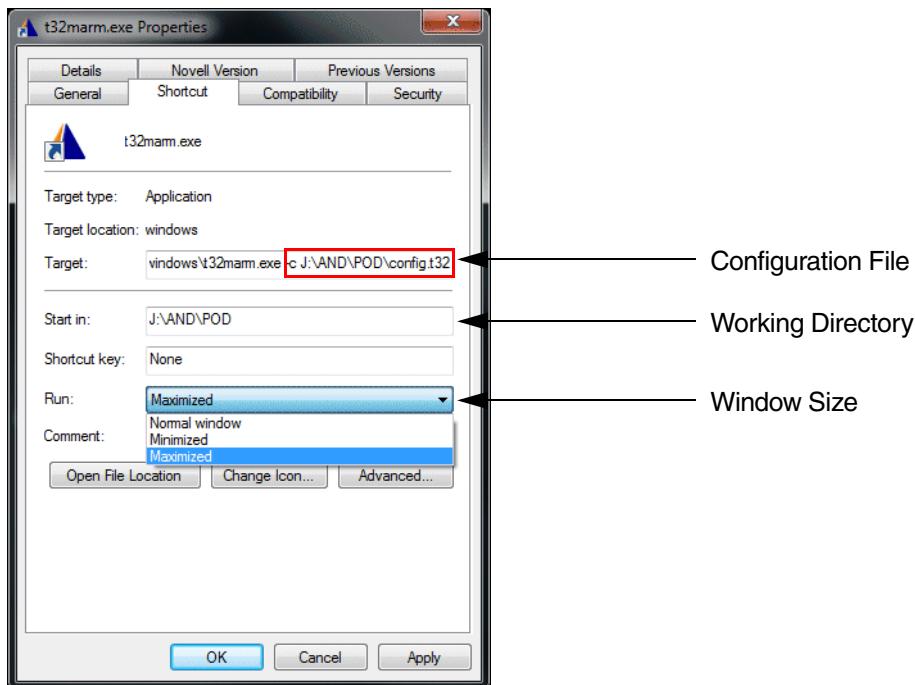


The screenshot shows the same TRACE32 B::List window as above, but with a smaller font size applied. The text is significantly smaller, fitting more lines into the same vertical space. The assembly code remains identical to the first screenshot.

# Application Properties (Windows only)

The **Properties** window allows you to configure some basic settings for the TRACE32 software.

To open the **Properties** window, right-click the desired TRACE32 icon in the **Windows Start** menu.



## Definition of the Configuration File

By default the configuration file **config.t32** in the TRACE32 system directory (parameter **SYS**) is used. The option **-c** allows you to define your own location and name for the configuration file.

```
C:\T32_ARM\bin\windows\t32marm.exe -c j:\and\config.t32
```

## Definition of a Working Directory

After its start TRACE32 PowerView is using the specified working directory. It is recommended not to work in the system directory.

**PWD**

TRACE32 command to display the current working directory

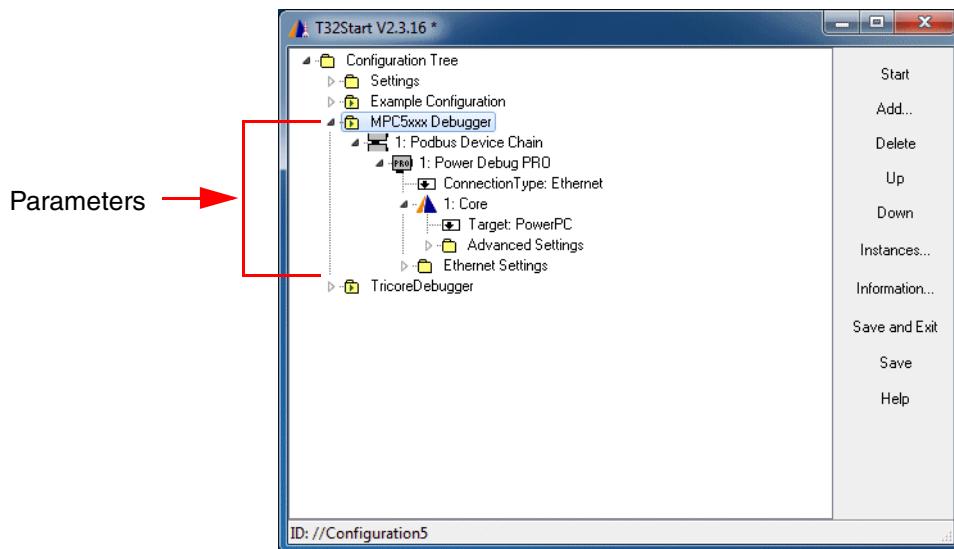
## Definition of the Window Size for TRACE32 PowerView

You can choose between Normal window, Minimized and Maximized.

# Configuration via T32Start (Windows only)

The basic parameters can also be set up in an intuitive way via **T32Start**.

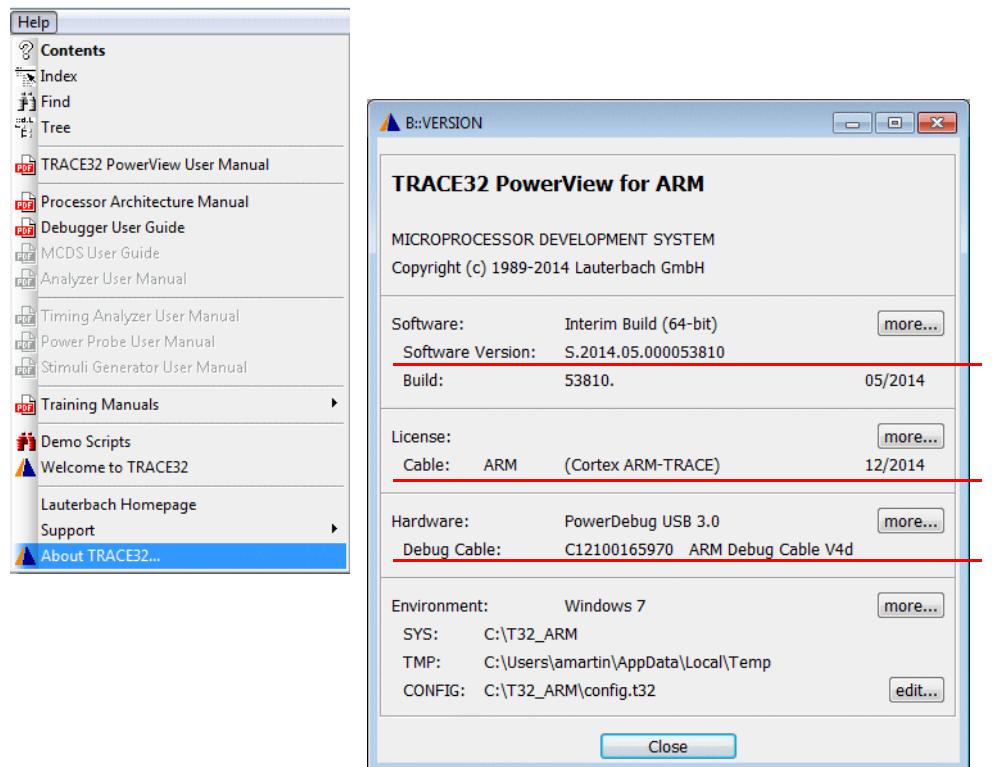
A detailed online help for **t32start.exe** is available via the **Help** button or in "**T32Start**" (app\_t32start.pdf).



# About TRACE32

If you want to contact your local Lauterbach support, it might be helpful to provide some basis information about your TRACE32 tool.

## Version Information



The VERSION window informs you about:

1. The version of the TRACE32 software.
2. The debug licenses programmed into the debug cable and the expiration date of your software warranty respectively the expiration date of your software maintenance.
3. The serial number of the debug cable.

### VERSION.view

Display the VERSION window.

### VERSION.HARDWARE

Display more details about the TRACE32 hardware modules.

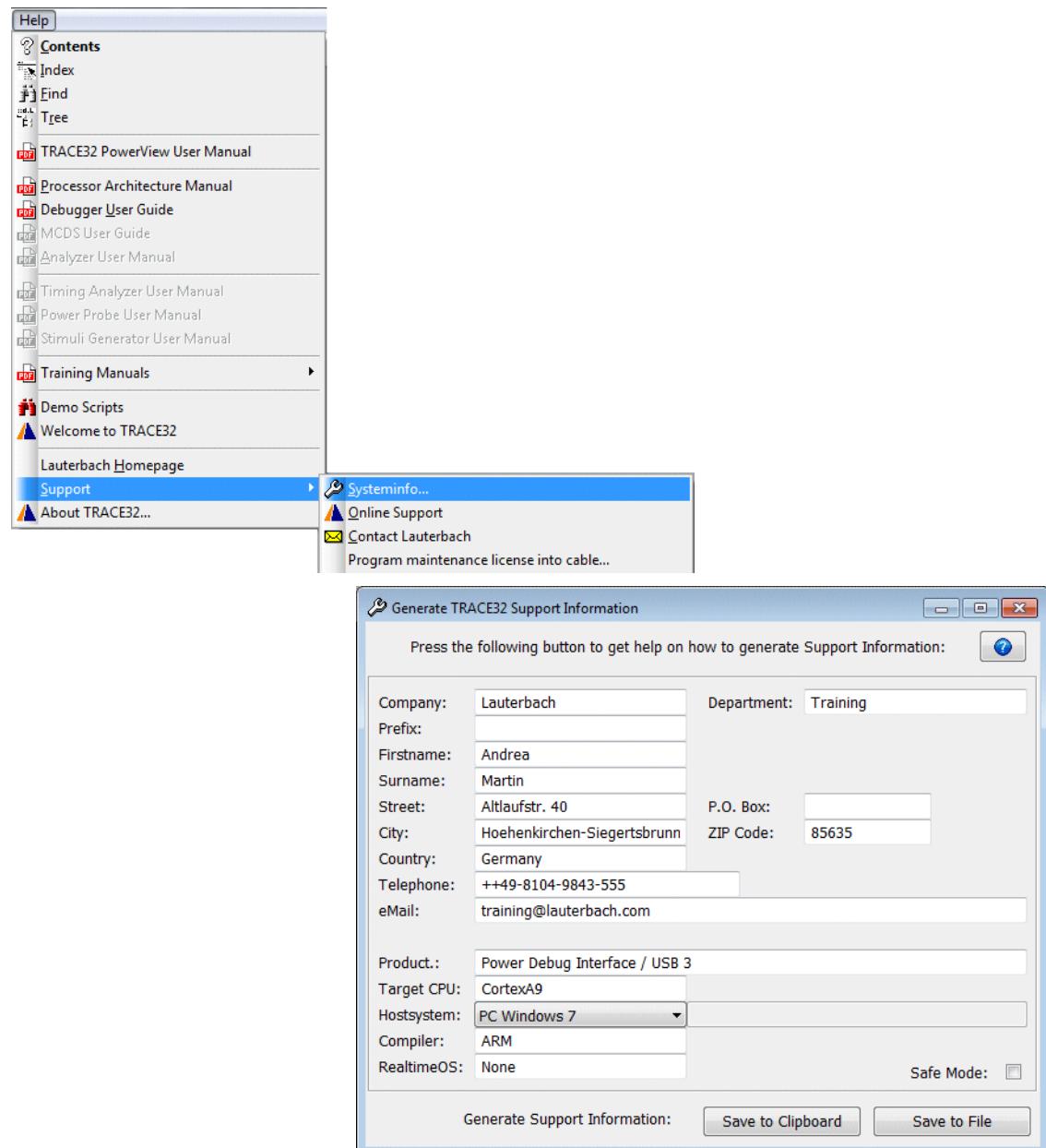
### VERSION.SOFTWARE

Display more details about the TRACE32 software.

# Prepare Full Information for a Support Email

Be sure to include detailed system information about your TRACE32 configuration.

1. To generate a system information report, choose **Help > Support > Systeminfo**.



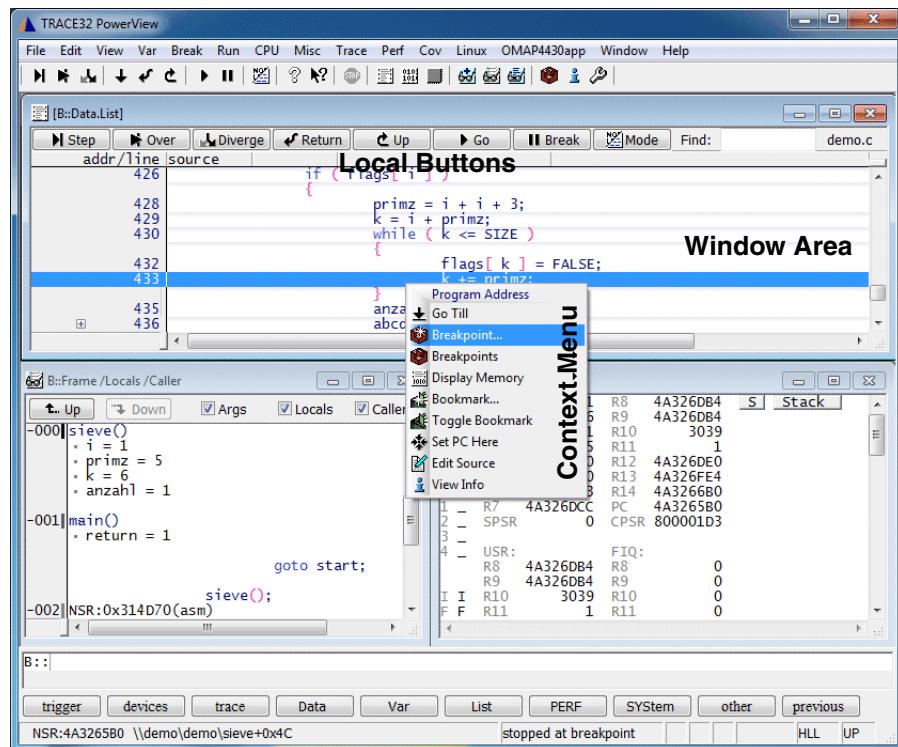
2. Preferred: click **Save to File**, and send the system information as an attachment to your e-mail.
3. Click **Save to Clipboard**, and then paste the system information into your e-mail.

# Establish your Debug Session

---

Before you can start debugging, the debug environment has to be set up. An overview on the most common setups is given in “[Establish Your Debug Session](#)” (tutor\_setup.pdf).

## TRACE32 PowerView Components



Main Menu Bar  
Main Tool Bar

Command Line  
Message Line  
SoftkeyLine  
State Line



The structure of the menu bar and the tool bar are defined by the file **t32.men** which is located in the TRACE32 system directory.

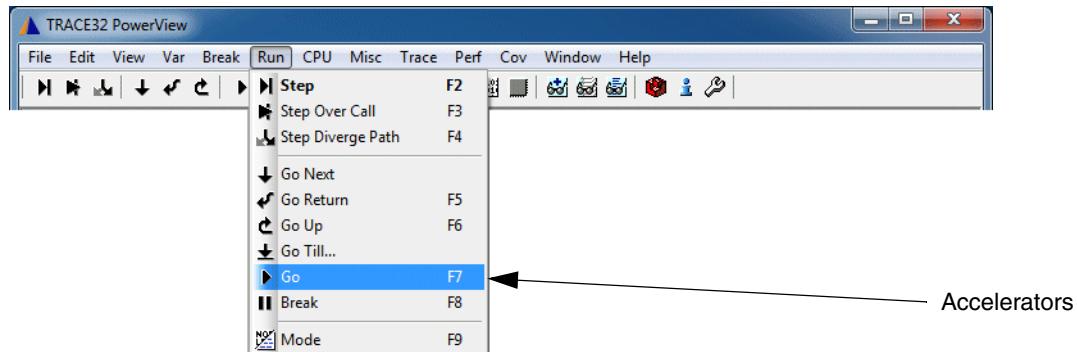
TRACE32 allows you to modify the menu bar and the tool bar so they will better fit your requirements. Refer to "[Training Menu](#)" (training\_menu.pdf) for details.

## Main Menu Bar and Accelerators

---

The main menu bar provides all important TRACE32 functions sorted by groups.

For often used commands accelerators are defined.



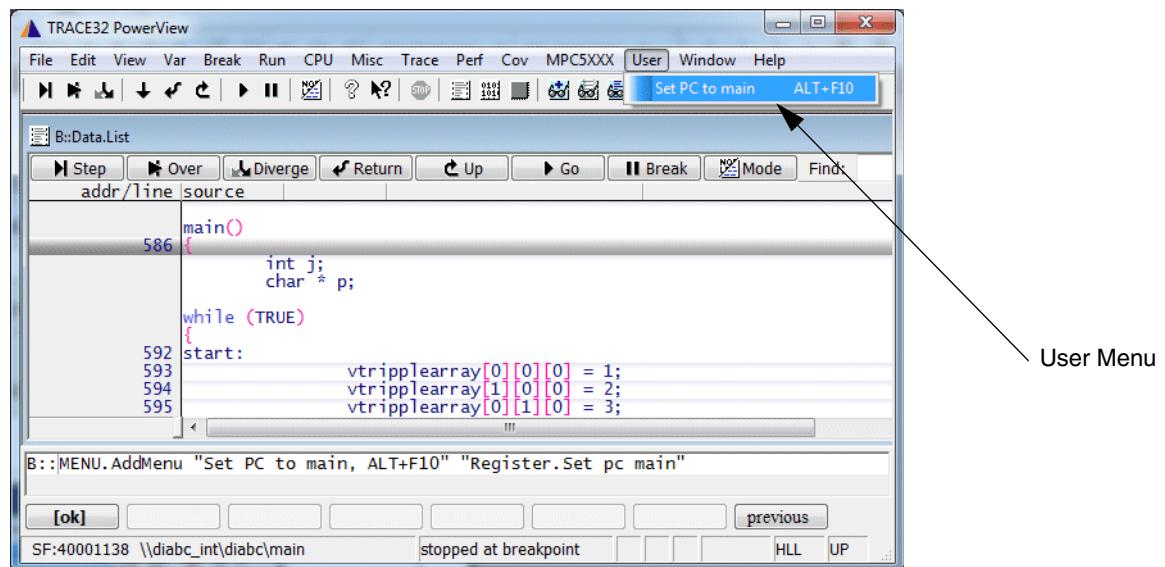
A user specific menu can be defined very easily:

**MENU.AddMenu <name> <command>** Add a user menu

**MENU.RESET** Reset menu to default

```
; user menu
MENU.AddMenu "Set PC to main" "Register.Set PC main"

; user menu with accelerator
MENU.AddMenu "Set PC to main, ALT+F10" "Register.Set PC main"
```



For more complex changes to the main menu bar refer to "[Training Menu](#)" (training\_menu.pdf).

Videos about the menu programming can be found here:  
[https://www.lauterbach.com/tut\\_customization.html](https://www.lauterbach.com/tut_customization.html)



## Main Tool Bar

The main tool bar provides fast access to often used commands.

The user can add his own buttons very easily:

**MENU.AddTool <tooltip\_text> <tool\_image> <command>**

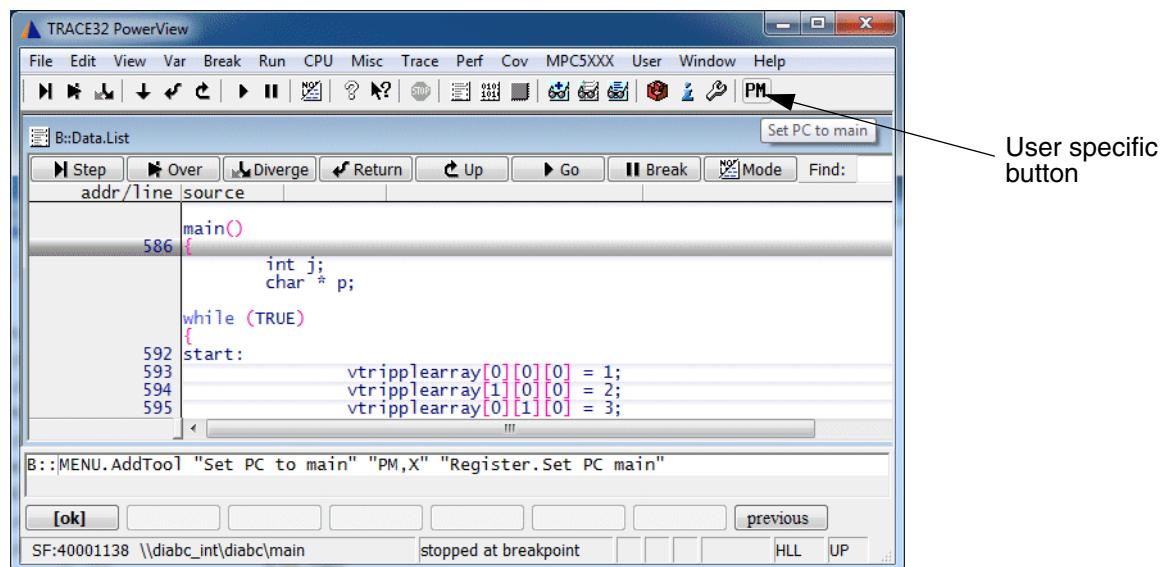
Add a button to the toolbar

**MENU.RESet**

Reset menu to default

```
; <tooltip text> here:      Set PC to main  
; <tool image> here:      button with capital letters PM in black  
; <command> here:        Register.Set PC main
```

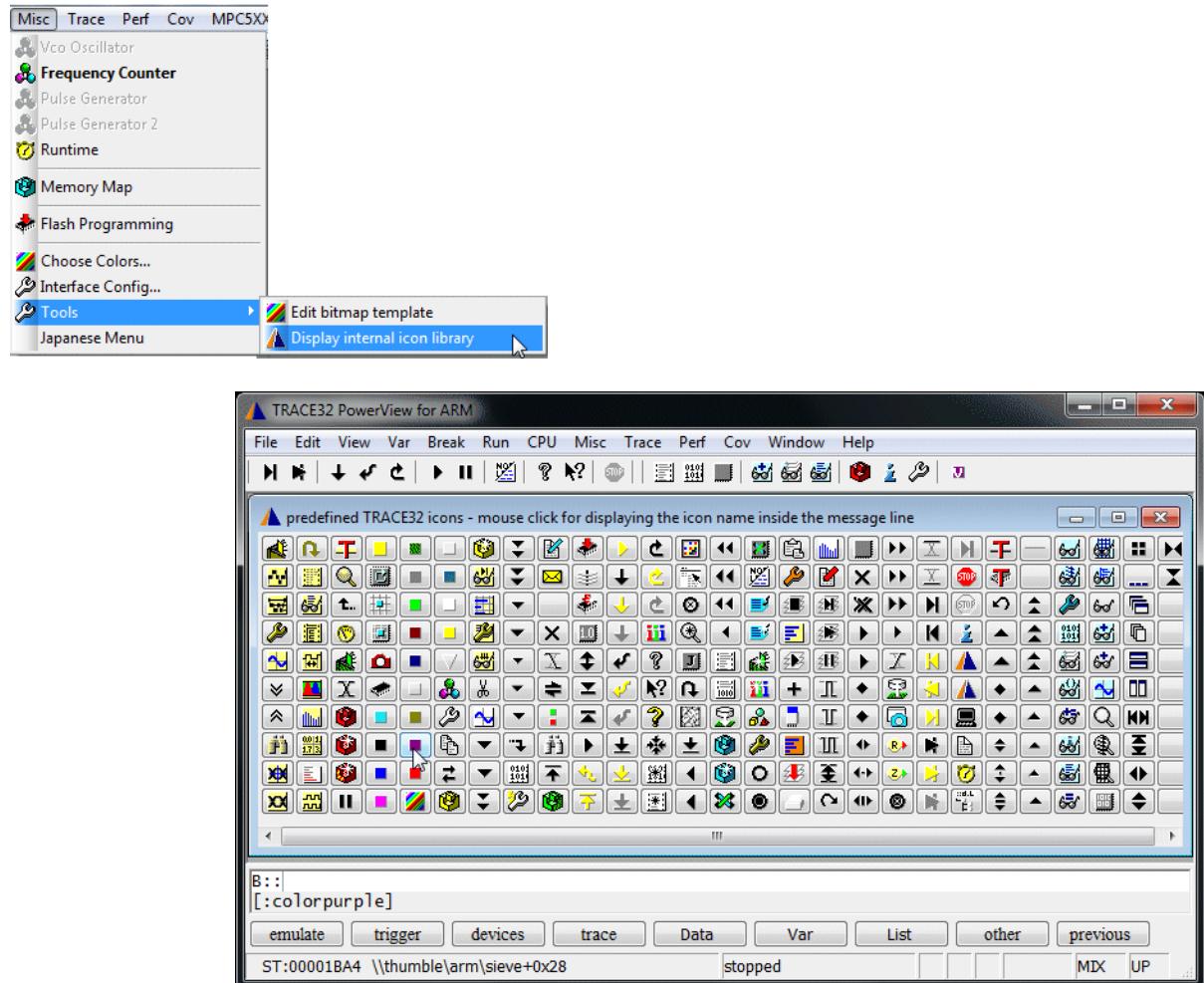
```
MENU.AddTool "Set PC to main" "PM,X" "Register.Set PC main"
```



Information on the *<tool image>* can be found in **Help -> Contents**

**TRACE32 Documents -> IDE User Interface -> PowerView Command Reference -> MENU -> Programming Commands -> TOOLITEM.**

All predefined TRACE32 icons can be inspected as follows:

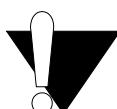


Or by following TRACE32 command:

```
ChDir.DO ~/demo/menu/internal_icons.cmm
```

The predefined icons can easily be used to create new icons.

```
; overprint the icon colorpurple with the character v in White color  
Menu.AddTool "Set PC to main" "v,W,colorpurple" "Register.Set PC main"
```



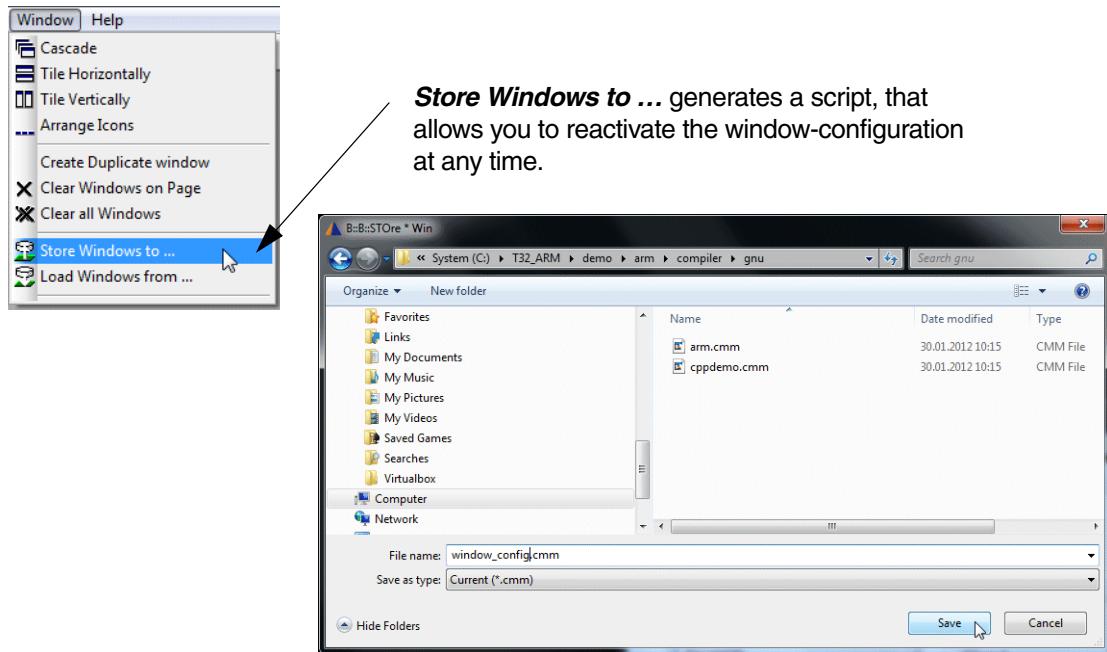
For more complex changes to the main tool bar refer to "["Training Menu"](#)" ([training\\_menu.pdf](#)).

Videos about the menu programming can be found here:  
[https://www.lauterbach.com/tut\\_customization.html](https://www.lauterbach.com/tut_customization.html)

# Window Area

## Save Page Layout

No information about the window layout is saved when you exit TRACE32 PowerView. To save the window layout use the **Store Windows to ...** command in the **Window** menu.



**Store Windows to ...** generates a script, that allows you to reactivate the window-configuration at any time.

### Script example:

```
// andT32_1000003 Sat Jul 21 16:59:55 2012

B:::

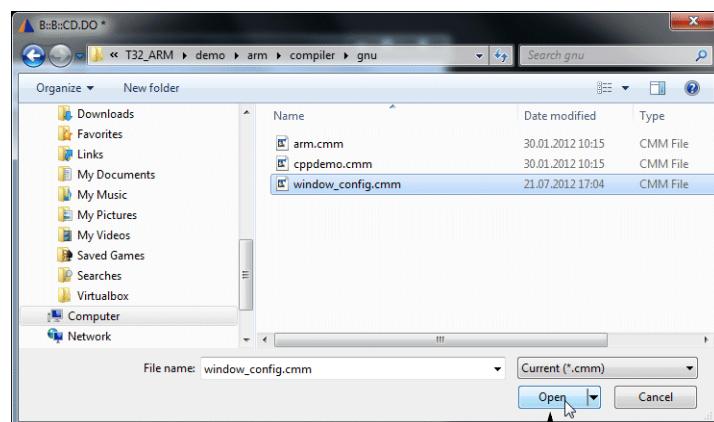
TOOLBAR ON
STATUSBAR ON
FramePOS 68.0 5.2857 107. 45.
WinPAGE.RESET

WinCLEAR
WinPOS 0.0 0.0 80. 16. 15. 1. W000
WinTABS 10. 10. 25. 62.
List

WinPOS 0.0 21.643 80. 5. 25. 1. W001
WinTABS 13. 0. 0. 0. 0. 0. 0.
Break.List

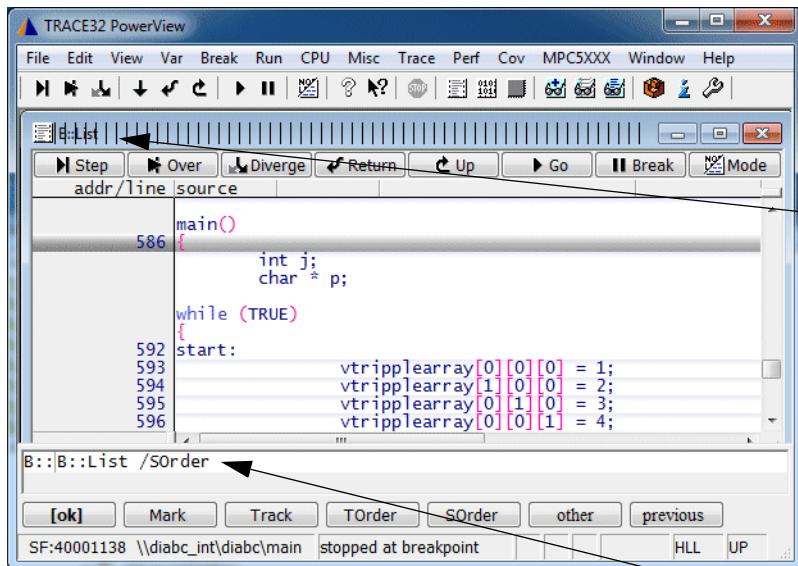
WinPAGE.select P000

ENDDO
```



Run the script to reactivate the stored  
window-configuration

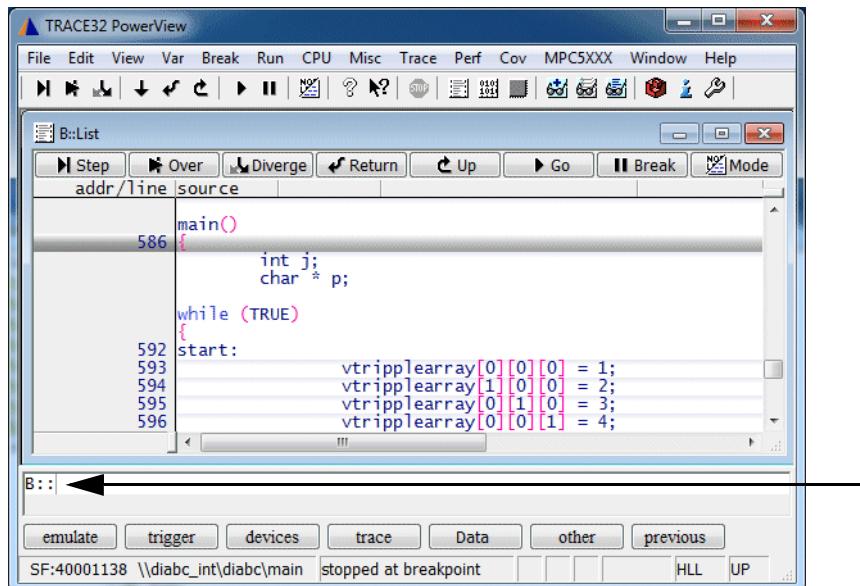
## Modify Window



The window header displays the command which was executed to open the window

By clicking with the right mouse button to the window header, the command which was executed to open the window is re-displayed in the command line and can be modified there

# Command Line

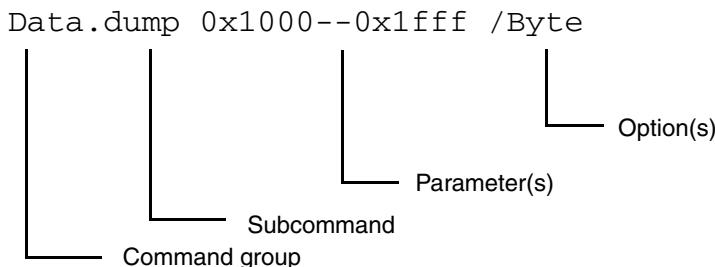


Command line

## Command Structure

**Device prompt:** the default device prompt is **B:::**. It stands for BDM which was the first on-chip debug interface supported by Lauterbach.

A TRACE32 command has the following structure:



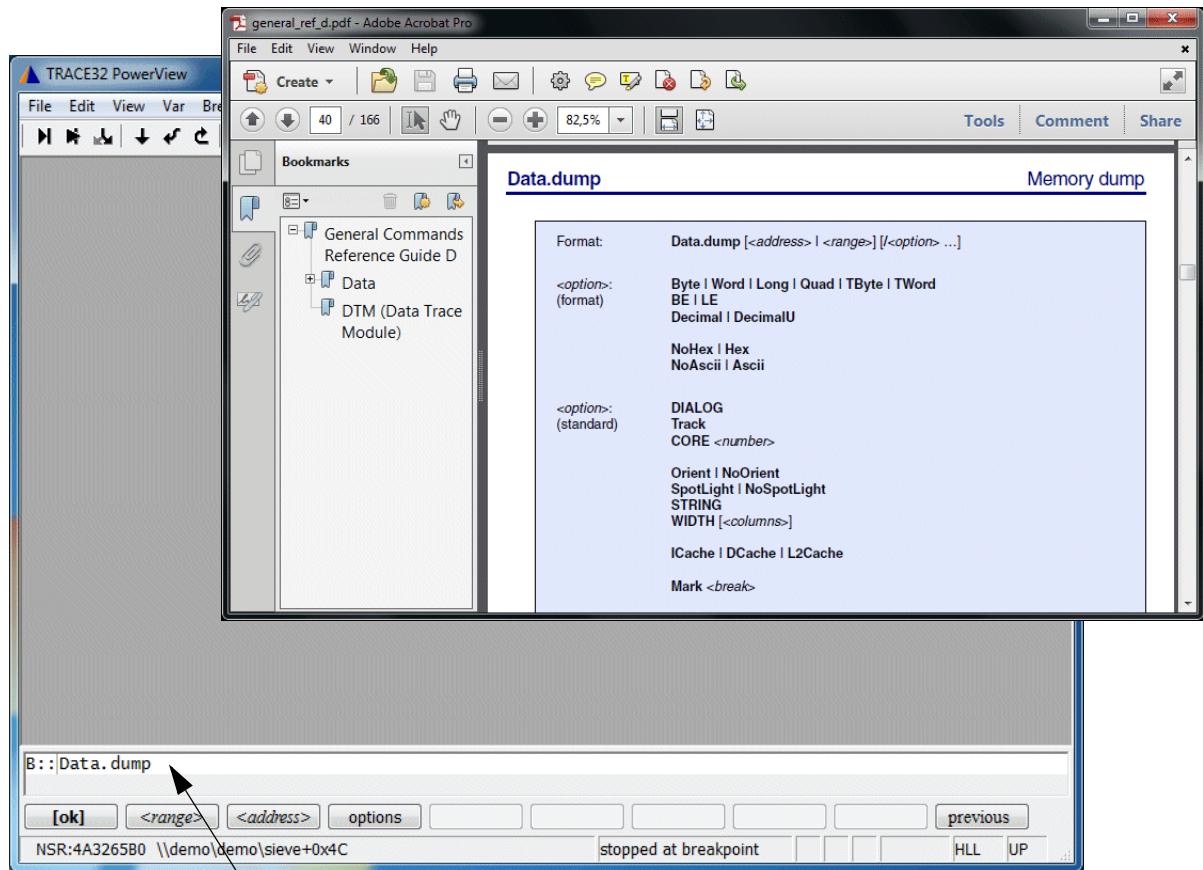
## Command Examples

<b>Data</b>	<b>Command group to display, modify ... memory</b>
Data.dump	Displays a hex dump
Data.Set	Modify memory
Data.LOAD.auto	Loads code to the target memory

<b>Break</b>	<b>Command group to set, list, delete ... breakpoints</b>
Break.Set	Sets a breakpoint
Break.List	Lists all set breakpoint
Break.Delete	Deletes a breakpoint

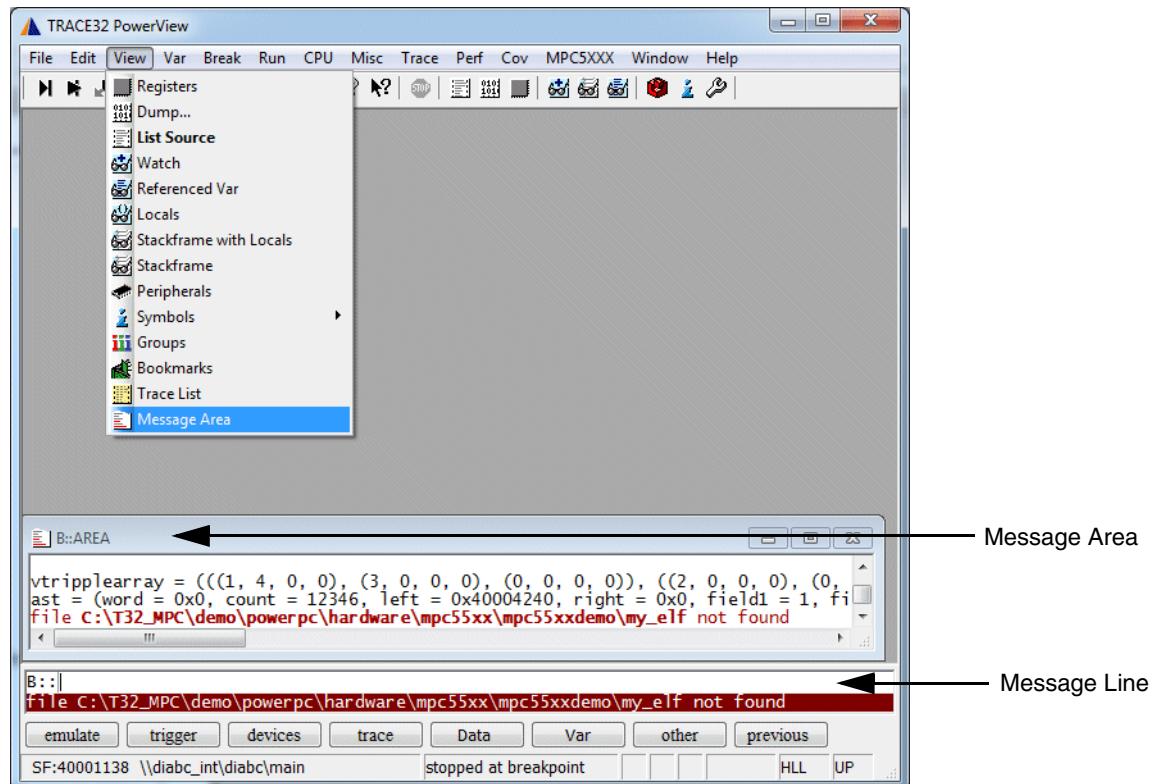
Each command can be abbreviated. The significant letters are always written in upper case letters.

Examples for the parameter syntax and the use of options will be presented throughout this training.



Enter the command to the command line.  
Add one blank.  
Push F1 to get the online help for the specified command.

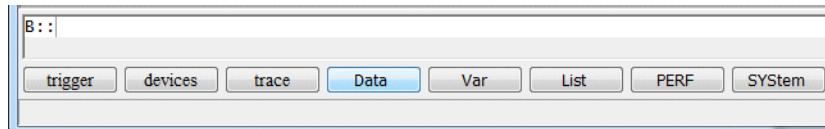
# Message Line



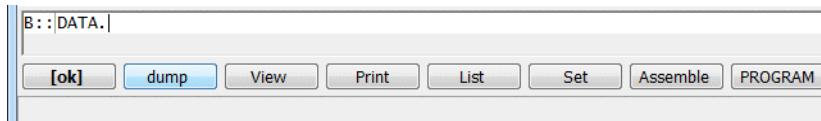
- **Message line** for system and error messages
- **Message Area window** for the display of the last system and error messages

The softkey line allows to enter a specific command step by step. Here an example:

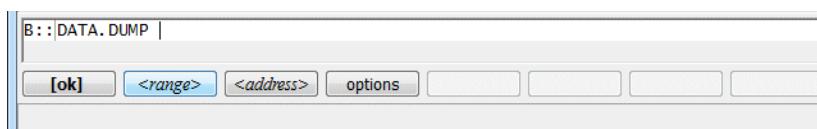
Select the command group, here **Data**.



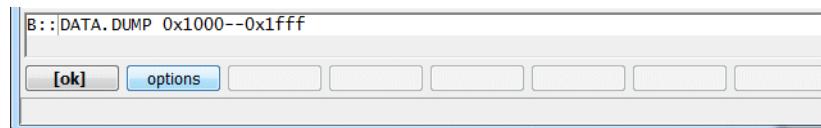
Select the subcommand, here **dump**.



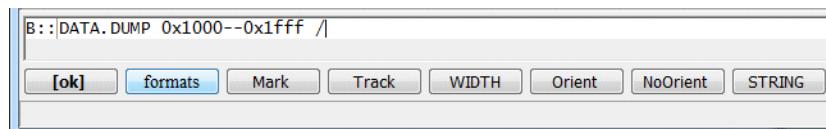
Angle brackets request an entry from the user,  
here e.g. the entry of a <range> or an <address>.



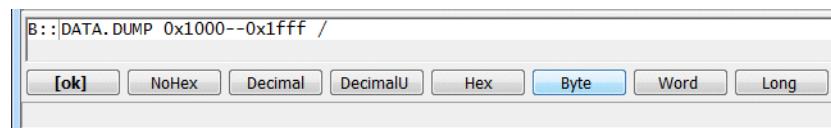
The display of the hex. dump can be adjusted to your needs by an option.



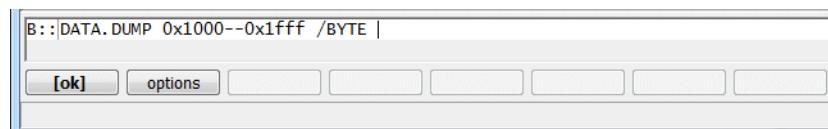
Select the option **formats** to get a list of all format options.



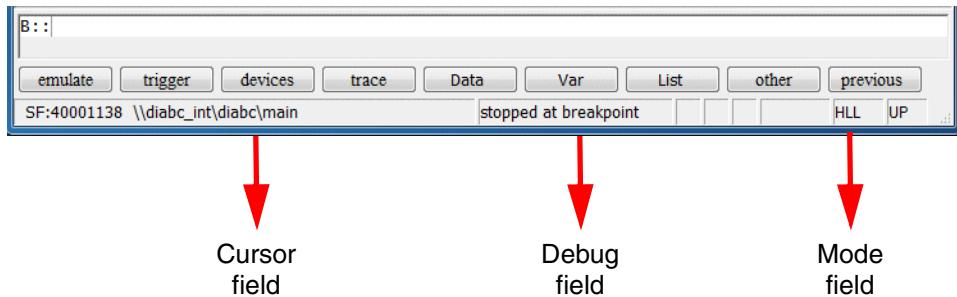
Select a format option, here **Byte**.



The command is complete now.



## State Line



The **Cursor** field of the state line provides:

- Boot information (Booting ..., Initializing ... etc.).
- Information on the item selected by one of the TRACE32 PowerView cursors.

The **Debug** field of the state line provides:

- Information on the debug communication (system down, system ready etc.)
- Information on the state of the debugger (running, stopped, stopped at breakpoint etc.)

The **Mode** field of the state line indicates the debug mode. The debug mode defines how source code information is displayed.

- Asm = assembler code
- HLL = programming language code/high level language
- Mix = a mixture of both

It also defines how single stepping is performed (assembler line-wise or programming language line-wise).

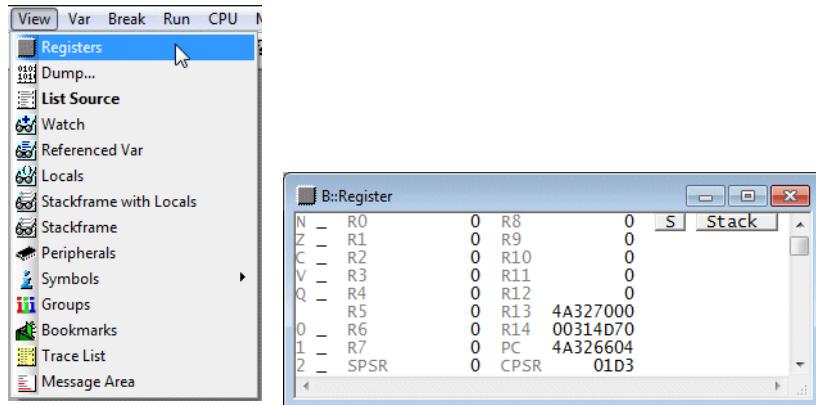


The debug mode can be changed by using the **Mode** pull-down.

# Registers

## Core Registers

### Display the Core Registers

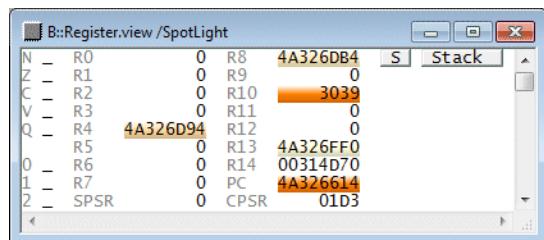


Register.view

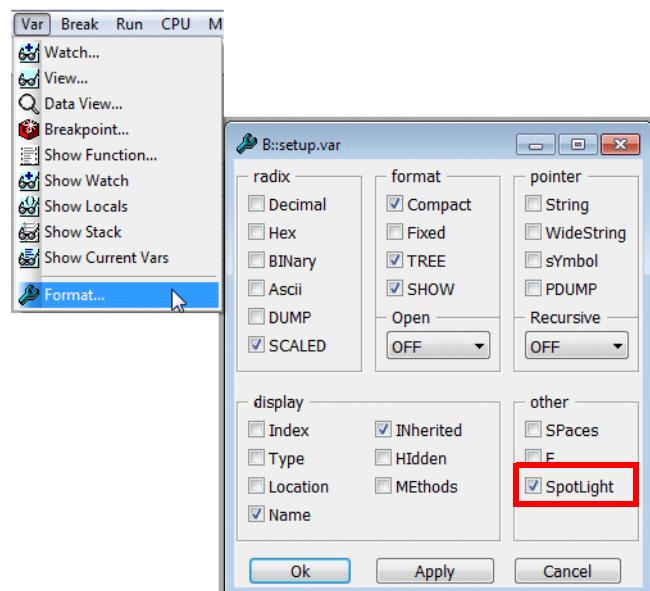
# Colored Display of Changed Registers

The option /SpotLight advises TRACE32 PowerView to mark changes.

```
Register.view /SpotLight ; The registers changed by the last  
; step are marked in dark red.  
  
; The registers changed by the  
; step before the last step are  
; marked a little bit lighter.  
  
; This works up to a level of 4.
```



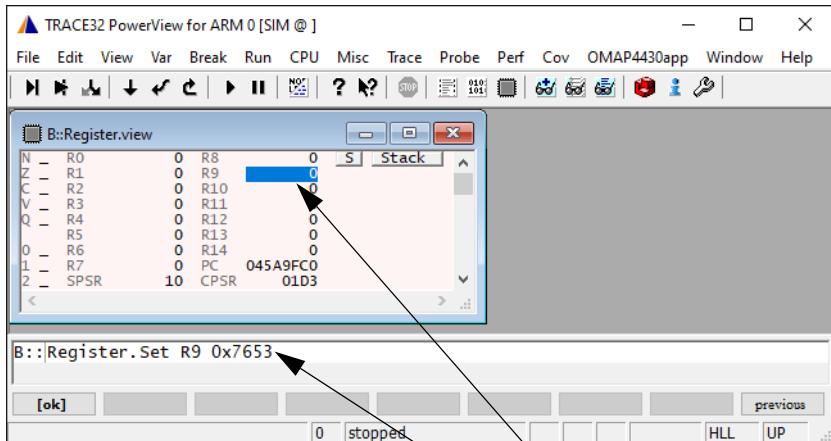
## Establish /SpotLight as default setting



### SETUP.Var %SpotLight

Establish the option SpotLight as default setting for  
- all Variable windows  
- Register window  
- PERipheral window  
- the HLL Stack Frame  
- Data.dump window

## Modify the Contents of a Core Register



By double clicking to the register contents  
a **Register.Set** command is automatically displayed  
in the command line.

Enter the new value and press return to modify the  
register contents.

**Register.Set <register> <value>**

Modify register

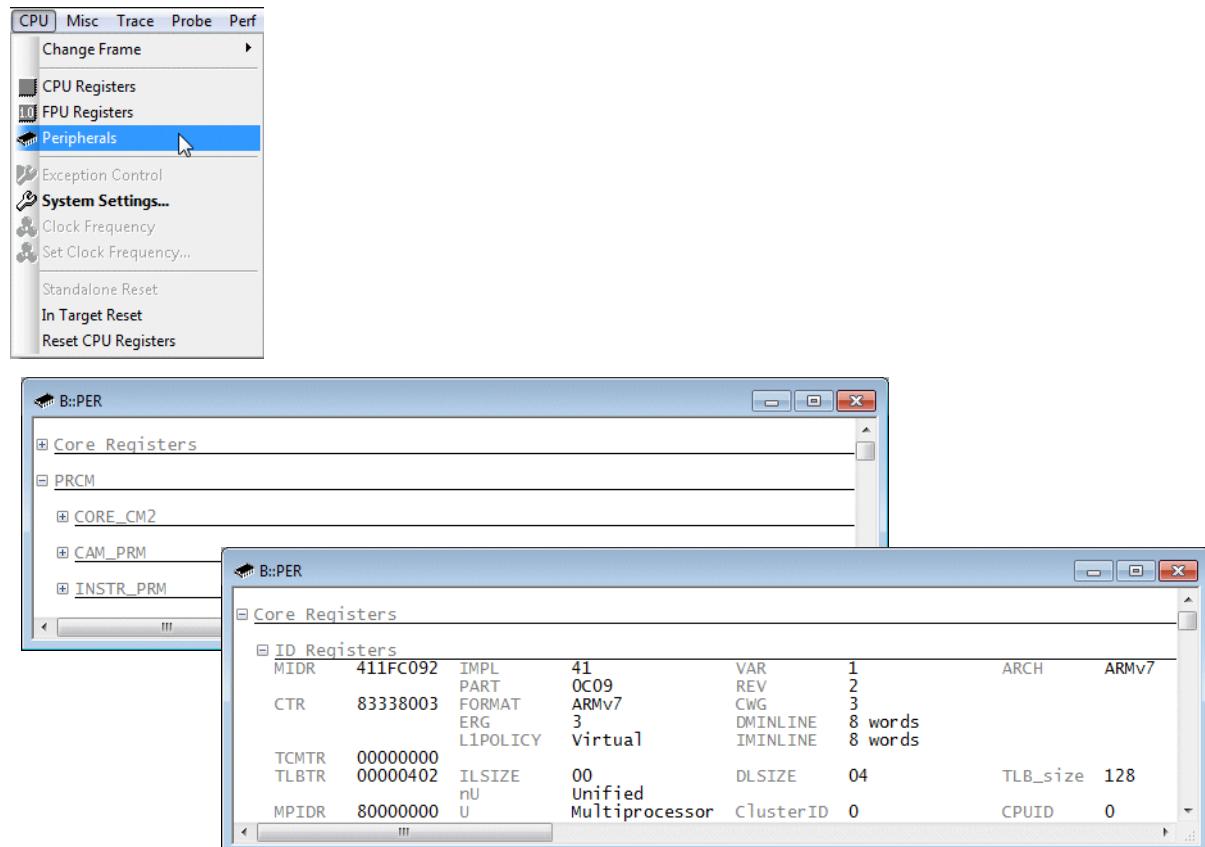
# Special Function Register

## Display the Special Function Registers

TRACE32 supports a free configurable window to display/manipulate configuration registers and the on-chip peripheral registers at a logical level. Predefined peripheral files are available for most standard processors/chips.

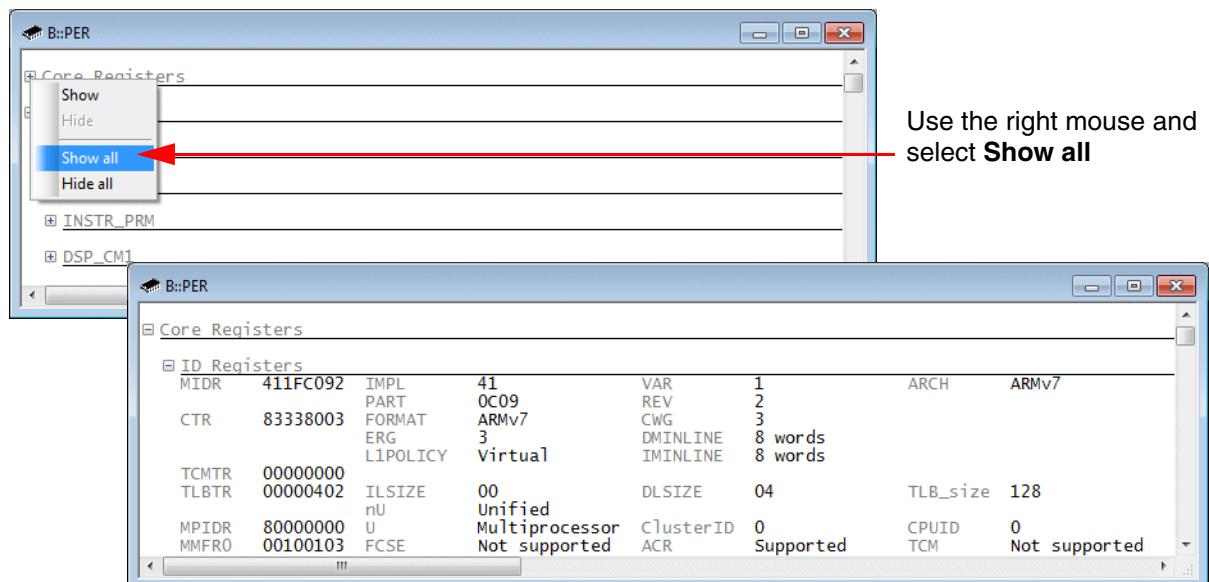
### Tree Display

The individual configuration registers/on-chip peripherals are organized by TRACE32 PowerView in a tree structure. On demand, details about a selected register can be displayed.



Please be aware, that TRACE32 permanently updates all windows. The default update rate is 10 times per second.

Sometimes it might be useful to expand the tree structure from the start.



### Commands:

**PER.view <filename> [<tree\_item>]** Display the configuration registers/on-chip peripherals

```
; Display all functional units in expanded mode
; , advises TRACE32 PowerView to use the default peripheral file
; * stands for all <tree-items>
PER.view , "*"
```

```
; Display the functional unit "ID Registers" within "Core Registers"
; in expanded mode
PER.view , "Core Registers, ID Registers"
```

B::PER.view , "Core Registers, ID Registers"

						ARCH	ARMv7
MIDR	411FC093	IMPL	41	VAR	1		
		PART	0C09	REV	3		
CTR	83338003	FORMAT	ARMv7	CWG	3		
		ERG	3	DMINLINE	8 words		
TCMTR	00000000	LIPOLICY	Virtual	IMINLINE	8 words		
TLBTR	00000402	ILSIZE	00	DLSIZE	04	TLB_size	128
		nU	Unified				
MPIDR	80000000	U	Multiprocessor	ClusterID	0	CPUID	0
MMFR0	00100103	FCSE	Not supported	ACR	Supported	TCM	Not supported
		OSS	Not supported	CC_CPUA	Supported	PMSA	Not supported

```
; Display the functional unit "DMA_Channel_0" within "sDMA_Module, sDMA"
; in expanded mode
PER.view , "sDMA_Module, sDMA, DMA_Channel_0"
```

B::PER.view , "sDMA\_Module, sDMA, DMA\_Channel\_0"

DMA4_CCEN1_0	000083E8	CURRENT_DESCRIPTOR_NBR	83E8	BUFFERING_DISABLE	0
DMA4_CCEN1_0	00550C3A	CURRENT_ELMNT_NBR	550C3A	SUPERVISOR	0
DMA4_CCFN1_0	0000F0C6	CURRENT_FRAME_NBR	F0C6	TRANSPARENT_COPY_ENABLE	1
DMA4_CCR1_0	0102A020	WRITE_PRIORITY	0		
		PREFETCH	0		
		BS	0		

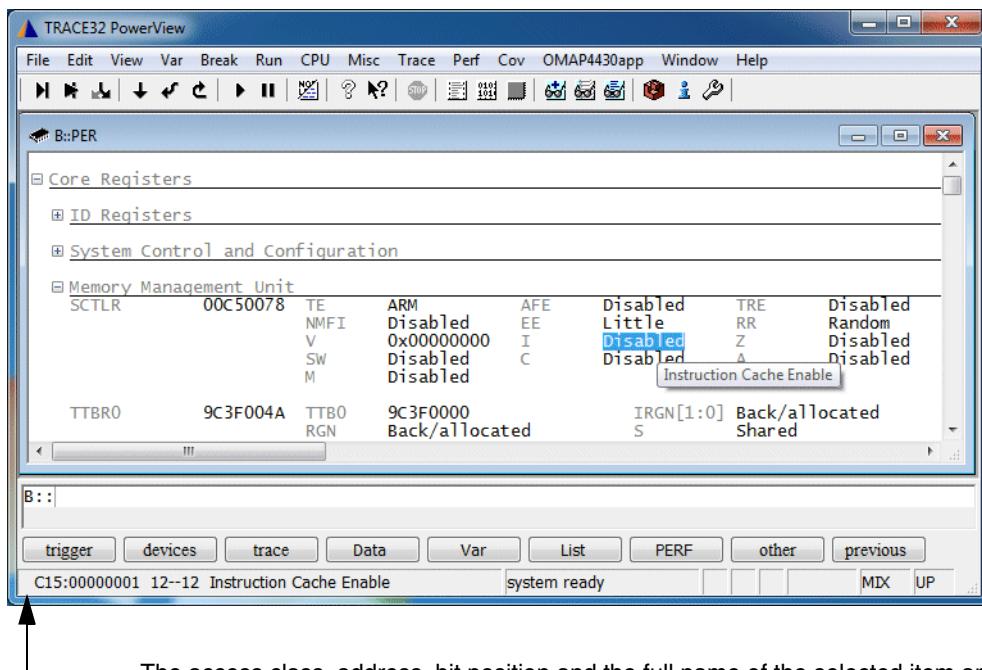
The following command sequence can be used to save the contents of all configuration registers/on-chip peripheral registers to a file.

```
; PRinTer.FileType ASCIIIE ; Select ASCII ENHANCED as output
                            ; format
                            ; (default output format)

PRinTer.FILE Per.lst      ; Define Per.lst as output file

WinPrint.PER.view          ; Save contents of all
                            ; configuration registers/on-chip
                            ; peripheral registers to the
                            ; specified file
```

# Details about a Single Special Function Register

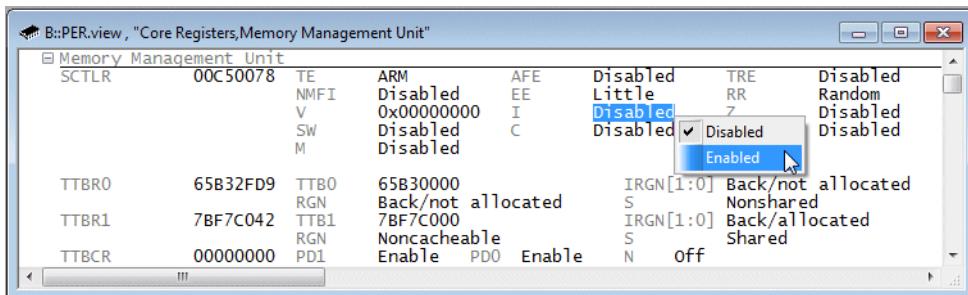


The access class, address, bit position and the full name of the selected item are displayed in the state line; the full name of the selected item is taken from the processor/chip manual.

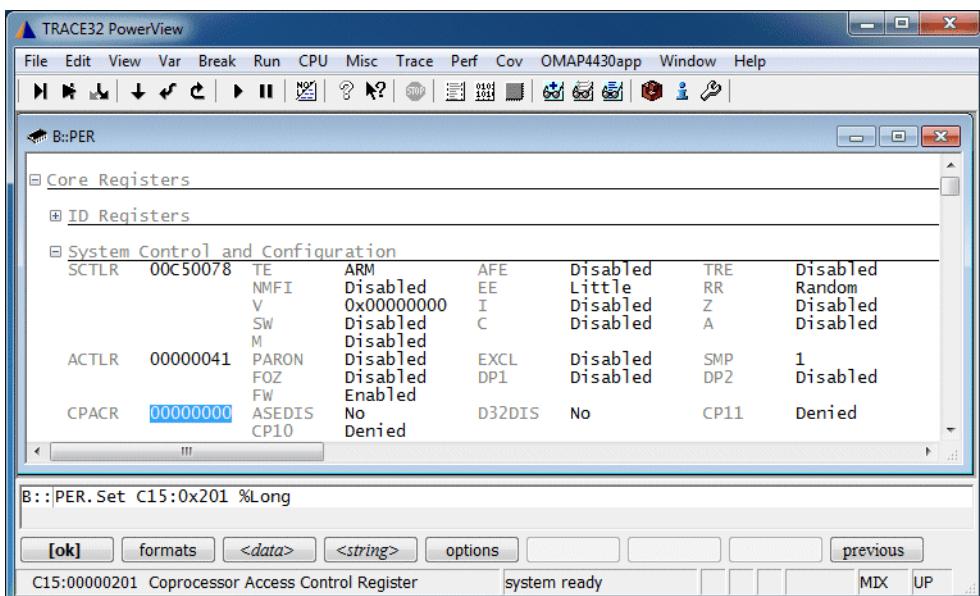
# Modify a Special Function Register

You can modify the contents of a configuration/on-chip peripheral register:

- By pressing the right mouse button and selecting one of the predefined values from the pull-down menu.



- By a double-click to a numeric value. A **PER.Set** command to change the contents of the selected register is displayed in the command line. Enter the new value and confirm it with return.



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

Modify configuration register/on-chip peripheral

**Data.Set** <address>|<range> [%<format>] <value>

Modify memory

**Data.Set** is equivalent to **PER.Set.simple** if the configuration register is memory mapped.

```
PER.Set.simple D:0xF87FFF10 %Long 0x00000b02
```

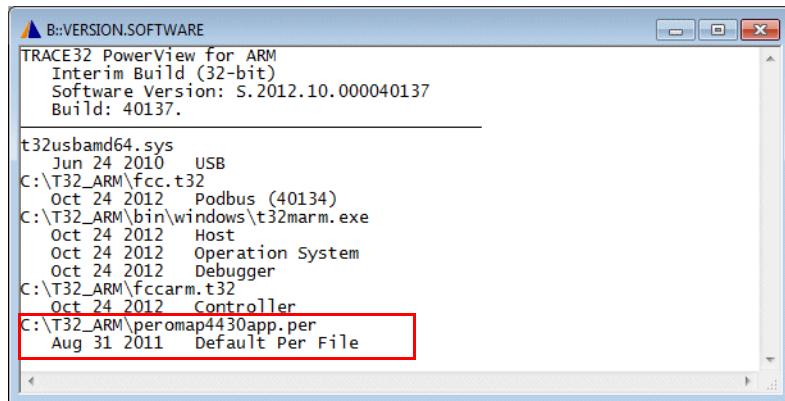
# The PER Definition File

The layout of the PER window is described by a PER definition file.

The definition can be changed to fit to your requirements using the **PER** command group.

The path and the version of the actual PER definition file can be displayed by using:

## VERSION.SOFTWARE



## PER.view <filename>

Display the configuration registers/on-chip peripherals specified by  
<filename>

```
PER.view C:\T32_ARM\percortexa9mpcore.per
```

# Memory Display and Modification

---

This training section introduces the most often used methods to display and modify memory:

- The **Data.dump** command, that displays a hex dump of a memory area, and the **Data.Set** command that allows to modify the contents of a memory address.
- The **List** (former **Data.List**) command, that displays the memory contents as source code listing.

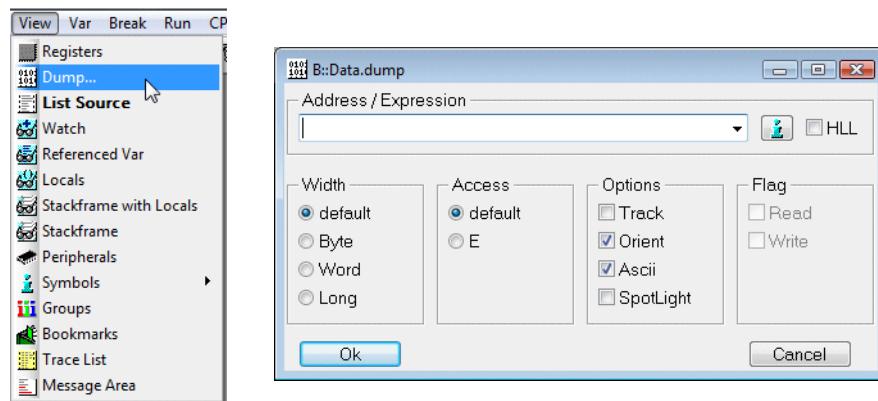
A so-called **access class** is always displayed together with a memory address. The following access classes are available for all processor architectures:

P:1000	<b>Program</b> address 0x1000
D:6814	<b>Data</b> address 0x6814

For additional access classes provided by your processor architecture refer to your “[Processor Architecture Manuals](#)”.

# The Data.dump Window

## Display the Memory Contents



## Use an Address to Specify the Start Address for the Data.dump Window

The screenshot shows the TRACE32 debugger interface. At the top, there is a title bar with the text "B::Data.dump". Below it is a settings dialog box titled "Address / Expression" which contains the value "0x6814". This input field is highlighted with a red rectangle. The dialog also includes sections for "Width" (radio buttons for "default", "Byte", "Word", and "Long"), "Access" (radio buttons for "default" and "E"), "Options" (checkboxes for "Track", "Orient", "Ascii", and "SpotLight"), and "Flag" (checkboxes for "Read" and "Write"). At the bottom of the dialog are "Ok" and "Cancel" buttons. Below the dialog is the main data dump window titled "B::Data.dump (0x6814) /DIALOG". The window displays memory starting at address 0x6814. The first few lines of memory are:

address	0	4	8	C
SD : 00006810	83421780	06004185	038255C0	A4404D68
SD : 00006820	2C10F820	54018000	28506482	A5DD1248
SD : 00006830	42010004	87354C60	08017201	2423DC18
SD : 00006840	00000040	2C014624	42A15055	0820A090
SD : 00006850	6243200A	050402C0	18004142	410B0449
SD : 00006860	02066040	500010F5	80800182	04B11560
SD : 00006870	982A2100	40000801	8512040D	10958806
SD : 00006880	02254894	42830588	A0800324	C0484990



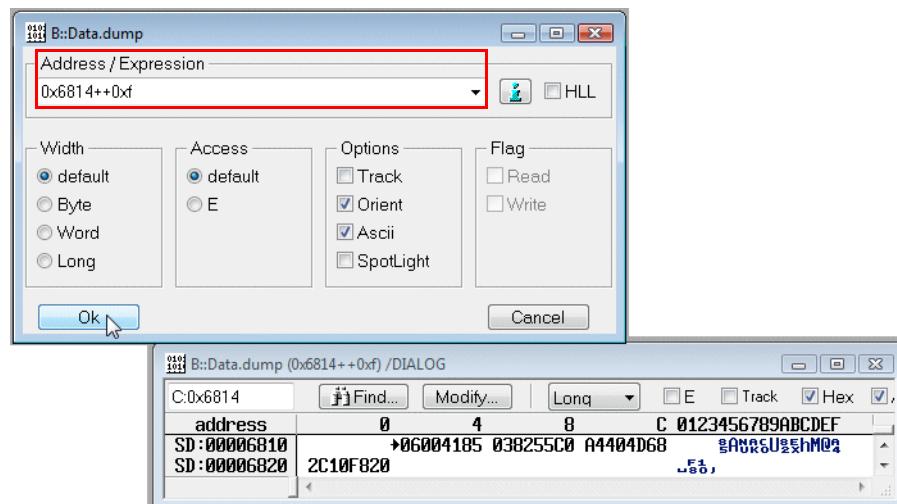
Please be aware, that TRACE32 permanently updates all windows. The default update rate is 10 times per second.

## Use an Address Range to Specify the Addresses for the Data.dump Window

If you enter an address range, only data for the specified address range are displayed. This is useful if a memory area close to memory-mapped I/O registers should be displayed and you do not want TRACE32 PowerView to generate read cycles for the I/O registers.

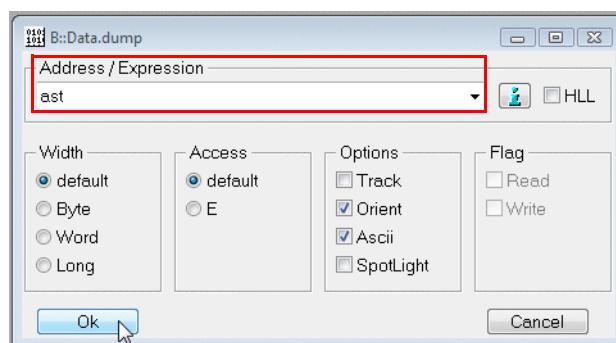
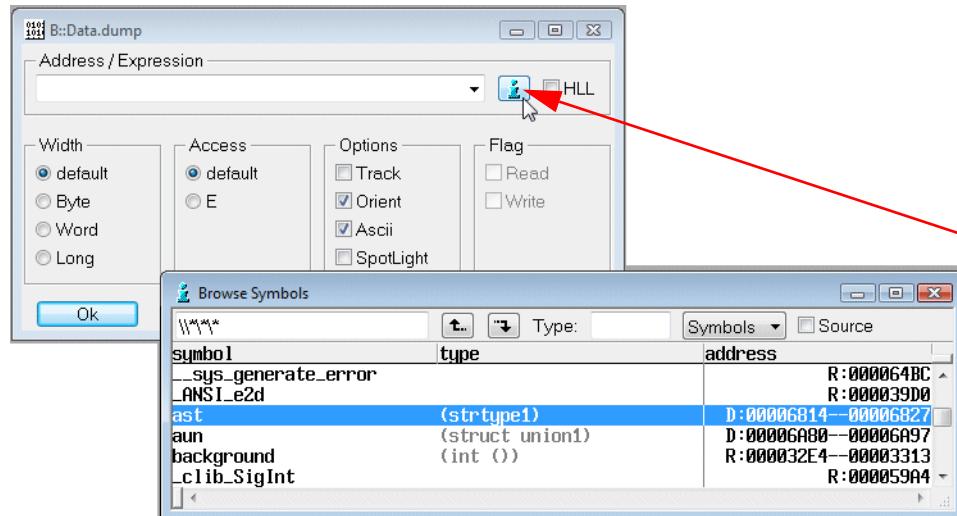
### Conventions for address ranges:

- <start\_address>--<end\_address>
- <start\_address>..<end\_address>
- <start\_address>++<offset\_in\_byte>
- <start\_address>++<offset\_in\_word> (for DSPs)



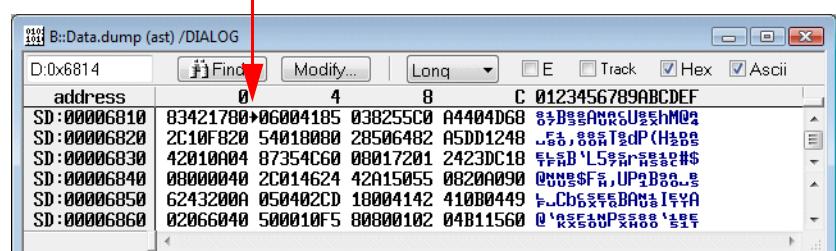
## Use a Symbol to Specify the Start Address for the Data.dump Window

Use **i** to select any symbol name or label known to TRACE32 PowerView.



By default an oriented display  
is used (line break at 2<sup>X</sup>).

A small arrow indicates  
the specified dump address.

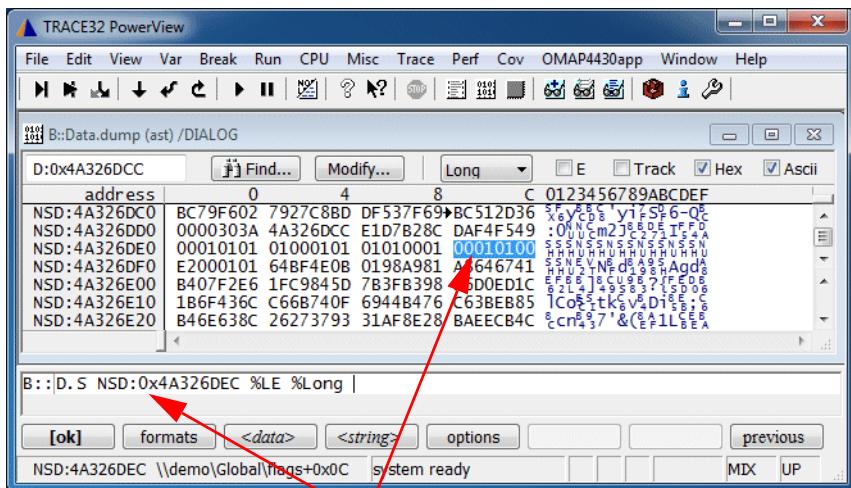


**Data.dump** <address> | <range> [/<option>]

Display a hex dump of the memory

```
Data.dump 0x6814 ; Display a hex dump starting at  
; address 0x6814  
  
Data.dump 0x6810--0x682f ; Display a hex dump of the  
; specified address range  
  
Data.dump 0x6810..0x682f ; Display a hex dump of the  
; specified address range  
  
Data.dump 0x6810++0x1f ; Display a hex dump of the  
; specified address range  
  
Data.dump ast ; Display a hex dump starting at  
; the address of the label ast  
  
Data.dump ast /Byte ; Display a hex dump starting at  
; the address of the label ast in  
; byte format
```

# Modify the Memory Contents



By a left mouse double-click to the memory contents

a **Data.Set** command is automatically  
displayed in the command line,  
you can enter the new value and  
confirm it with return.

**Data.Set <address>|<range> [%<format>] <value> [/<option>]**

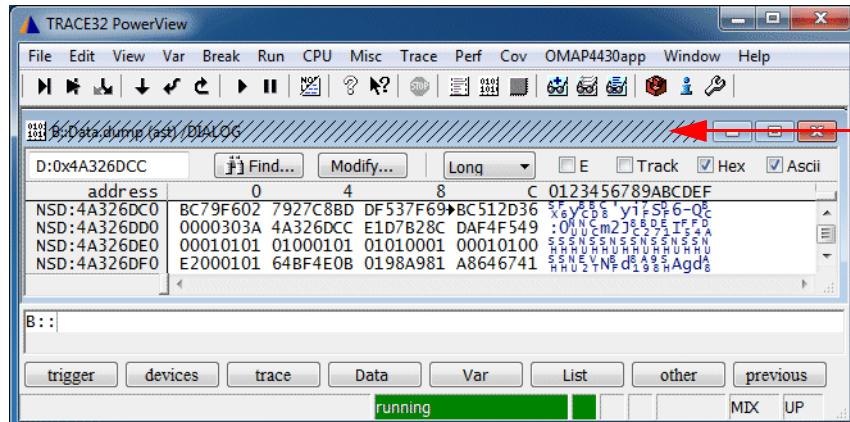
```
Data.Set 0x6814 0xaa ; Write 0xaa to the address
                      ; 0x6814

Data.Set 0x6814 %Long 0xaaaa ; Write 0xaaaa as a 32 bit value to
                             ; the address 0x6814, add the
                             ; leading zeros automatically

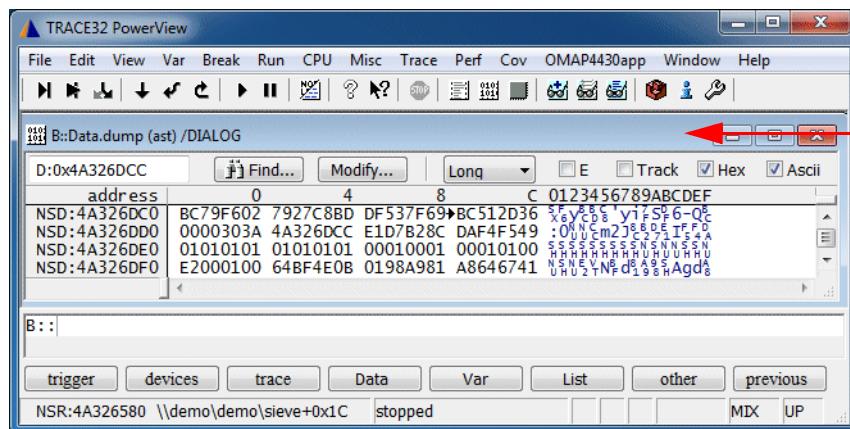
Data.Set 0x6814 %LE %Long 0xaaaa ; Write 0xaaaa as a 32 bit value to
                                 ; the address 0x6814, add the
                                 ; leading zeros automatically
                                 ; Use Little Endian mode
```

# Run-time Memory Access

TRACE32 PowerView updates the displayed memory contents by default only if the cores is stopped.



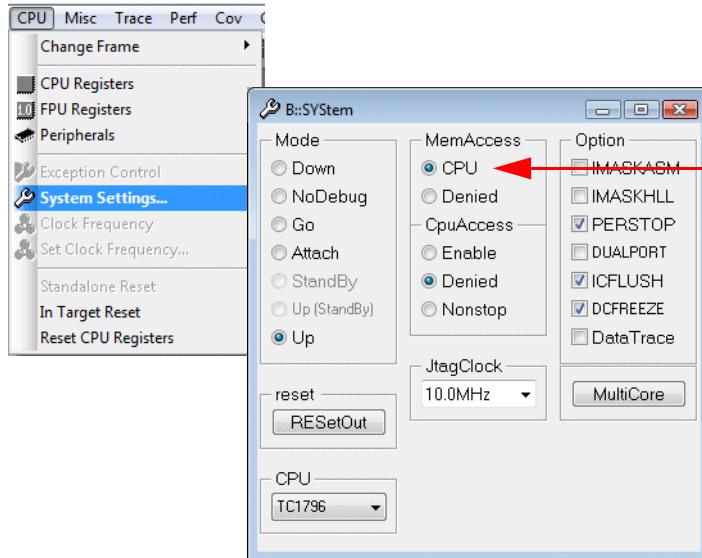
A hatched window frame indicates that the information display is frozen because the core is executing the program.



The plain window frame indicates that the information is updated, because the program execution is stopped.

Various cores allow a debugger to read and write physical memory (not cache) while the core is executing the program. The debugger has in most cases direct access to the processor/chip internal bus, so no extra load for the core is generated by this feature.

Open the **SYStem** window in order to check if your processor architecture allows a debugger to read/write memory while the core is executing the program:



**MemAccess Enable/NEXUS/DAP** indicates that the core allows the debugger to read/write the memory while the core is executing the program.

Please be aware that caches, MMUs, tightly-coupled memories and suchlike add conditions to the run-time memory access or at worst make its use impossible.

## Restrictions

The following description is only a rough overview on the restrictions. Details about your core can be found in the [Processor Architecture Manual](#).

## Cache

If run-time memory access for a cached memory location is enabled the debugger acts as follows:

- **Program execution is stopped**

The data is read via the cache respectively written via the cache.

- **Program execution is running**

Since the debugger has no access to the caches while the program execution is running, the data is read from physical memory. The physical memory contains the current data only if the cache is configured as write-through for the accessed memory location, otherwise out-dated data is read.

Since the debugger has no access to the cache while the program execution is running, the data is written to the physical memory. The new data has only an effect on the current program execution if the debugger can invalidate the cache entry for the accessed memory location. This useful feature is not available for most cores.

## MMU

Debuggers have no access to the TLBs while the program execution is running. As a consequence run-time memory access can not be used, especially if the TLBs are dynamically changed by the program.

In the exceptional case of static TLBs, the TLBs can be scanned into the debugger. This scanned copy of the TLBs can be used by the debugger for the address translation while the program execution is running.

## Tightly-coupled Memory

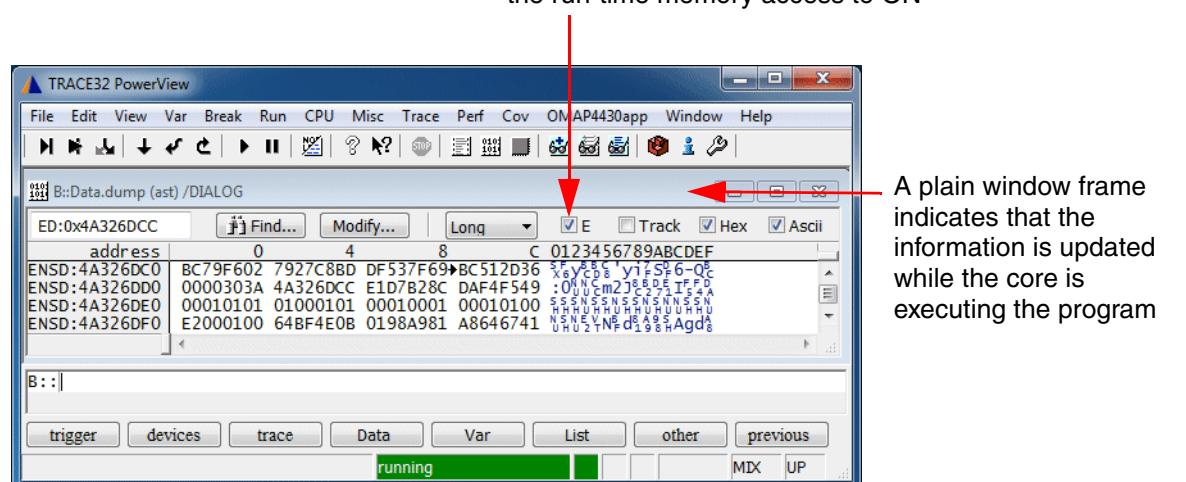
Tightly-coupled memory might not be accessible via the system memory bus.

## Usage

The usage of the non-intrusive run-time memory access has to be configured explicitly. Two methods are provided:

- Configure the run-time memory access for a specific memory area.
- Configure run-time memory access for all windows that display memory contents (not available for all processor architectures).

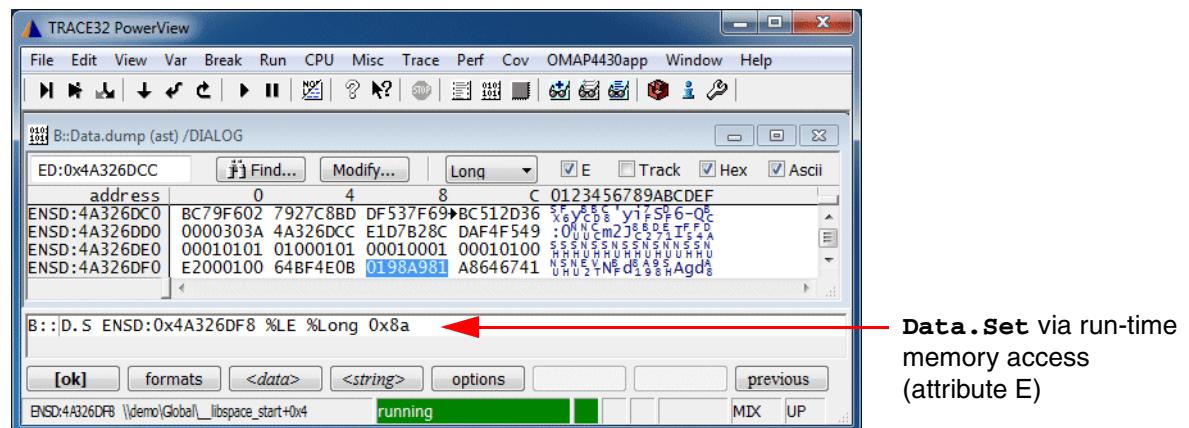
## Configure the run-time memory access for a specific memory area:



If the **E** check box is enabled, the attribute E is added to the memory class:

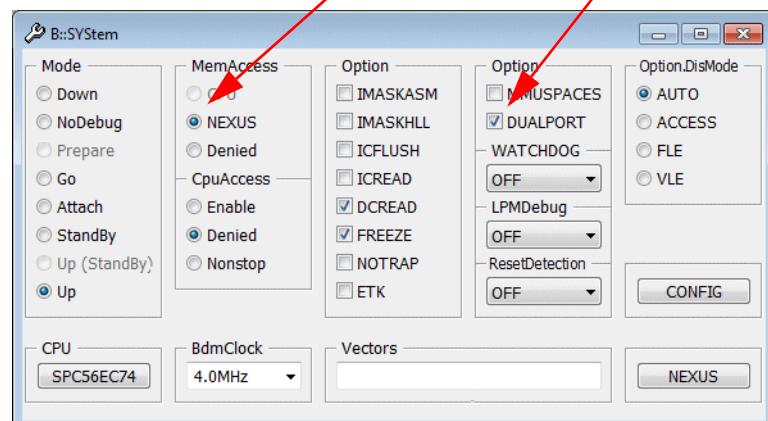
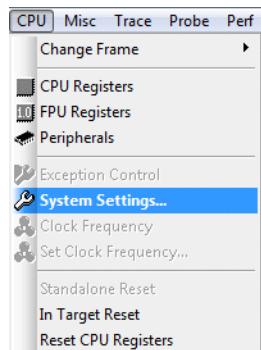
EP:1000	Program address 0x1000 with run-time memory access
ED:6814	Data address 0x6814 with run-time memory access

Write accesses to the memory work correspondingly:



```
SYStem.MemAccess Enable ; Enable the non-intrusive  
; run-time memory access  
  
;...  
  
Go ; Start program execution  
  
Data.dump E:0x6814 ; Display a hex dump starting at  
; address 0x6814 via run-time  
; memory access  
  
Data.Set E:0x6814 0xAA ; Write 0xAA to the address  
; 0x6814 via run-time memory  
; access
```

## Configure the run-time memory access for all windows that display memory (not available for all cores):



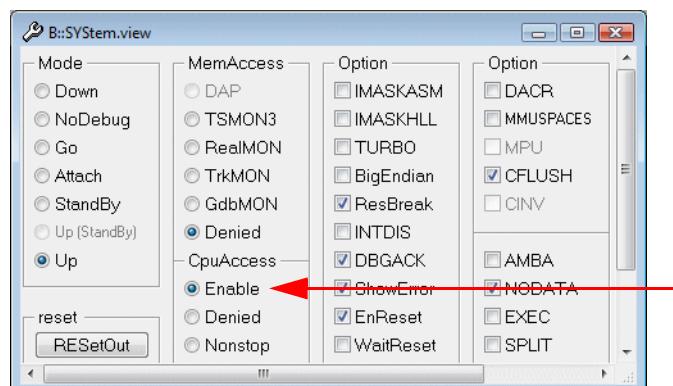
If MemAccess **Enable/NEXUS/DAP** is selected and **DUALPORT** is checked, run-time memory is configured for all windows that display memory

All windows that display memory have a plain window frame, because they are updated while the core is executing the program

Write access is possible for all memories while the core is executing the program

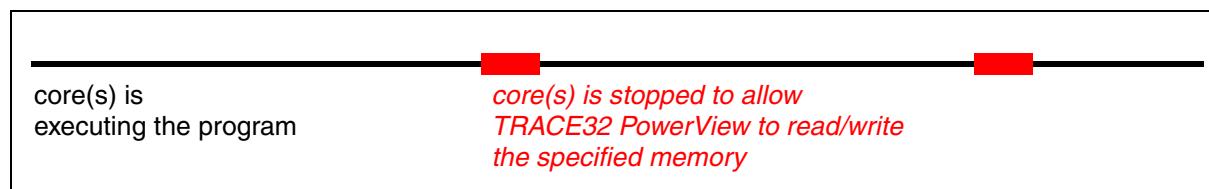
```
SYStem.MemAccess Enable ; Enable the non-intrusive  
                        ; run-time memory access  
  
SYStem.Option.DUALPORT ON ; Activate the run-time memory  
                          ; access for all windows that  
                          ; display memory  
  
                          ; this SYStem.Option is only  
                          ; available for some processor  
                          ; architectures  
  
;...  
  
Go ; Start program execution  
  
Data.dump 0x6814 ; Display a hex dump starting at  
                   ; address 0x6814 via run-time  
                   ; memory access  
  
Data.Set 0x6814 0xAA ; Write 0xAA to the address  
                      ; 0x6814 via run-time memory  
                      ; access
```

If your processor architecture doesn't allow a debugger to read or write memory while the core is executing the program, you can activate an intrusive run-time memory access if required.



**CpuAccess Enable** allows an intrusive run-time memory access

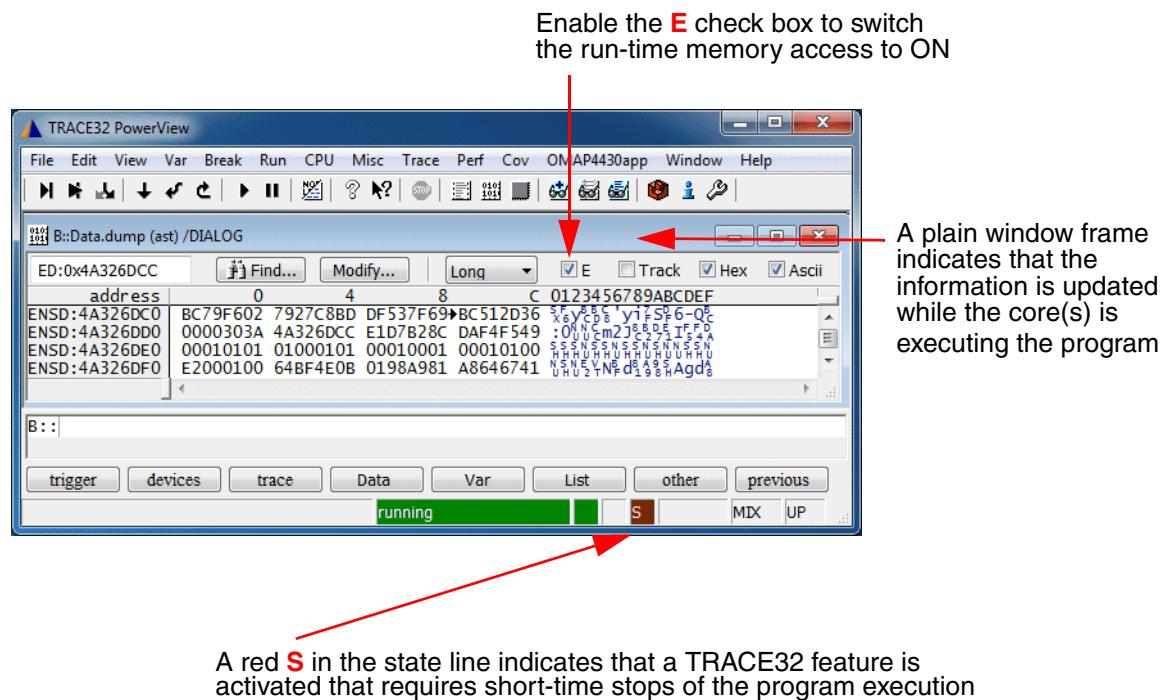
If an intrusive run-time memory access is activated, TRACE32 stops the program execution periodically to read/write the specified memory area. Each update takes at least **50 us**.



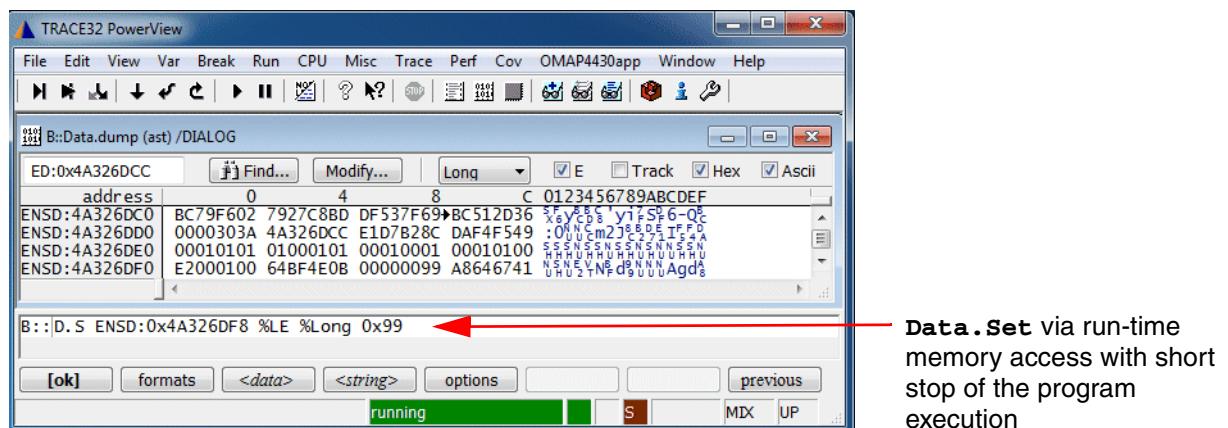
The time taken by a short stop depends on various factors:

- The time required by the debugger to start and stop the program execution on a processor/core (main factor).
- The number of cores that need to be stopped and restarted.
- Cache and MMU assessments that need to be performed to read the information of interest.
- The type of information that is read during the short stop.

An intrusive run-time memory access is only possible for a **specific memory area**.

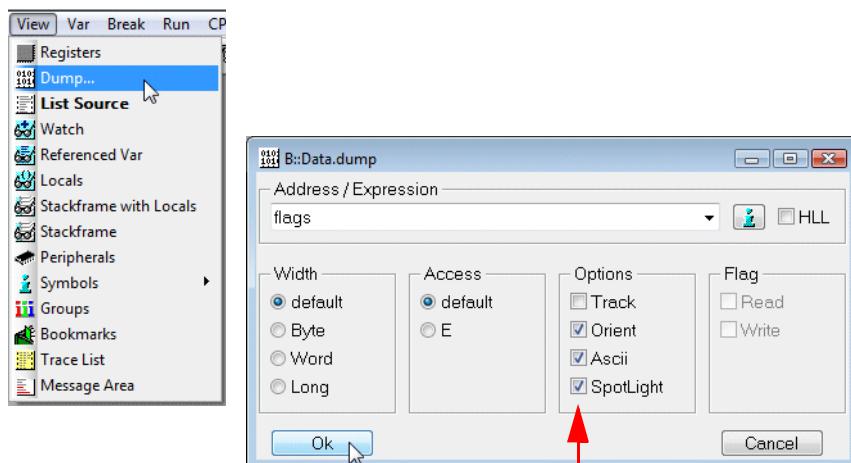


Write accesses to the memory work correspondingly:



```
SYStem.CpuAccess Enable ; Enable the intrusive  
; run-time memory access  
  
;...  
  
Go ; Start program execution  
  
Data.dump E:0x6814 ; Display a hex dump starting at  
; address 0x6814 via an intrusive  
; run-time memory access  
  
Data.Set E:0x6814 0xAA ; Write 0xAA to the address  
; 0x6814 via an intrusive  
; run-time memory access
```

# Colored Display of Changed Memory Contents



Enable the option **SpotLight** to mark the memory contents changed by the last 4 single steps in orange, older changes being lighter.

B::Data.dump (flags) /SpotLight /DIALOG							
address	0	1	2	3	4	5	6
SD:00007E78	16	A5	3D	90	01	00	01
							01
SD:00007E80	01	01	01	01	01	01	01
							01
SD:00007E88	01	01	01	01	01	01	01
							01
SD:00007E90	00	47	03	4A	50	C2	90
							A1
SD:00007E98	8A	01	01	85	33	15	99
							B2
SD:00007EA0	1A	00	10	4B	6C	12	0D
							02
SD:00007EA8	16	06	2D	31	52	48	81
							58

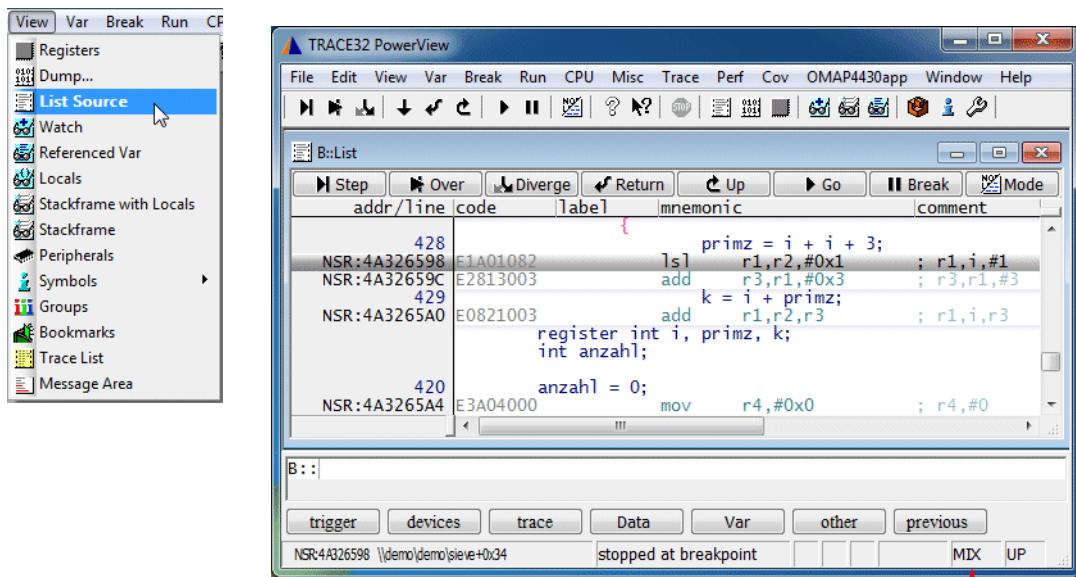
Data.dump flags /SpotLight

; Display a hex dump starting at  
; the address of the label flags

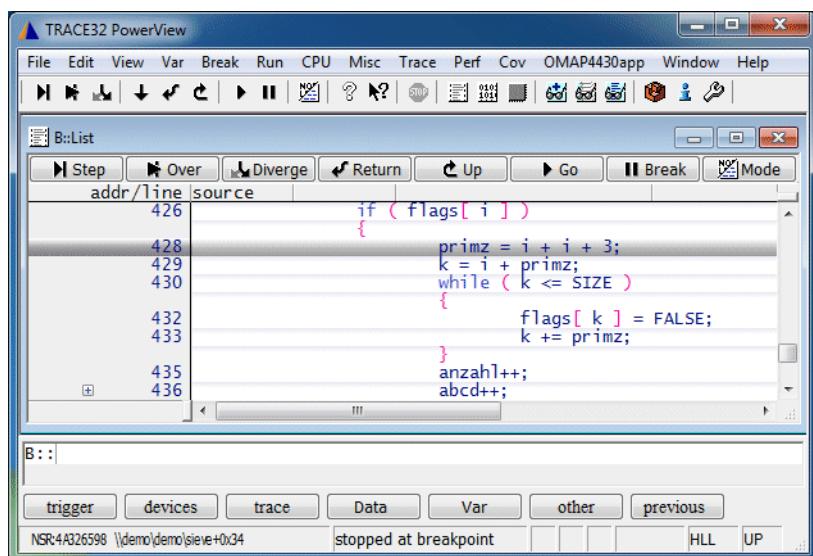
; Mark changes

# The List Window

## Displays the Source Listing Around the PC

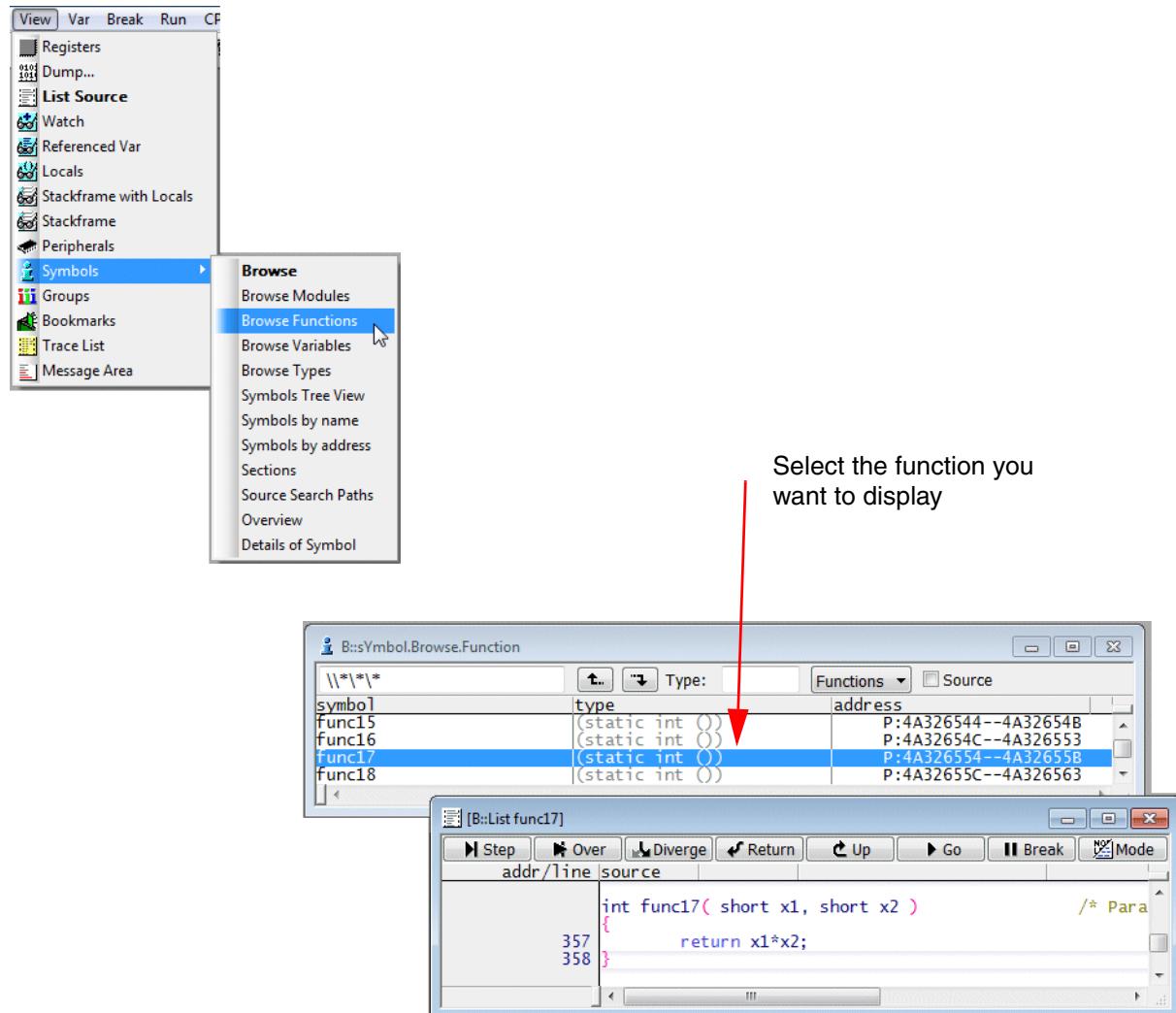


If MIX mode is selected for debugging, assembler and HLL information is displayed



If HLL mode is selected for debugging, only HLL information is displayed

# Displays the Source Listing of a Selected Function



**List** [<address>] [/<option>]

Display source listing

**Data.List** [<address>] [/<option>]

Display source listing

```
List ; Display a source listing  
; around the PC  
  
List E: ; Display a source listing,  
; allow scrolling while the  
; program execution is running  
  
List * ; Open the symbol browser to  
; select a function for display  
  
List func17 ; Display a source listing of  
; func17
```

# Breakpoints

---

Videos about the breakpoint handling can be found here:

[https://www.lauterbach.com/tut\\_breakpoints.html](https://www.lauterbach.com/tut_breakpoints.html)

## Breakpoint Implementations

---

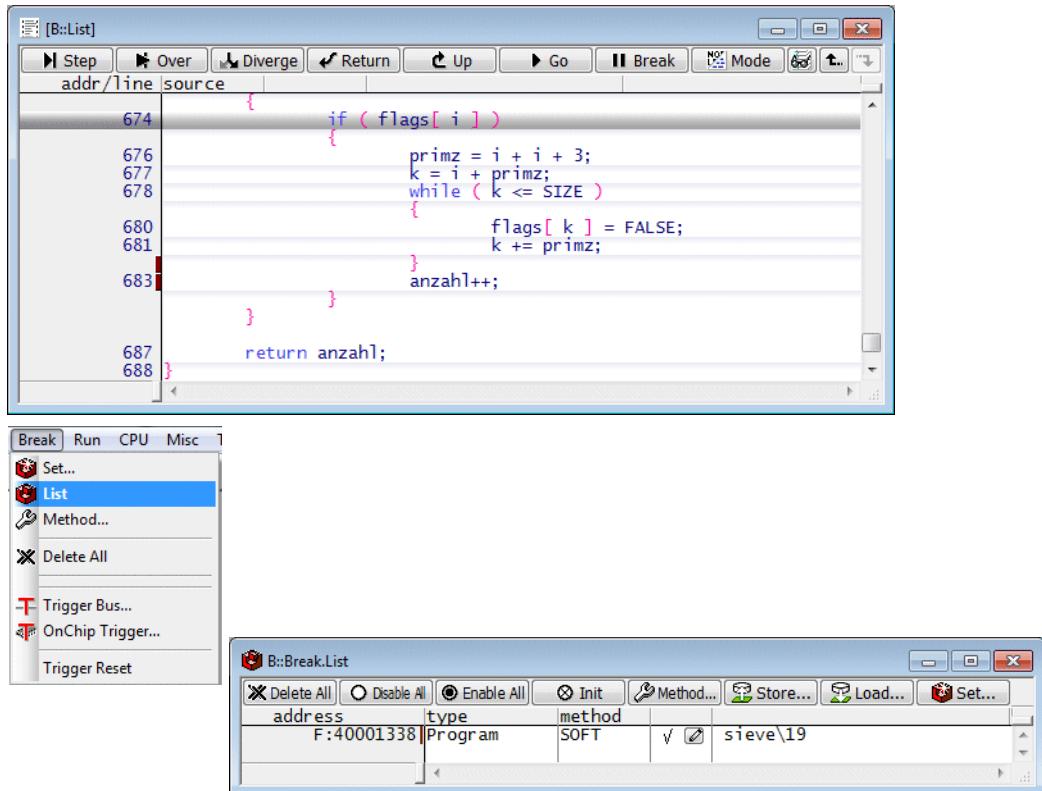
A debugger has two methods to realize breakpoints: Software breakpoints and Onchip breakpoints.

### Software Breakpoints in RAM

---

The default implementation for breakpoints on instructions is a Software breakpoint. If a Software breakpoint is set the original instruction at the breakpoint address is patched by a special instruction (usually TRAP) to stop the program and return the control to the debugger.

The number of software breakpoints is unlimited.



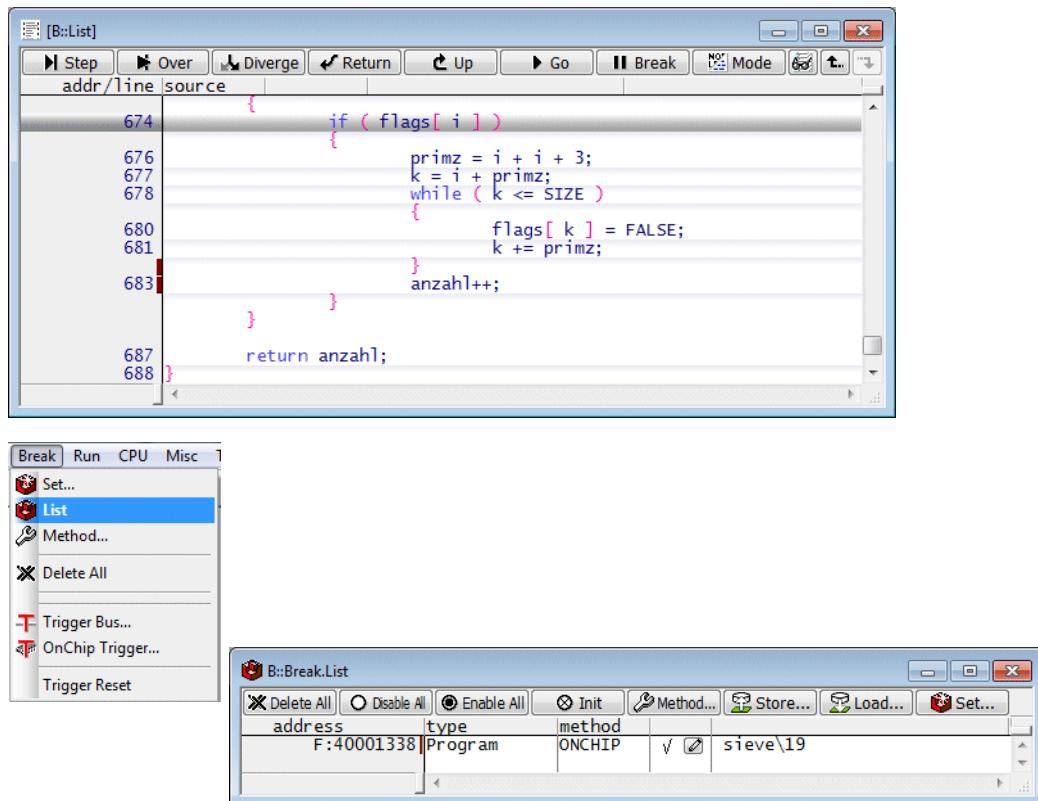
Breakpoints on instructions are called **Program** breakpoints by TRACE32 PowerView.

	Please be aware that TRACE32 PowerView always tries to set an Onchip breakpoint, when the setting of a Software Breakpoint fails.
---	---

TRACE32 allows to set Software breakpoints to FLASH. Please be aware that the affected FLASH sector has to be erased and programmed in order to patch the break instruction used by the Software breakpoint. This usually takes some time and reduces the number of FLASH erase cycles. For details refer to “[Software Breakpoints in FLASH](#)” (norflash.pdf).

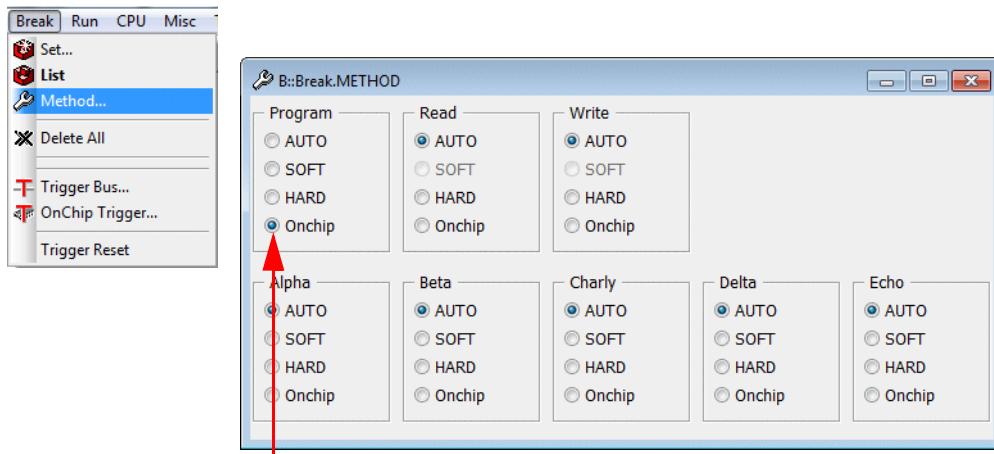
## Onchip Breakpoints in NOR Flash

Most core(s) provide a small number of Onchip breakpoints in form of breakpoint registers. These Onchip breakpoints can be used to set breakpoints to instructions in read-only memory like onchip or NOR FLASH.



Since Software breakpoints are used by default for Program breakpoints, TRACE32 PowerView can be informed explicitly where to use Onchip breakpoints. Depending on your memory layout, the following methods are provided:

1. If the code is completely located in read-only memory, the default implementation for the Program breakpoints can be changed.



Change the implementation of Program breakpoints to **Onchip**

#### Break.METHOD Program Onchip

Advise TRACE32 PowerView to implement Program breakpoints always as Onchip breakpoints

2. If the code is located in RAM and onchip/NOR FLASH you can define code ranges where Onchip breakpoints are used.

**MAP.BOnchip <range>**

Advise TRACE32 PowerView to implement Program breakpoints as Onchip breakpoints within the defined address range

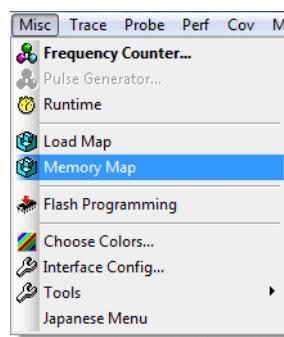
**MAP.List**

Check your settings

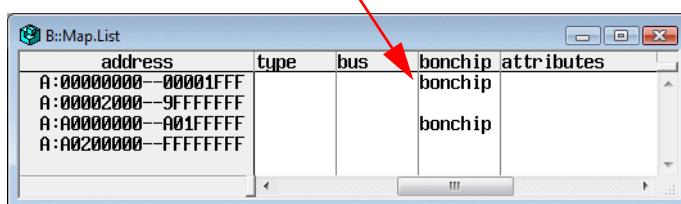
MAP.BOnchip 0x0++0x1FFF

MAP.BOnchip 0xA0000000++0x1FFFF

Check your settings as follows:



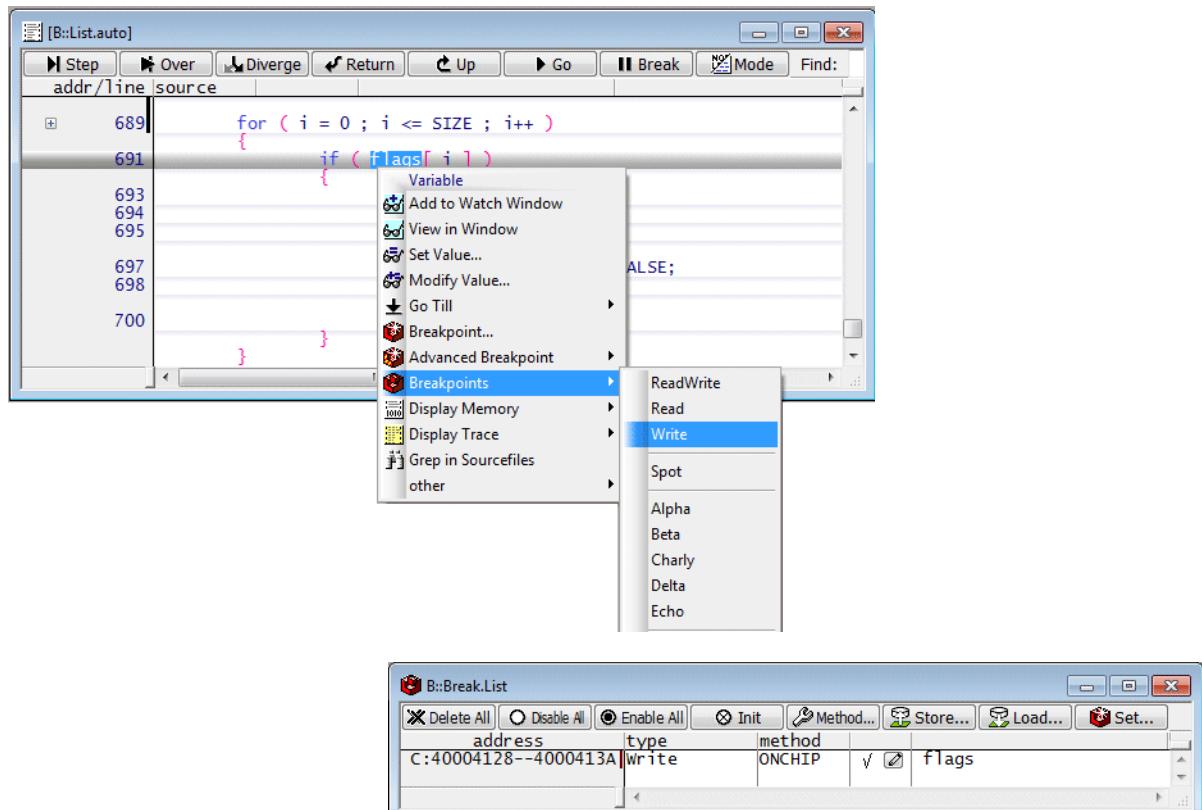
For the specified address ranges Program breakpoints are implemented as Onchip breakpoints. For all other memory areas Software breakpoints are used.



address	type	bus	bonchip	attributes
A:00000000--00001FFF				
A:00002000--9FFFFFFF				
A:A0000000--A01FFFFF				
A:A0200000--FFFFFFFF				

# Onchip Breakpoints on Read/Write Accesses

Onchip breakpoints can be used to stop the core at a read or write access to a memory location.



# Onchip Breakpoints by Processor Architecture

Refer to your [Processor Architecture Manual](#) for a detailed list of the available Onchip breakpoints.

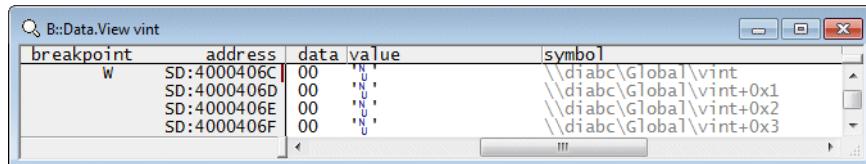
For some processor architectures Onchip breakpoints can only mark **single addresses** (e.g Cortex-A9). Most processor architectures, however, allow to mark **address ranges** with Onchip breakpoints. It is very common that one Onchip breakpoint marks the start address of the address range while the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

<b>Break.CONFIG.VarConvert</b> <b>ON</b>	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.
<b>Break.CONFIG.VarConvert</b> <b>OFF</b>	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.



```
; 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	N	\diabc\Global\vint
W	SD:4000406D	00	N	\diabc\Global\vint+0x1
W	SD:4000406E	00	N	\diabc\Global\vint+0x2
W	SD:4000406F	00	N	\diabc\Global\vint+0x3
	SD:40004070	00	N	\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
```

B::Break.List			
address	type	method	
C:20005524--20005537	write	ONCHIP	✓ <input checked="" type="checkbox"/> str2

```
Break.List /Onchip
```

B::Break.List /Onchip				
address	type	method	onchip	resource
C:20005520--20005537	write	ONCHIP	01	(vppuLong)--(str2+0x13)

## ETM Breakpoints for ARM or Cortex-A/R

ETM breakpoints extend the number of available breakpoints. Some Onchip breakpoints offered by ARM and Cortex-A/R cores provide restricted functionality. ETM breakpoints can help you to overcome some of these restrictions.

ETM breakpoints always show a break-after-make behavior with a rather large delay. Thus, use ETM breakpoints only if necessary.

	<b>Program Breakpoints</b>	<b>Read/Write Breakpoints</b>	<b>Data Value Breakpoints</b>
<b>ARM7 ARM9</b>	<p><b>Onchip breakpoints:</b> up to 2, but address range only as bit mask (Reduced to 1 if software breakpoints are used)</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> up to 2, but address range only as bit mask</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip Breakpoint:</b> up to 2, but address range only as bit mask</p> <p><b>ETM breakpoints:</b> up to 2 data value breakpoints for exact address ranges</p>
<b>ARM11</b>	<p><b>Onchip breakpoints:</b> 6, but only single addresses</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges possible</p>	<p><b>Onchip breakpoints:</b> 2, but only single addresses</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges possible</p>	<p><b>Onchip breakpoints:</b> no data value breakpoints possible</p> <p><b>ETM breakpoints:</b> up to 2 data value breakpoints for exact address ranges</p>
<b>Cortex-A5</b>	<p><b>Onchip breakpoints:</b> 3, but only single addresses</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> 2, but address range only as bit mask</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> no data value breakpoints possible</p> <p><b>ETM breakpoints:</b> up to 2 data value breakpoints for exact address ranges</p>
<b>Cortex-A7 Cortex-R7</b>	<p><b>Onchip breakpoints:</b> 6, but only single addresses</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> 4, but address range only as bit mask</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> no data value breakpoints possible</p> <p><b>ETM breakpoints:</b> up to 2 data value breakpoints for exact address ranges</p>
<b>Cortex-A8</b>	<p><b>Onchip breakpoints:</b> 6, but address range only as bit mask</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> 2, but address range only as bit mask</p> <p><b>ETM breakpoints:</b> up to 2 exact address ranges</p>	<p><b>Onchip breakpoints:</b> no data value breakpoints possible</p> <p><b>ETM breakpoints:</b> up to 2 data value breakpoints for exact address ranges</p>

	<b>Program Breakpoints</b>	<b>Read/Write Breakpoints</b>	<b>Data Value Breakpoints</b>
<b>Cortex-R4</b> <b>Cortex-R5</b>	<b>Onchip breakpoints:</b> 2..8, but address range only as bit mask  <b>ETM breakpoints:</b> up to 2 exact address ranges	<b>Onchip breakpoints:</b> 1..8, but address range only as bit mask  <b>ETM breakpoints:</b> up to 2 exact address ranges	<b>Onchip breakpoints:</b> no data value breakpoints possible  <b>ETM breakpoints:</b> up to 2 data value breakpoints for exact address ranges
<b>Cortex-A9</b> <b>Cortex-A15</b> <b>Cortex-A17</b>	<b>Onchip breakpoints:</b> 6, but only single addresses  <b>ETM breakpoints:</b> 2 exact address ranges	<b>Onchip breakpoints:</b> 4, but address range only as bit mask  <b>ETM breakpoints:</b> —	<b>Onchip breakpoints:</b> no data value breakpoints possible  <b>ETM breakpoints:</b> —

	<b>Program Breakpoints</b>	<b>Read/Write Breakpoints</b>	<b>Data Value Breakpoints</b>
<b>Cortex-A3x</b> <b>Cortex-A5x</b> <b>Cortex-A6x</b> <b>Cortex-A7x</b> Neoverse	<b>Onchip breakpoints:</b> 6, but only single addresses  <b>ETM breakpoints:</b> 2 exact address ranges (more on request)	<b>Onchip breakpoints:</b> 4, but address range only as bit mask  <b>ETM breakpoints:</b> —	<b>Onchip breakpoints:</b> no data value breakpoints possible  <b>ETM breakpoints:</b> —
<b>Cortex-R52</b>	<b>Onchip breakpoints:</b> 8, but only single addresses  <b>ETM breakpoints:</b> up to 2 exact address ranges	<b>Onchip breakpoints:</b> 8, but address range only as bit mask  <b>ETM breakpoints:</b> —	<b>Onchip breakpoints:</b> no data value breakpoints possible  <b>ETM breakpoints:</b> —

No ETM breakpoints are available for the Cortex-M family.

Please refer to the description of the [ETM.StoppingBreakPoints](#) command, if you want to use the ETM breakpoints.

## Breakpoint Types

---

TRACE32 PowerView provides the following breakpoint types for standard debugging.

Breakpoint Types	Possible Implementations
<b>Program</b>	Software (Default) Onchip
<b>Read, Write, ReadWrite</b>	Onchip (Default)

# Program Breakpoints

Set a Program breakpoint by a left mouse double-click to the instruction

```
[B::List]
Step Over Diverge Return Up Go Break Mode
addr/line source
674 { if ( flags[ i ] )
676     primz = i + i + 3;
677     k = i + primz;
678     while ( k <= SIZE )
680     {
681         flags[ k ] = FALSE;
682         k += primz;
683     }
684     anzahl++;
685 }
686 return anzahl;
687 }
```

The **red program breakpoint indicator** marks all code lines for which a Program breakpoint is set.

The program stops before the instruction marked by the breakpoint is executed (break before make).

Disable the Program breakpoint by a left mouse double-click to the red program breakpoint indicator.  
The program breakpoint indicator becomes grey.

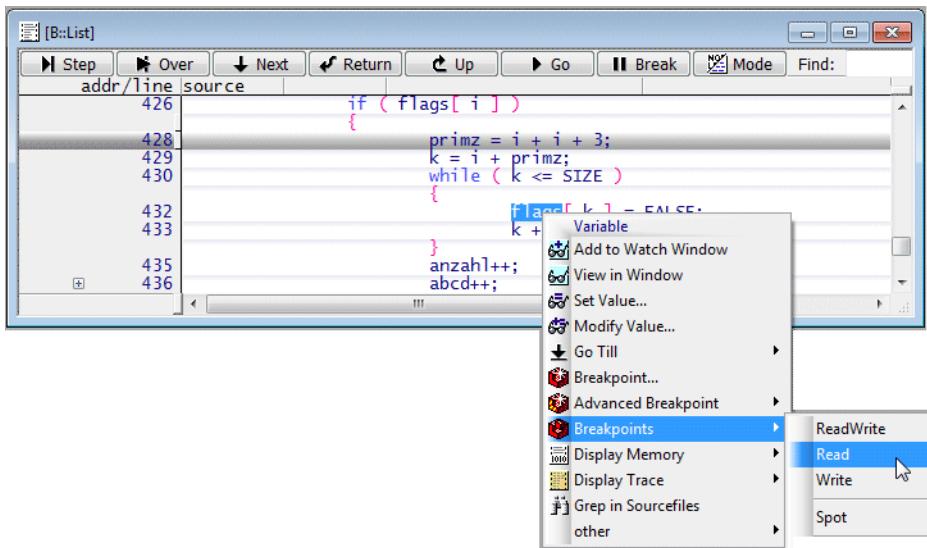
```
[B::List]
Step Over Next Return Up Go Break Mode
addr/line code Label mnemonic comment
426 { if ( flags[ i ] )
NSR:4A326590 E7DC1002 ldrb r1,[r12,+r2]
NSR:4A326594 E3510000 cmp r1,#0x0 ; r1,#0
NSR:4A326598 0A000016 beq 0xA3265F8
428 {
NSR:4A32659C E1A01082 lsl r1,r2,#0x1 ; r1,i,#1
NSR:4A3265A0 E2813003 add r3,r1,#0x3 ; r3,r1,#3
429 NSR:4A3265A4 E0821003 add r1,r2,r3 ; r1,i,primz
register int i, primz, k;
int anzahl;
```

**Break.Set <address> /Program [/DISABLE]**

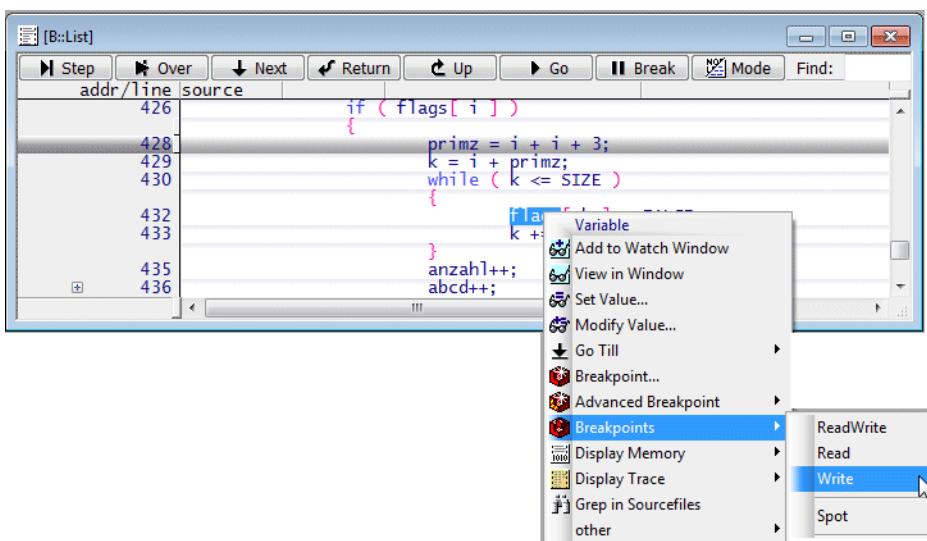
Set a Program breakpoint to the specified address.  
The Program breakpoint can be disabled if required.

```
Break.Set 0xA34f /Program ; set a Program breakpoint to  
; address 0xA34f  
  
Break.Set func1 /Program ; set a Program breakpoint to the  
; entry of func1  
; (first address of function func1)  
  
Break.Set func1+0x1c /Program ; set a Program breakpoint to the  
; instruction at address  
; func1 plus 28 bytes  
; (assuming that byte is the  
; smallest addressable unit)  
  
Break.Set func11\7 ; set a Program breakpoint to the  
; 7th line of code of the function  
; func11  
; (line in compiled program)  
  
Break.Set func17 /Program /DISable ; set a Program breakpoint to the  
; entry of func17  
; disable Program breakpoint  
  
Break.List ; list all breakpoints
```

# Read/Write Breakpoints

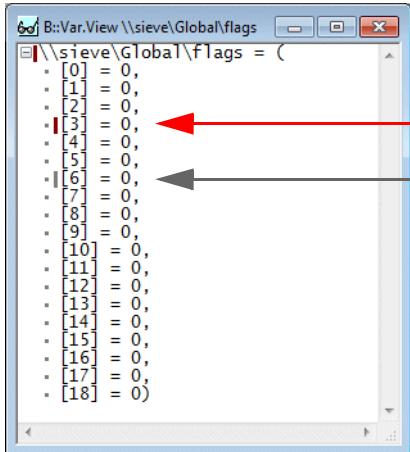


Core stops at  
a read access  
to the variable



Core stops at  
a write access  
to the variable

On most core(s) the program stops after the read or write access (break after make).



If an HLL variable is displayed, a small **red breakpoint indicator** marks an active Read/Write breakpoint.

A small **grey breakpoint indicator** marks a disabled Read/Write breakpoint.

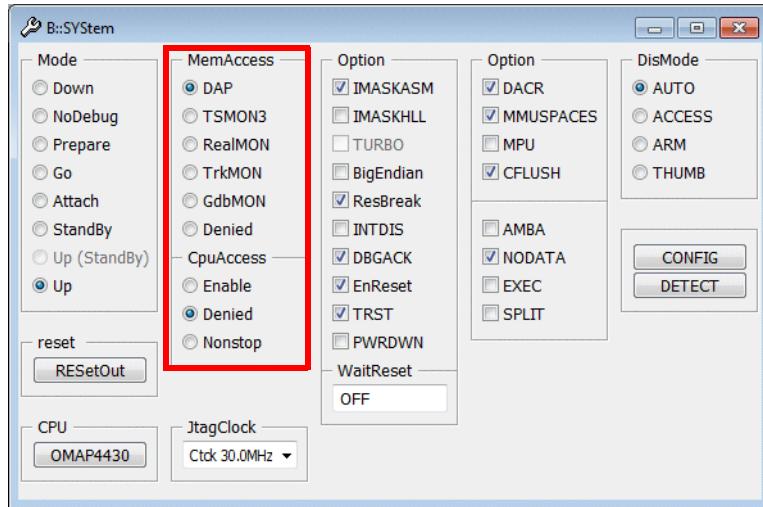
**Break.Set <address> | <range> /Read | /Write | /ReadWrite [/DISable]**

; allow HLL expression to specify breakpoint

**Var.Break.Set <hll\_expression> /Read | /Write | /ReadWrite [/DISable]**

```
Break.Set 0xB56 /Read  
Break.Set ast /Write  
Break.Set vpchar+5 /ReadWrite /DISable  
Var.Break.Set flags /Write  
Var.Break.Set flags[3] /Read  
Var.Break.Set ast->count /ReadWrite /DISable  
Break.List
```

## Breakpoint Setting at Run-time



### Software breakpoints

- If **MemAccess** Enable/NEXUS/DAP is enabled, Software breakpoints can be set while the core(s) is executing the program. Please be aware that this is not possible if an instruction cache and an MMU is used.
- If **CpuAccess** is enabled, Software breakpoints can be set while the core(s) is executing the program. If the breakpoint is set via CpuAccess the real-time behavior is influenced.
- If **MemAccess** and **CpuAccess** is Denied Software breakpoints can only be set when the program execution is stopped.

The behavior of **Onchip breakpoints** is core dependent. E.g. on all ARM/Cortex cores Onchip breakpoints can be set while the program execution is running.

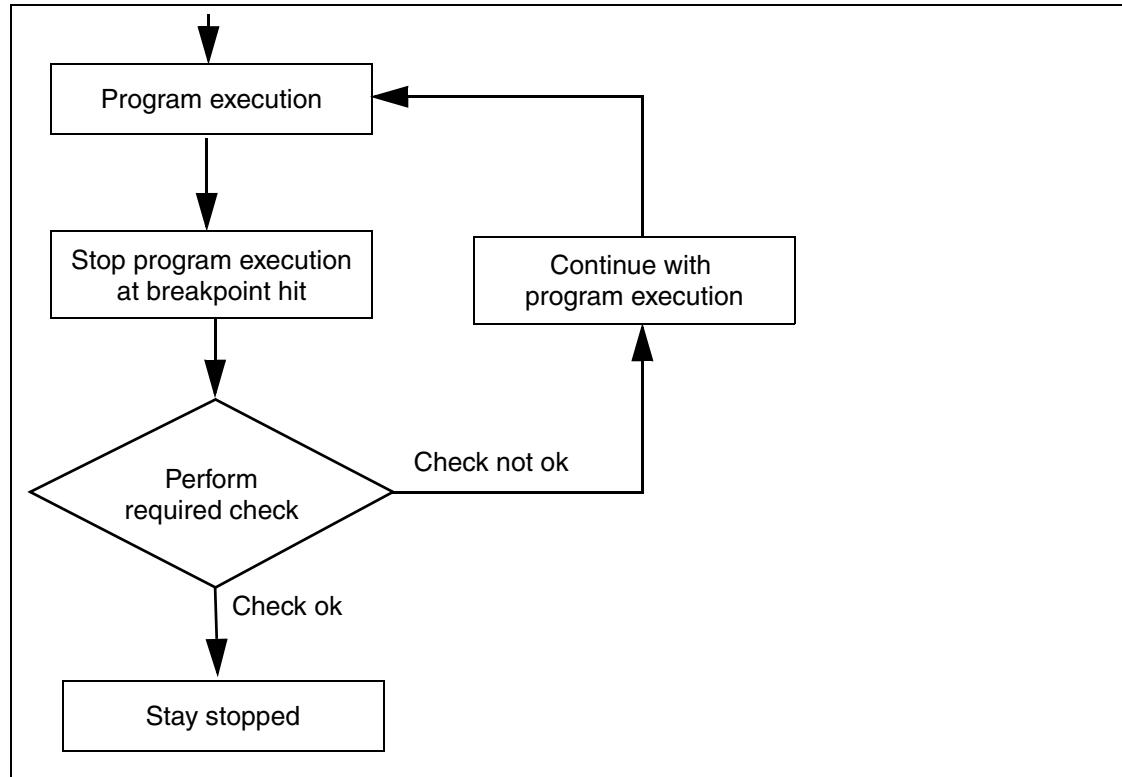
# Real-time Breakpoints vs. Intrusive Breakpoints

TRACE32 PowerView offers in addition to the basic breakpoints (Program/Read/Write) also complex breakpoints. Whenever possible these breakpoints are implemented as real-time breakpoints.

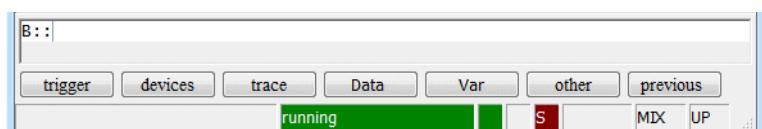
**Real-time breakpoints** do not disturb the real-time program execution on the core(s), but they require a complex on-chip break logic.

If the on-chip break logic of a core does not provide the required features or if Software breakpoints are used, TRACE32 has to implement an intrusive breakpoint.

Intrusive breakpoint perform as follows:



Each stop to perform the check suspends the program execution for at least 1 ms. For details refer to "["StopAndGo Mode"](#)" (glossary.pdf)



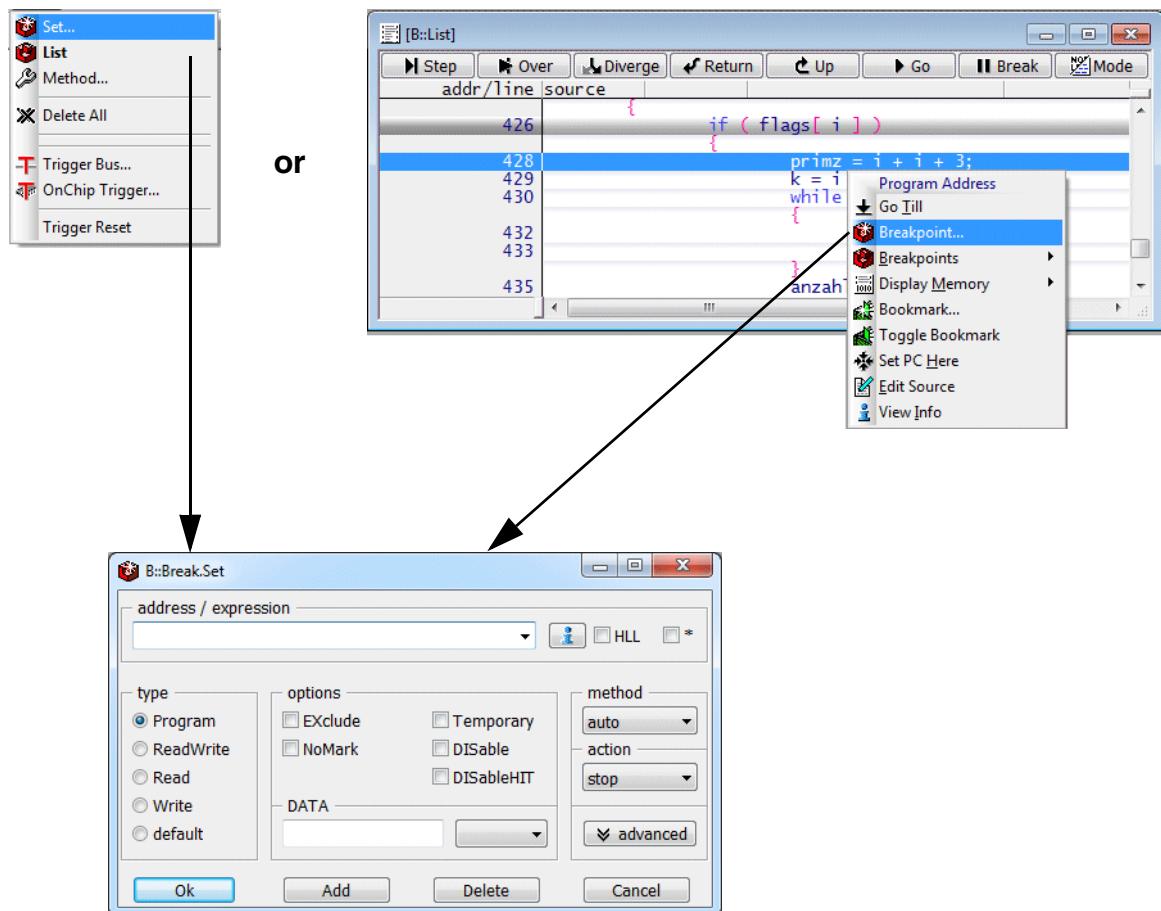
The (short-time) display of a red S in the state line indicates that an intrusive breakpoint was hit.

Intrusive breakpoints are marked with a special breakpoint indicator:



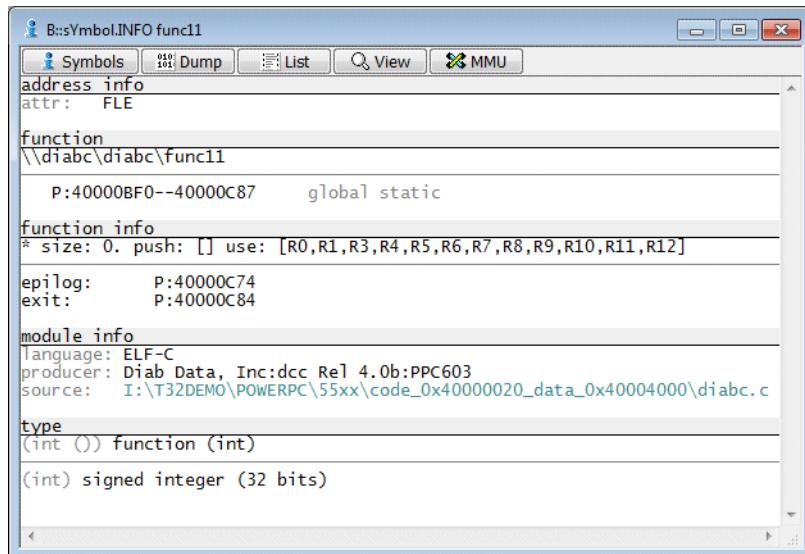
# Break.Set Dialog Box

There are two standard ways to open a **Break.Set** dialog.



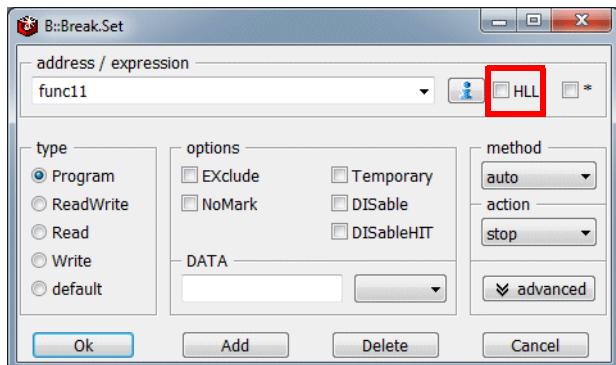
# The HLL Check Box - Function Name

```
sYmbol.INFO func11 ; display symbol information  
; for function func11
```



## Function Name/HLL Check Box OFF

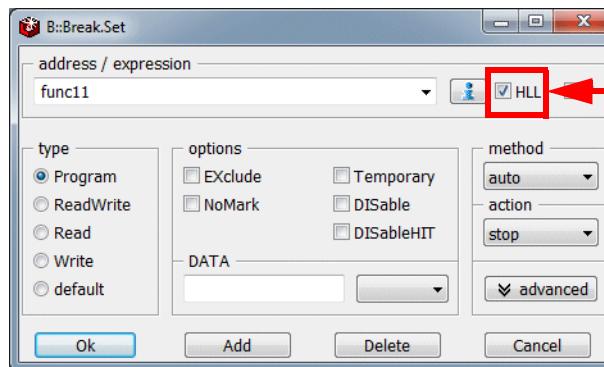
Program breakpoint is set to the function entry (first address of the function).



B::Break.List			
<input type="checkbox"/> Delete All	<input type="radio"/> Disable All	<input checked="" type="radio"/> Enable All	<input type="checkbox"/> Init
address	type	method	
F:40000BF0	Program	SOFT	<input checked="" type="checkbox"/> Func11

```
Break.Set func11
```

- If the on-chip break logic supports ranges for Program breakpoints, a Program breakpoint implemented as Onchip is set to the full address range covered by the function.
- If the on-chip break logic provides only bitmasks to realizes breakpoints on instruction ranges, a Program breakpoint implemented as Onchip is set by using the smallest bitmask that covers the complete address range of the function.
- otherwise this breakpoint is rejected with an error message.

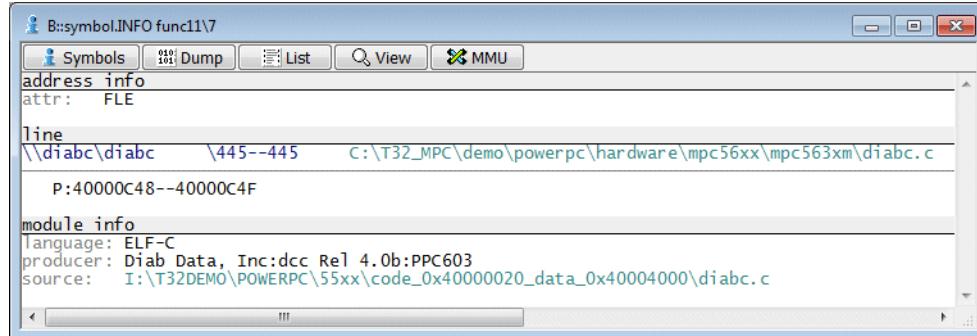


B::Break.List			
address	type	method	
F:40000BF0--40000C87	Program	ONCHIP	✓ <input checked="" type="checkbox"/> func11

```
Var.Break.Set func11
```

# The HLL Check Box - Program Line Number

```
sYmbol.INFO func11\7 ; display debug information  
; for 7th program line in  
; function func11
```



## Program Line Number/HLL Check Box OFF

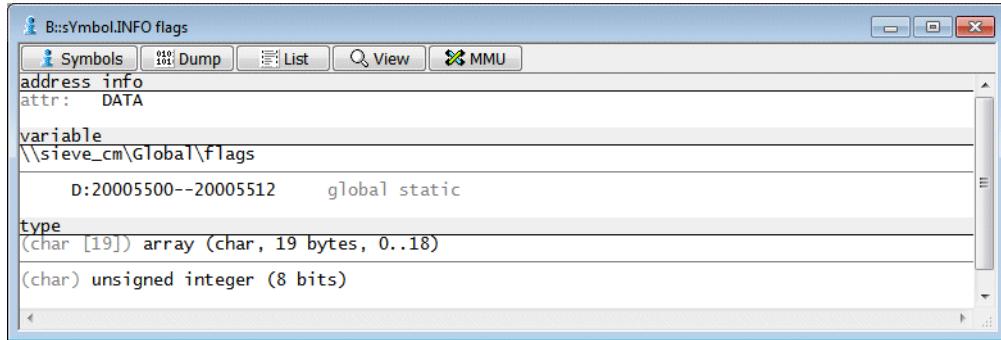
Program breakpoint is set to the first assembler instruction generated for the program line number.

This screenshot shows two windows related to breakpoints. The top window is 'B::Break.Set' with the address/expression set to 'func11\7'. The 'HLL' checkbox is highlighted with a red box. The bottom window is 'B::Break.List', showing a single entry for a program breakpoint at address F:40000C48 with type 'Program' and method 'SOFT'. The 'HLL' checkbox is also present here.

```
Break.Set func11\7
```

# The HLL Check Box - Variable

```
sYmbol.INFO flags ; display symbol information  
; for variable flags
```



## Variable/HLL Check Box OFF

Selected breakpoint (ReadWrite/Read/Write) is set to the start address of the variable.

The top window is "B::Break.Set". It has fields for "address / expression" (set to "flags"), "type" (radio buttons for Program, ReadWrite, Read, Write, default, with Write selected), "options" (checkboxes for EXclude, Temporary, NoMark, DISable, DISableHIT, with EXclude checked), "method" (dropdown menu set to "auto"), and "action" (dropdown menu set to "stop"). A red box highlights the "HLL" checkbox, which is unchecked. Buttons at the bottom include "Ok", "Add", "Delete", and "Cancel".

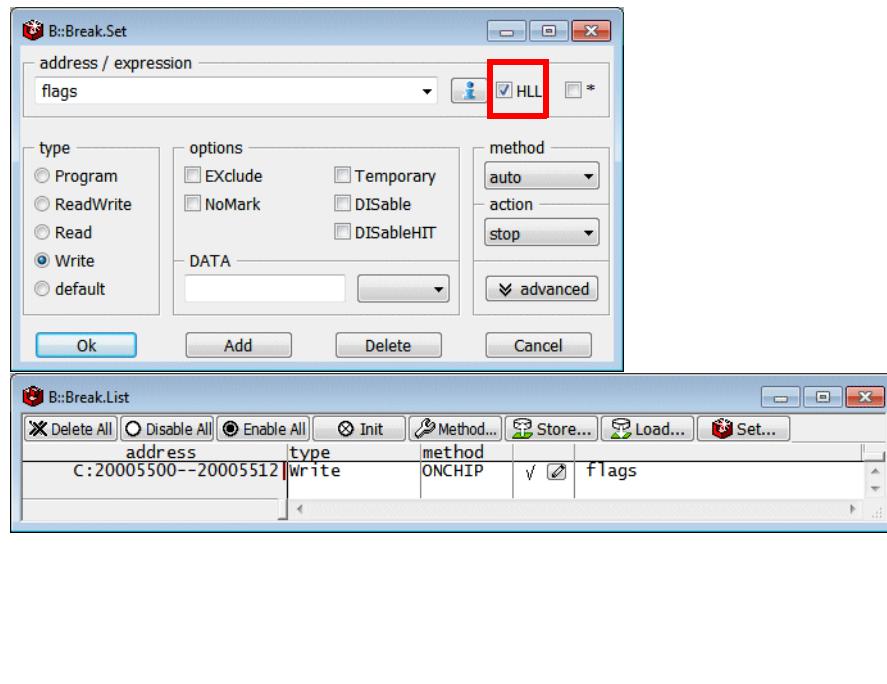
The bottom window is "B::Break.List". It has buttons for "Delete All", "Disable All", "Enable All", "Init", "Method...", "Store...", "Load...", and "Set...". A table lists breakpoints:

address	type	method	Flags
C:20005500	Write	ONCHIP	✓

```
Break.Set flags
```

## Variable/HLL Check Box ON

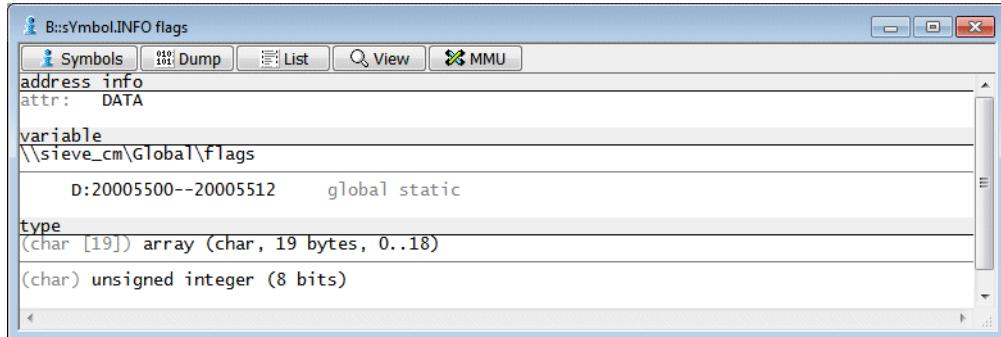
- If the on-chip break logic supports ranges for Read/Write breakpoints, the specified breakpoint is set to the complete address range covered by the variable.
- If the on-chip break logic provides only bitmasks to realizes Read/Write breakpoints on address ranges, the specified breakpoint is set by using the smallest bitmask that covers the address range used by the variable.



```
Var.Break.Set flags
```

# The HLL Check Box - HLL Expression

```
sYmbol.INFO flags ; display symbol information  
; for variable flags
```



## Variable/HLL Check Box Must Be ON

If you want to use an HLL expression to specify the address range for a Read/Write breakpoint, the HLL check box has to be checked.

- If the on-chip break logic supports ranges for Read/Write breakpoints, the specified breakpoint is set to the complete address range covered by the HLL expression.
- If the on-chip break logic provides only bitmasks to realizes Read/Write breakpoints on address ranges, the specified breakpoint is set by using the smallest bitmask that covers the address range used by the HLL expression.

The top window is titled "B::Break.Set". It has a dropdown menu "address / expression" containing "flags[3]". To its right is a button with a person icon and a checked checkbox labeled "HLL". Below this are sections for "type" (radio buttons for Program, ReadWrite, Read, Write, default), "options" (checkboxes for EXclude, Temporary, NoMark, DISable, DISableHIT), and "method" (dropdown menus for auto, action, stop, advanced). At the bottom are buttons for Ok, Add, Delete, and Cancel.

The bottom window is titled "B::Break.List". It has a toolbar with buttons for Delete All, Disable All, Enable All, Init, Method..., Store..., Load..., Set..., and a search bar. A table lists a single breakpoint: address C:20005503, type write, method ONCHIP, and flags fTags[3].

```
Var.Break.Set flags[3]
```

## Allow Wildcards in address/expression

Set Program breakpoints the all function that match the defined name pattern.

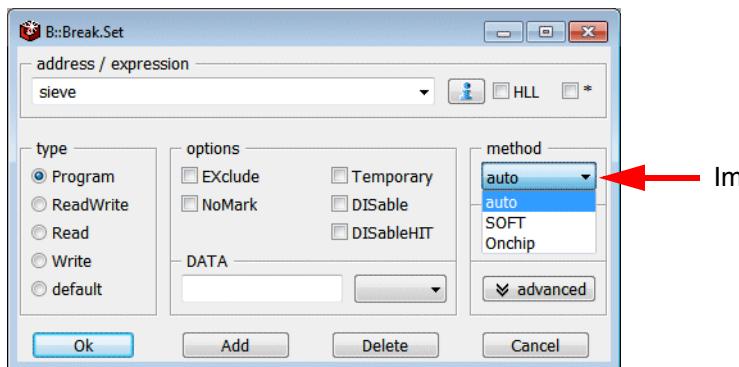
The screenshot shows two windows related to breakpoint management:

- B::Break.Set Dialog:** This dialog is used to set a new breakpoint. The "address / expression" field contains "func2\*". A red arrow points to the checkbox labeled "\*", which is checked. Other settings include "Program" selected in the "type" group, "SOFT" in the "method" dropdown, and "stop" in the "action" dropdown. Buttons for "Ok", "Add", "Delete", and "Cancel" are at the bottom.
- B::Break.List Window:** This window displays a list of existing breakpoints. The columns are "address", "type", "method", and "name". The "name" column lists various functions starting with "func2", such as "func2a", "func2b", "func2c", etc. The "method" column shows mostly "SOFT".

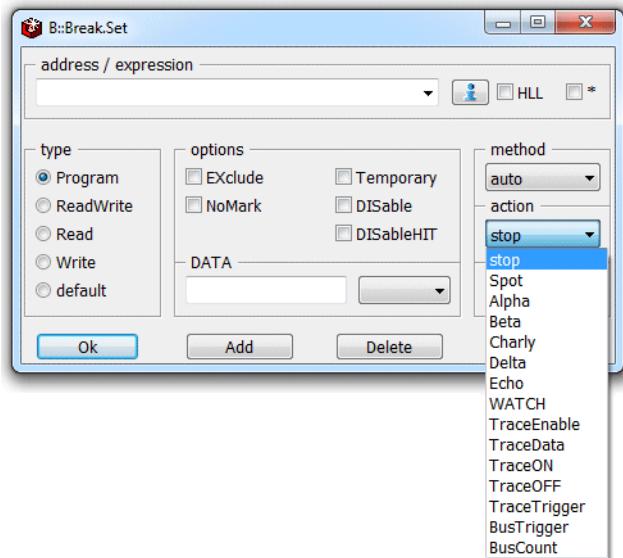
Requires sufficient resources if Onchip breakpoints are used.

```
Break.SetPattern func2*
```

# Implementations



Implementation	
<b>auto</b>	Use breakpoint implementation as predefined in TRACE32 PowerView.
<b>SOFT</b>	Implement breakpoint as Software breakpoint.
<b>Onchip</b>	Implement breakpoint as Onchip breakpoint.



By default the program execution is stopped when a breakpoint is hit (action **stop**). TRACE32 PowerView provides the following additional reactions on a breakpoint hit:

Action (debugger)	
<b>Spot</b>	The program execution is stopped shortly at a breakpoint hit to update the screen. As soon as the screen is updated, the program execution continues.
<b>Alpha</b>	Set an Alpha breakpoint.
<b>Beta</b>	Set a Beta breakpoint.
<b>Charly</b>	Set a Charly breakpoint.
<b>Delta</b>	Set a Delta breakpoint.
<b>Echo</b>	Set an Echo breakpoint.
<b>WATCH</b>	Trigger the debug pin at the specified event (not available for all processor architectures).

Alpha, Beta, Charly, Delta and Echo breakpoint are only used in very special cases. For this reason no description is given in the general part of the training material.

<b>Action (on-chip or off-chip trace)</b>	
<b>TraceEnable</b>	Advise on-chip trace logic to generate trace information on the specified event.
<b>TraceON</b>	Advise on-chip trace logic to start with the generation of trace information at the specified event.
<b>TraceOFF</b>	Advise on-chip trace logic to stop with the generation of trace information at the specified event.
<b>TraceTrigger</b>	Advise on-chip trace logic to generate a trigger at the specified event. TRACE32 PowerView stops the recording of trace information when a trigger is detected.

A detailed description for the Actions (on-chip and off-chip trace) can be found in the following manuals:

- [“Training Arm ETM”](#) (training\_arm\_etm.pdf).
- [“Training Cortex-M Trace”](#) (training\_cortexm\_etm.pdf).
- [“Training AURIX Trace”](#) (training\_aurix\_trace.pdf).
- [“Training Hexagon-ETM”](#) (training\_hexagon\_etm.pdf).
- [“Training Nexus”](#) (training\_nexus.pdf).

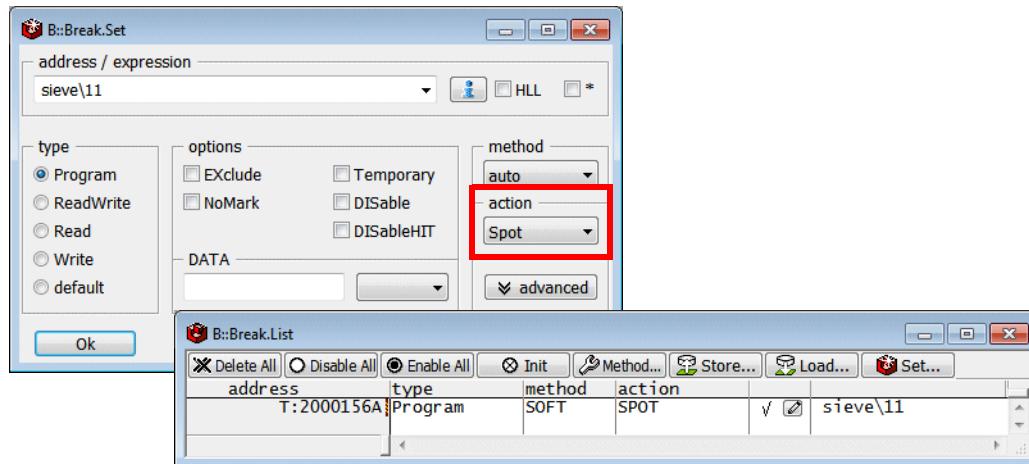
or with the description of the **Break.Set** command.

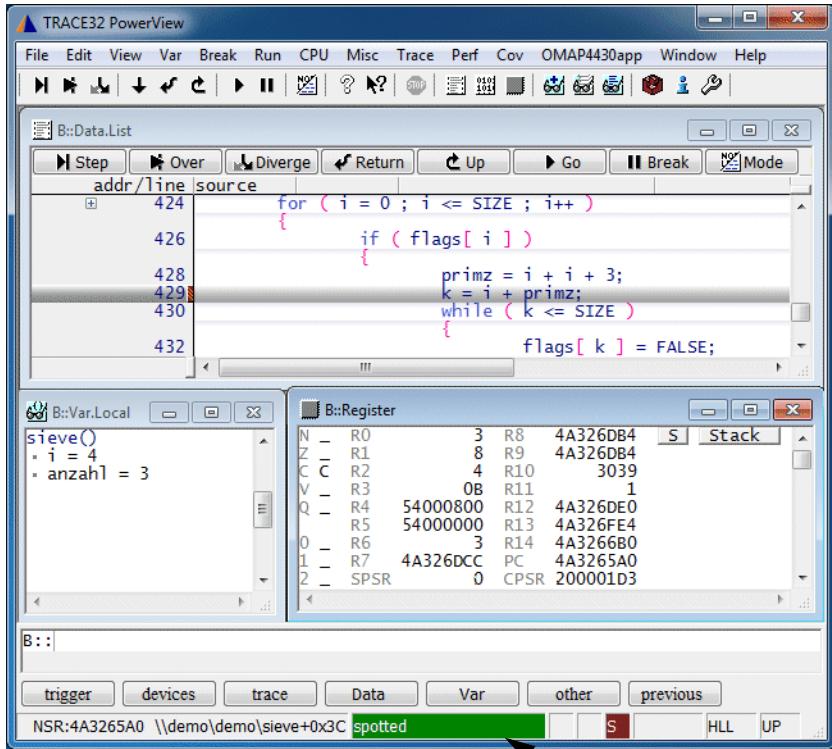
## Example for the Action Spot

The information displayed within TRACE32 PowerView is by default only updated, when the core(s) stops the program execution.

The action Spot can be used to turn a breakpoint into a watchpoint. The core stops the program execution at the watchpoint, updates the screen and restarts the program execution automatically. Each stop takes **50 ... 100 ms** depending on the speed of the debug interface and the amount of information displayed on the screen.

**Example:** Update the screen whenever the program executes the instruction sieve\11.

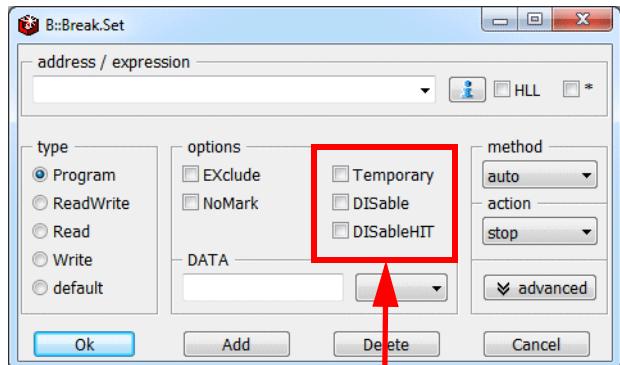




spotted indicates a breakpoint with the action Spot

```
Break.Set sieve\11 /Spot
```

# Options



Options

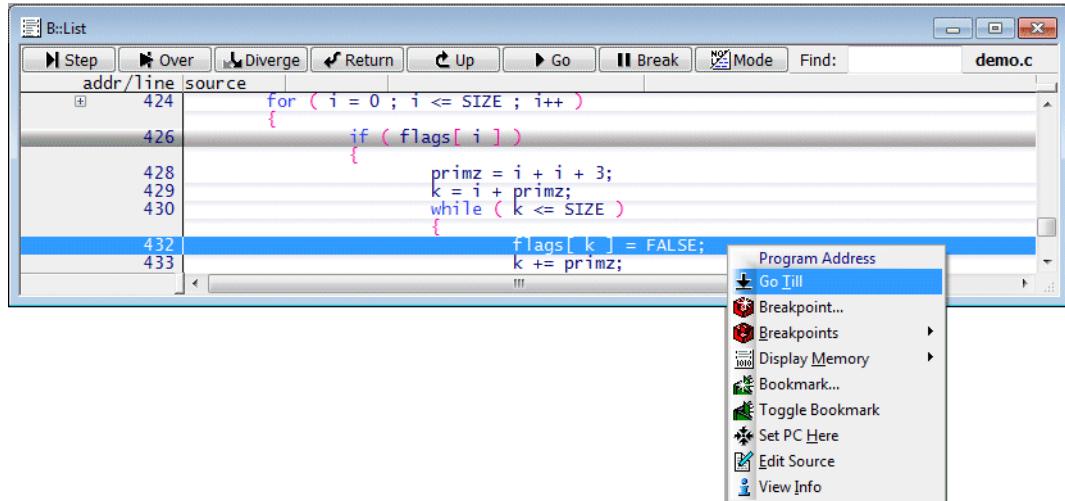
<b>Temporary</b>	<b>OFF:</b> Set a permanent breakpoint (default). <b>ON:</b> Set a temporary breakpoint. All temporary breakpoints are deleted the next time the core(s) stops the program execution.
<b>DISable</b>	<b>OFF:</b> Breakpoint is enabled (default). <b>ON:</b> Set breakpoint, but disabled.
<b>DISableHIT</b>	<b>ON:</b> Disable the breakpoint after the breakpoint was hit.

## Example for the Option Temporary

Temporary breakpoints are usually not set via the **Break.Set** dialog, but they are often used while debugging.

### Examples:

- **Go Till**

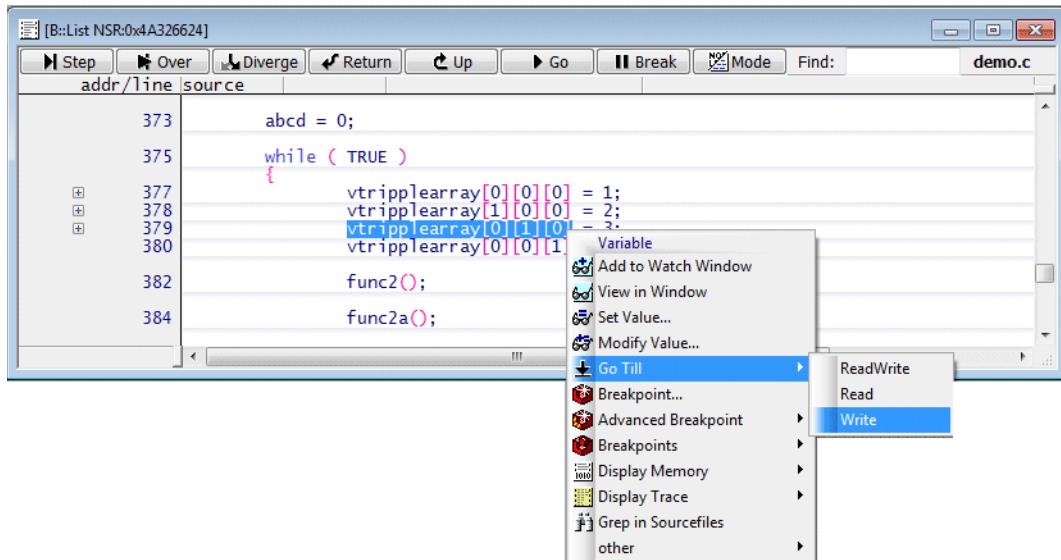


### Go <address> [ <address> ...]

```
; set a temporary Program breakpoint to
; the entry of the function func4
; and start the program execution
Go func4

; set a temporary Program breakpoints to
; the entries of the functions func4, func8 and func9
; and start the program execution
Go func4 func8 func9
```

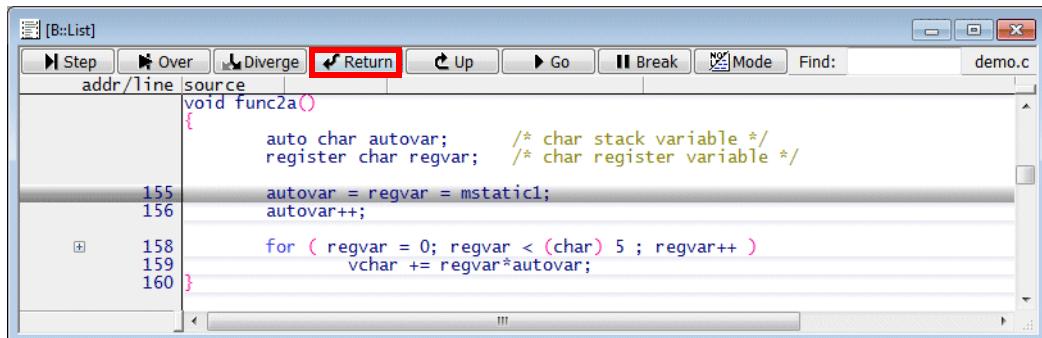
- Go Till -> Write



**Var.Go <hl\_expression> [/Write]**

```
; set a temporary write breakpoint to the variable
; vtripplearray[0][1][0] and start the program execution
Var.Go vtripplearray[0][1][0] /Write
```

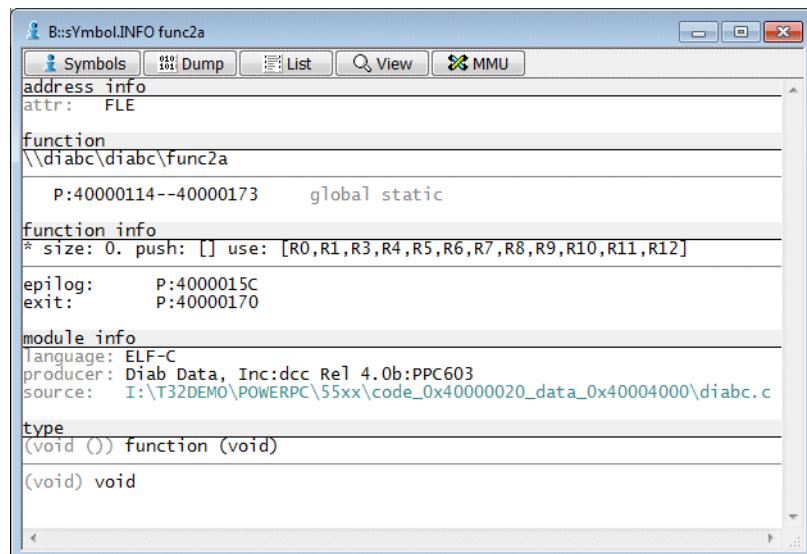
- **Go.Return and similar commands**



### Go.Return

```
; first Go.Return
; set a temporary breakpoint to the start of the function epilogue
; and start the program execution
Go.Return
; stopping at the function epilog first has the advantage that the
; local variables are still valid at this point.

; second Go.Return
; set a temporary breakpoint to the function return
; and start the program execution
Go.Return
```



## DATA Breakpoints

---

The DATA field offers the possibility to combine a Read/Write breakpoint with a specific data value.

- DATA breakpoints are implemented as real-time breakpoints if the core supports **Data Value Breakpoints** (for details on your core refer to “[Onchip Breakpoints by Processor Architecture](#)”, page 77).

TRACE32 PowerView indicates a real-time breakpoints by a full red bar.



TRACE32 PowerView allows inverted data values if this is supported by the on-chip break logic.

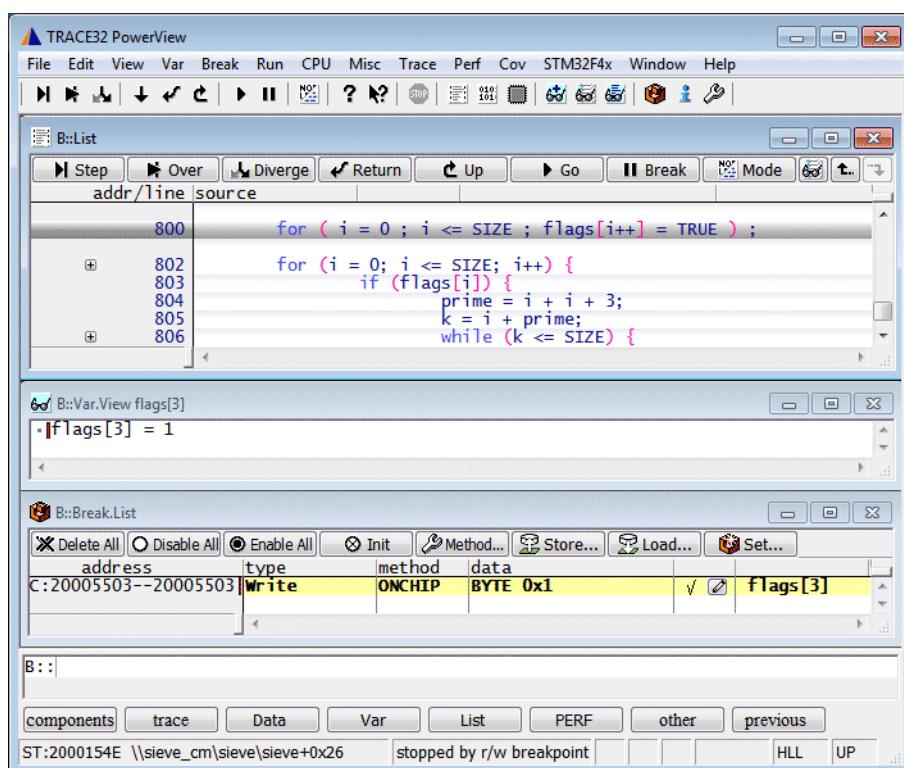
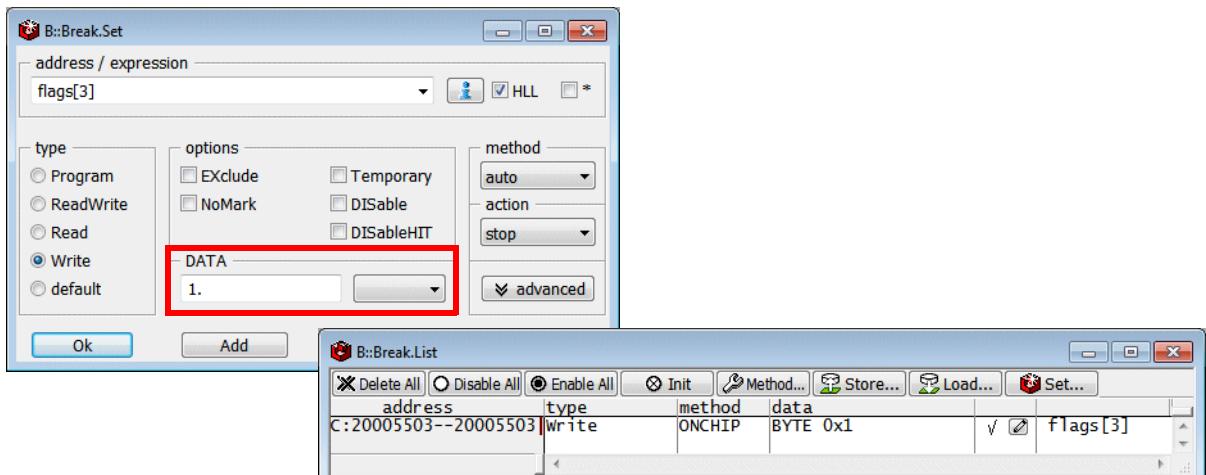
- DATA breakpoints are implemented as intrusive breakpoints if the core does not support Data Value Breakpoints. For details on the intrusive DATA breakpoints refer to the description of the [Break.Set](#) command.

TRACE32 PowerView indicates an intrusive breakpoint by a hatched red bar.



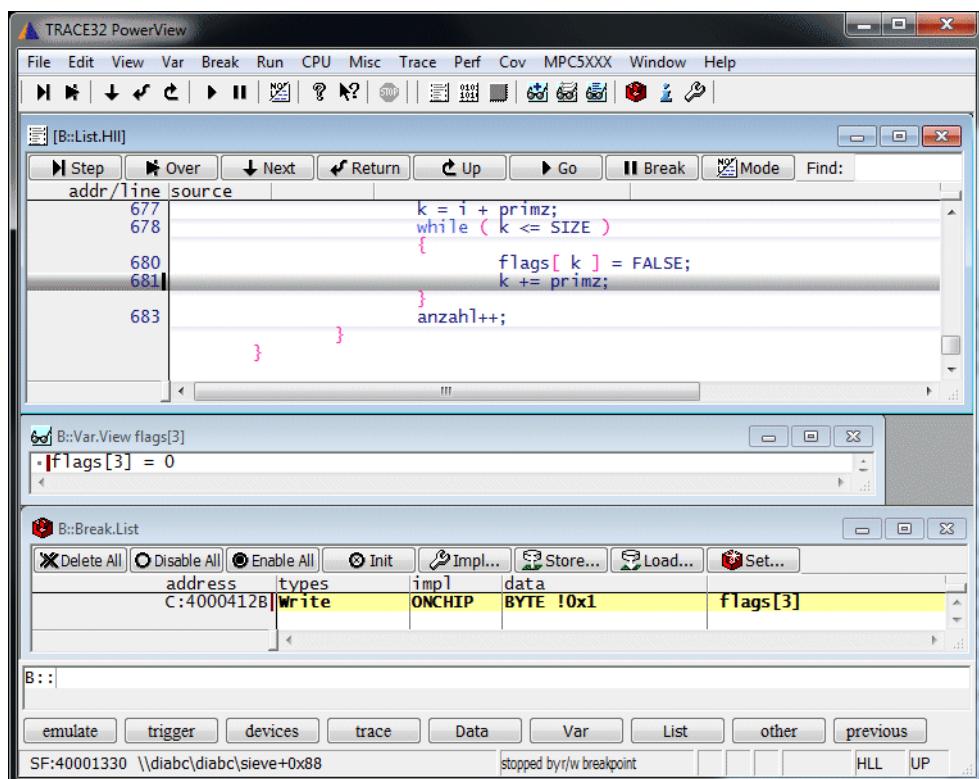
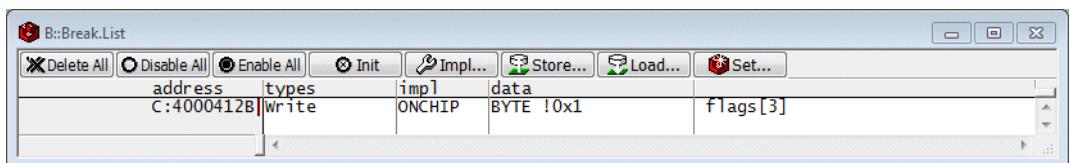
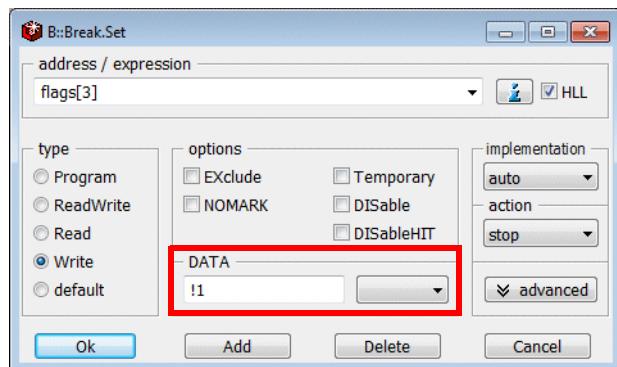
TRACE32 PowerView allows inverted data values for intrusive DATA breakpoints.

**Example:** Stop the program execution if a 1 is written to flags[3].



```
Var.Break.Set flags[3] /Write /DATA.auto 1.
```

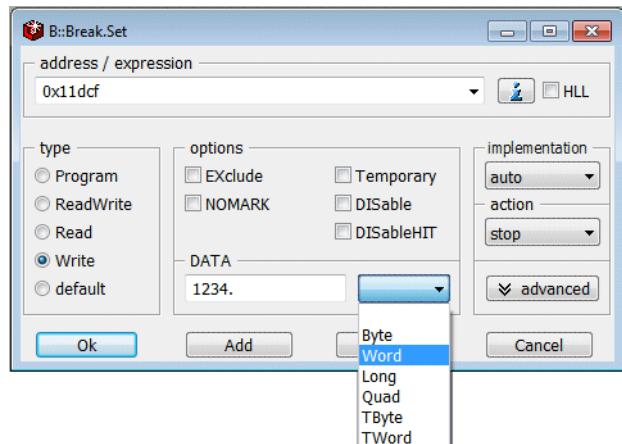
**Example:** Stop the program execution if another value than 1 is written to flag[3].



```
Var.Break.Set flags[3] /Write /DATA.auto !1.
```

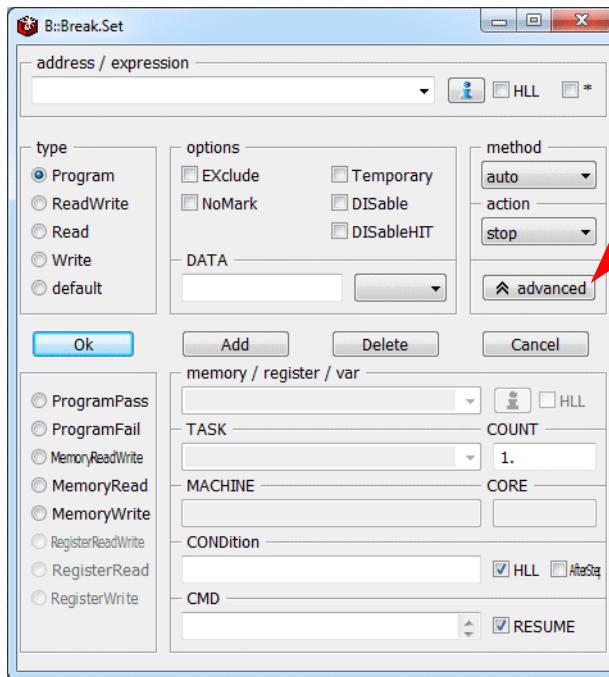
If an HLL expression is used TRACE32 PowerView gets the information if the data is written via a byte, word or long access from the symbol information.

If an address or symbol is used the user has to specify the access width, so that the correct number of bits is compared.



```
Break.Set 0x11dcf /Write /DATA.Word 1234.
```

# Advanced Breakpoints



If the **advanced** button is pushed additional input fields are provided

Advanced breakpoint input fields

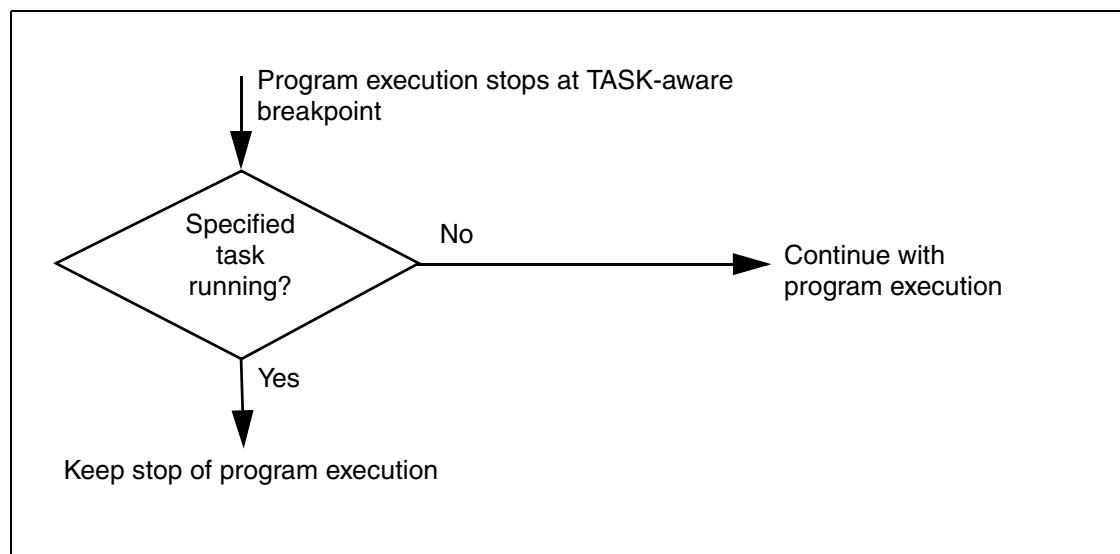
## TASK-aware Breakpoints

If OS-aware debugging is configured (refer to “[OS-aware Debugging](#)” in TRACE32 Glossary, page 31 (glossary.pdf)), TASK-aware breakpoints allow to stop the program execution at a breakpoint 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). For details on the real-time TASK-aware breakpoints refer to the description of the **Break.Set** command.

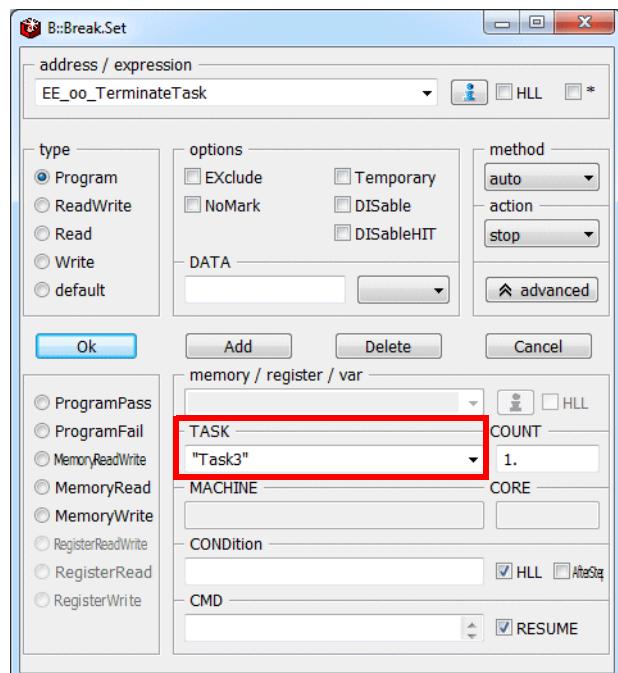
### Intrusive TASK-aware Breakpoint

Processing:



Each stop at the TASK-aware breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

**Example:** Stop the program execution at the entry to the function EE\_oo\_TerminateTask only if the task/process "Task3" is running.



B::Break.List						
address	type	method	task			
V:40001080	Program	SOFT	"Task3"	/	EE_oo_TerminateTask	

```
Break.Set EE_oo_TerminateTask /Program /TASK "Task3"
```

TRACE32 PowerView

File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX EE\_cpu\_0 Window Help

B:List.auto

Step	Over	Diverge	Return	Up	Go	Break	Mode	Find
addr/line	code	label	mnemonic	comment				
62		StatusType EE_oo_TerminateTask(void)						
SV:40001080	182106F0	EE_oo_Te..:e_stwu	r1,-0x10(r1) ; r1,-16(r1)					
SV:40001084	0080	se_mflr	r0					
		register EE_FREG np_flags;						
65		EE_ORTI_set_service_in(EE_SERVICETRACE_TERMINATETASK);						
SV:40001086	1C8D8080	e_add16i	r4,r13,-0x7F80 ; r4,r13,-32640					
		...	E_OS_RESOURCE if the task still occupy resources					
			E_OS_CALLLEVEL if called at interrupt level					

B:Break.List

X Delete All	O Disable All	Enable All	Init	Method...	Store...	Load...	Set...	
ress	type	method	task					
V:40001080	Program	SOFT	"(other)"					

B::

components trace Data Var List PERF SYStem other previous

running S MIX UP

The red S indicates that an intrusive breakpoint is used

TRACE32 PowerView

File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX EE\_cpu\_0 Window Help

B:List.auto

Step	Over	Diverge	Return	Up	Go	Break	Mode	Find
addr/line	code	label	mnemonic	comment				
62		StatusType EE_oo_TerminateTask(void)						
SV:40001080	182106F0	EE_oo_Te..:e_stwu	r1,-0x10(r1) ; r1,-16(r1)					
SV:40001084	0080	se_mflr	r0					
		register EE_FREG np_flags;						
65		EE_ORTI_set_service_in(EE_SERVICETRACE_TERMINATETASK);						
SV:40001086	1C8D8080	e_add16i	r4,r13,-0x7F80 ; r4,r13,-32640					
		...	E_OS_RESOURCE if the task still occupy resources					
			E_OS_CALLLEVEL if called at interrupt level					

B:Break.List

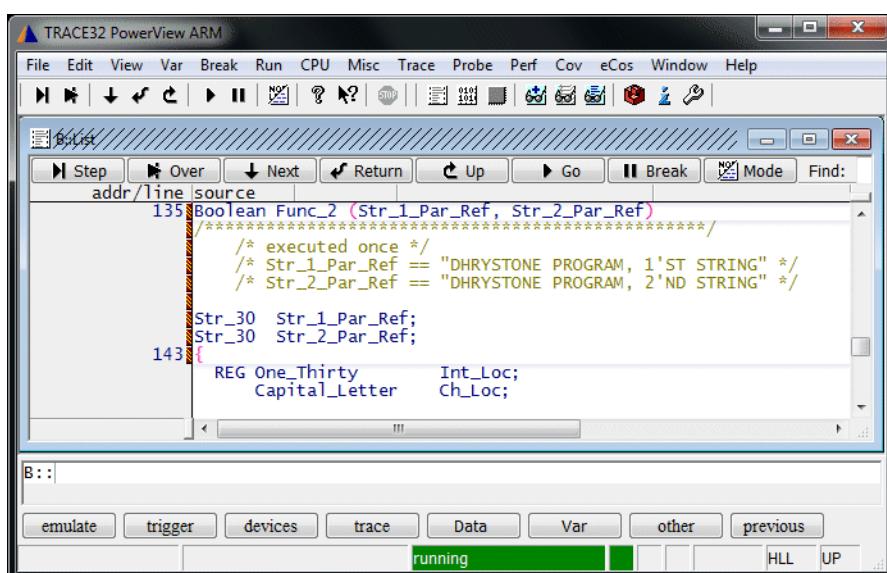
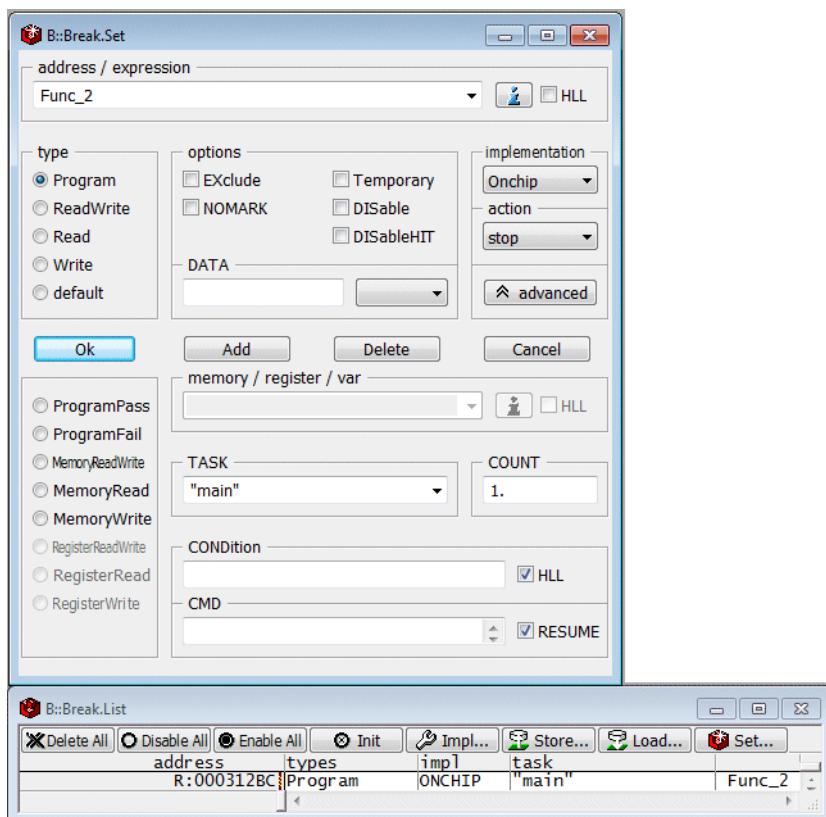
X Delete All	O Disable All	Enable All	Init	Method...	Store...	Load...	Set...	
ress	type	method	task					
V:40001080	Program	SOFT	"(other)"					

B::

components trace Data Var List PERF SYStem other previous

SV:40001080 \ppcie\_terminatE (other) stopped at breakpoint MIX UP

Example for ARM9: Stop the program execution at the entry to the function Func\_2 only if the taskF "main" is running (Onchip breakpoint).

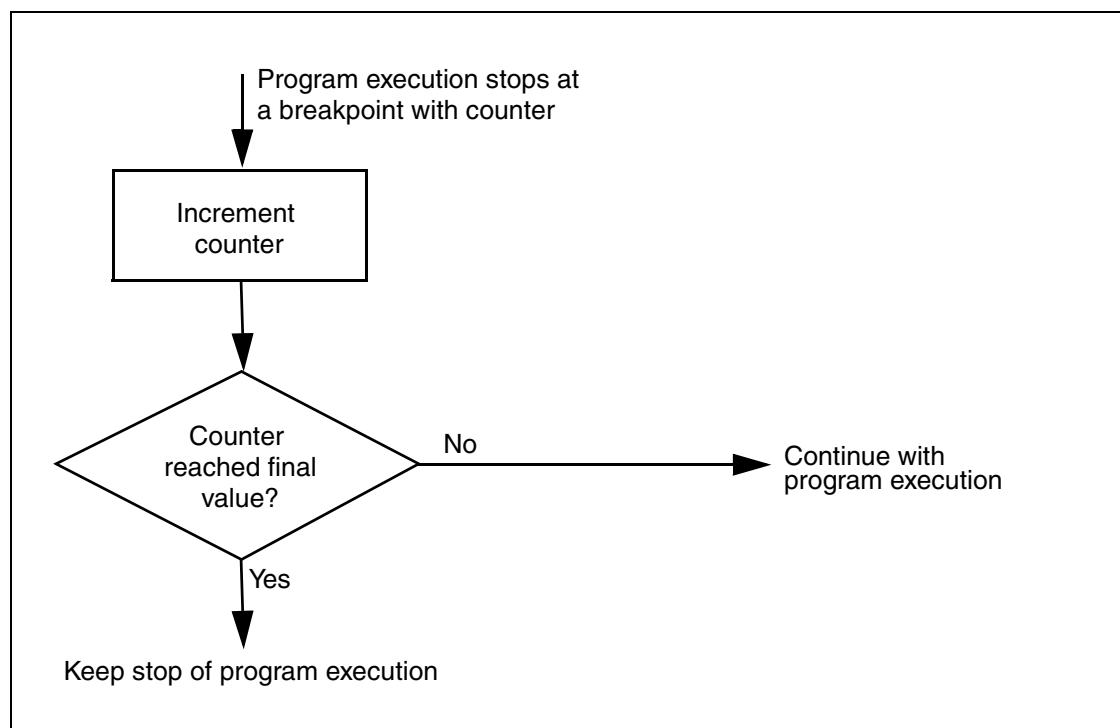


Counters allow to stop the program execution on the *n th* hit of a breakpoint.

## Software Counter

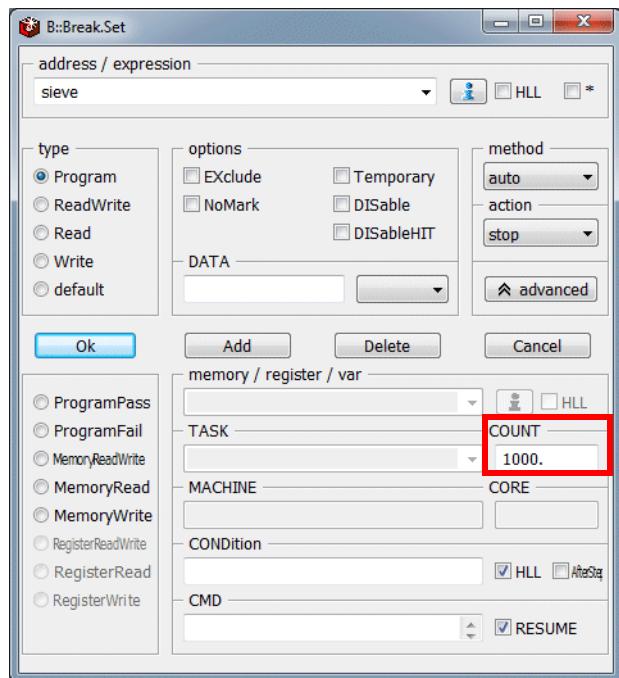
If the on-chip break logic of the core does not provide counters or if a Software breakpoint is used, counters are implemented as software counters.

Processing:



Each stop at a Counter breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

**Example:** Stop the program execution after the function sieve was entered 1000. times.



B::Break.List					
address	type	method	count		
F:400012A8	Program	SOFT	0./1000.	✓	<input checked="" type="checkbox"/>
					sieve

```
Break.Set sieve /COUNT 1000.
```

The current counter value is displayed in the **Break.List** window

address	type	method	count
F:400012A8	Program	SOFT	75./1000.

B::: components trace Data Var List PERF other previous running S HLL UP

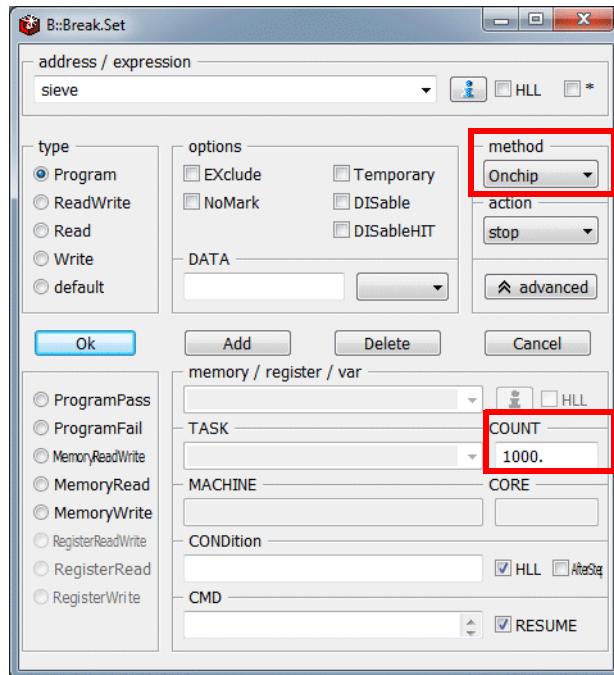
The red S indicates an intrusive breakpoint

address	type	method	count
F:400012A8	Program	SOFT	0./1000.

B::: components trace Data Var List PERF other previous SF:400012A8 \\\diabc\diabc\sieve stopped at breakpoint HLL UP

The on-chip break logic of some cores e.g. MPC55xx provides counters. They are used together with Onchip breakpoints.

**Example:** Stop the program execution after the function sieve was entered 1000. times.



B::Break.List					
address	type	method	count	store	load
F:400012A8	Program	ONCHIP	0./1000.	✓	✗
				sieve	

```
Break.Set sieve /COUNT 1000. /Onchip
```

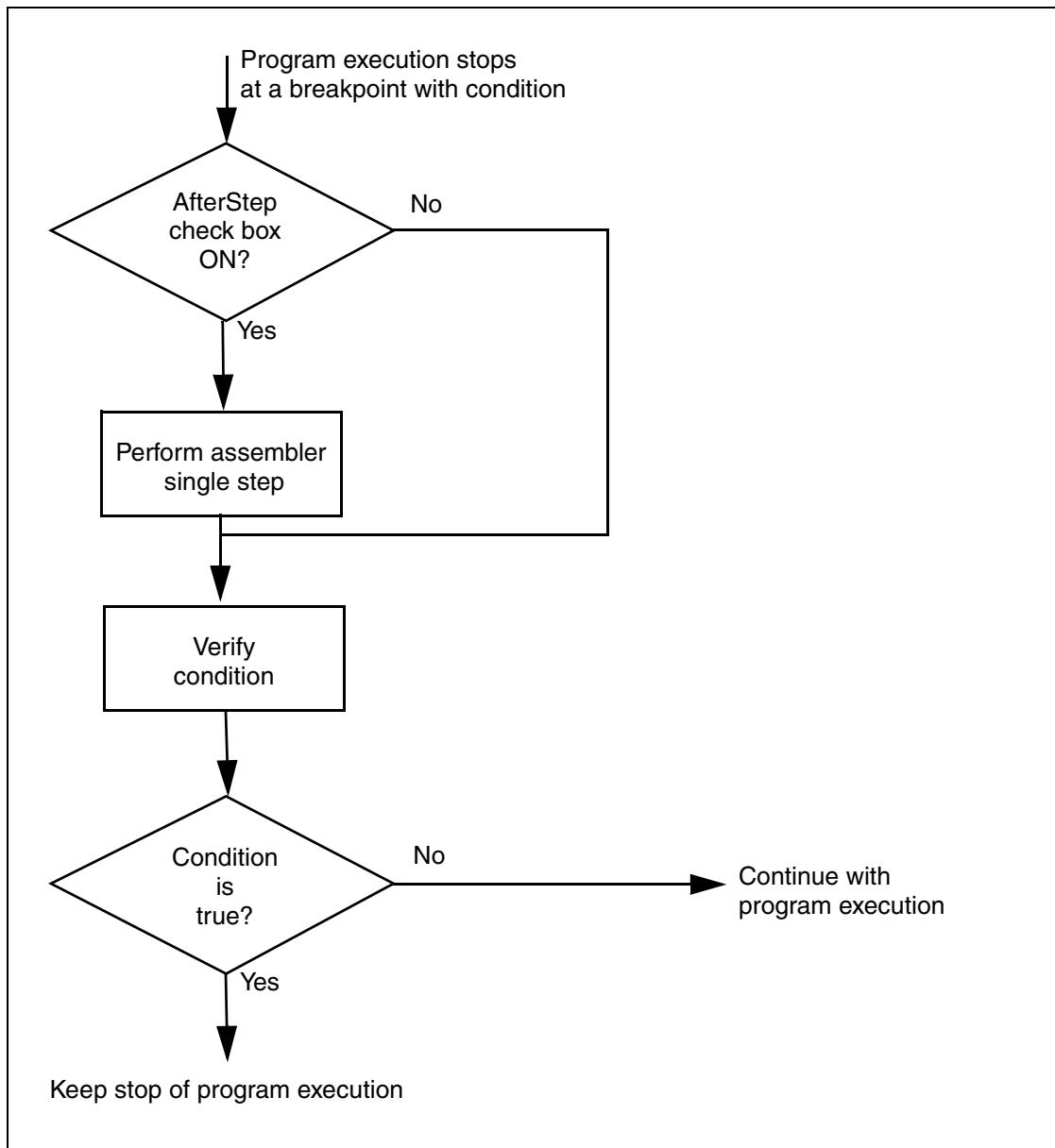
The counters run completely in real-time. No current counter value can be displayed while the program execution is running. As soon as the counter reached its final value, the program execution is stopped.

## CONDition

The program execution is stopped at the breakpoint only if the specified condition is true.

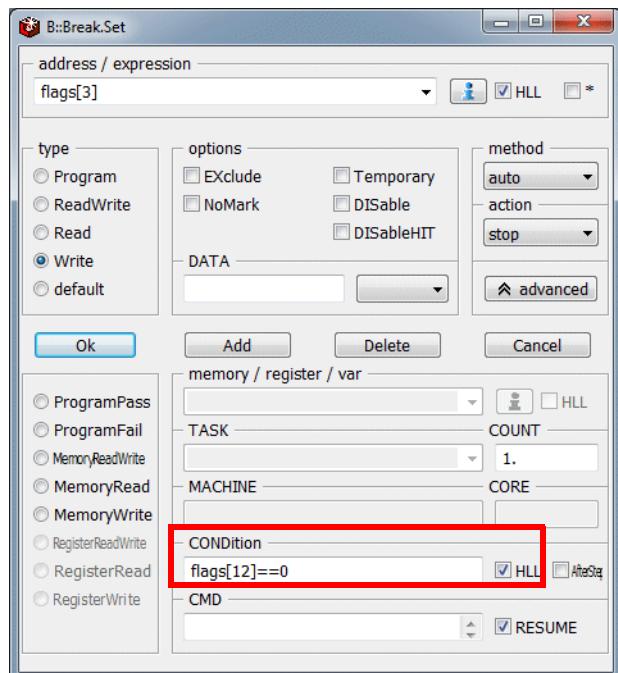
CONDition breakpoints are always intrusive.

Processing:

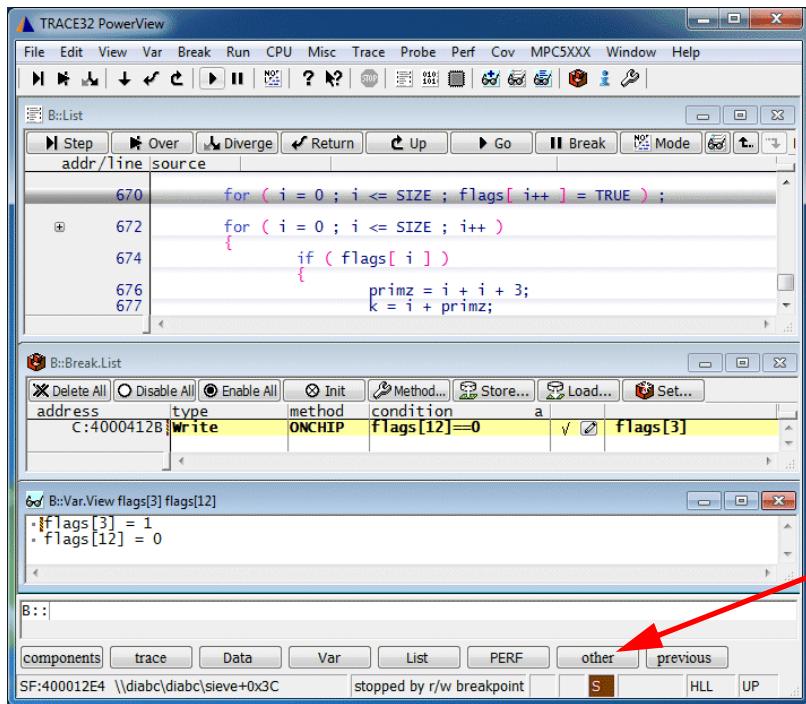


Each stop at a CONDition breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

**Example:** Stop the program execution on a write to flags[3] only if flags[12] is equal to 0 when the breakpoint is hit.



B::Break.List						
<input type="checkbox"/> Delete All	<input type="checkbox"/> Disable All	<input checked="" type="checkbox"/> Enable All	<input type="checkbox"/> Init	<input type="checkbox"/> Method...	<input type="checkbox"/> Store...	<input type="checkbox"/> Load...
address	type	method	condition	a	v	Set...
C:4000412B	Write	ONCHIP	flags[12]==0			

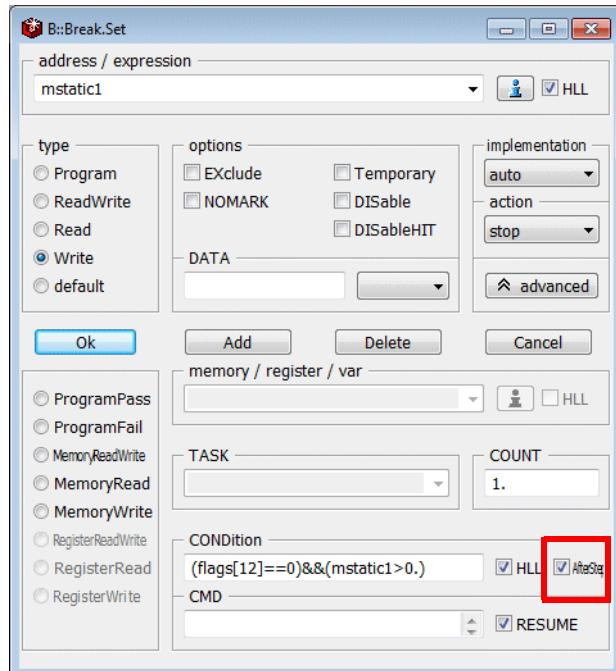


The red S indicates  
an intrusive breakpoint

## Example: “Break-before-make” Read/Write breakpoints only

Stop the program execution at a write access to the variable mstatic1 only if flags[12] is equal to 0 and mstatic1 is greater 0.

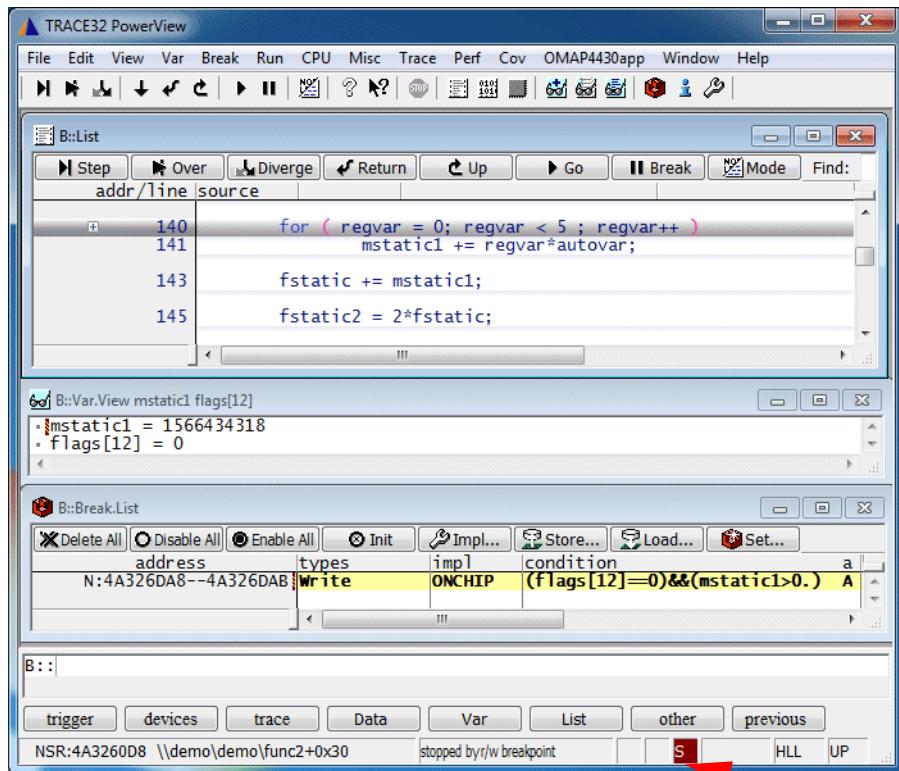
Perform an assembler single step because the processor architecture stops before the write access is performed.



AfterStep checked

B::Break.List				
<input type="checkbox"/> Delete All	<input type="radio"/> Disable All	<input type="radio"/> Enable All	<input type="radio"/> Init	<input type="radio"/> Impl...
address	types	impl	condition	
N:4A326DA8--4A326DAB	Write	ONCHIP	(flags[12]==0)&&(mstatic1>0.)	A

```
Var.Break.Set mstatic1 /Write /VarCondition (flags[12]==0)&&(mstatic1>0)
/AfterStep
```

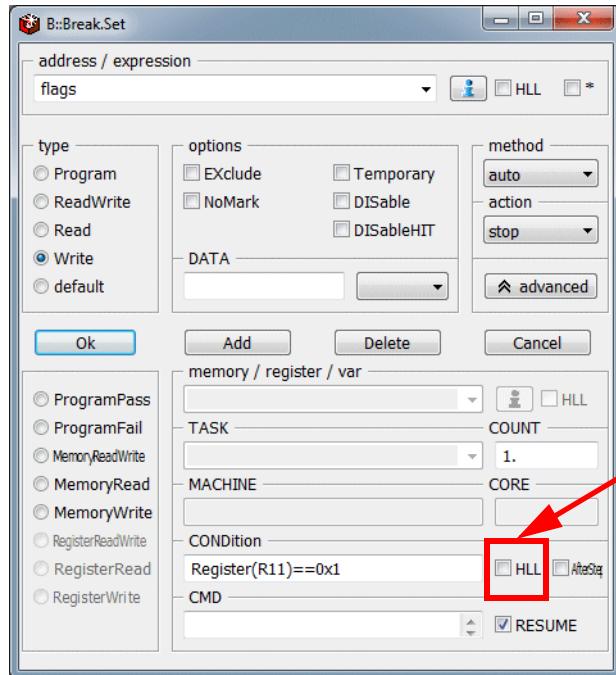


The red S indicates an intrusive breakpoint

## Conditions not in HLL Syntax

It is also possible to write register-based or memory-based conditions.

**Examples:** Stop the program executions on a write to the address flags if Register R11 is equal to 1.

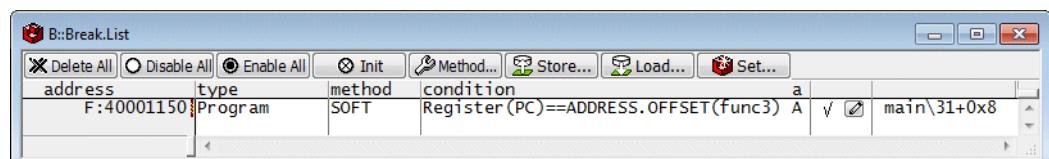
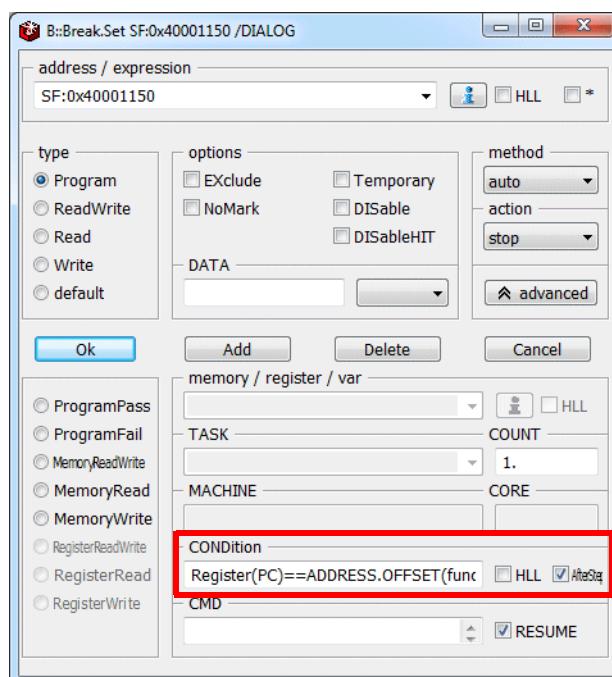
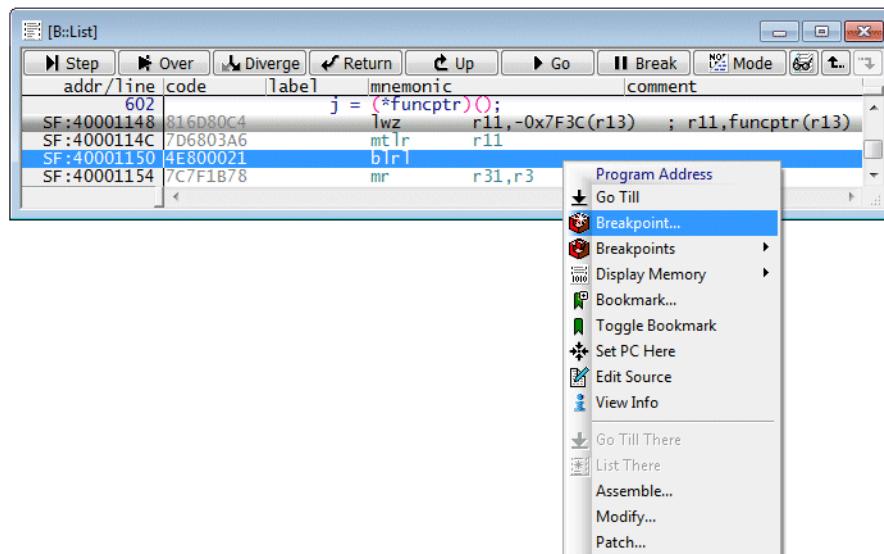


Switch HLL OFF ->  
TRACE32 syntax can be used  
to specify the condition

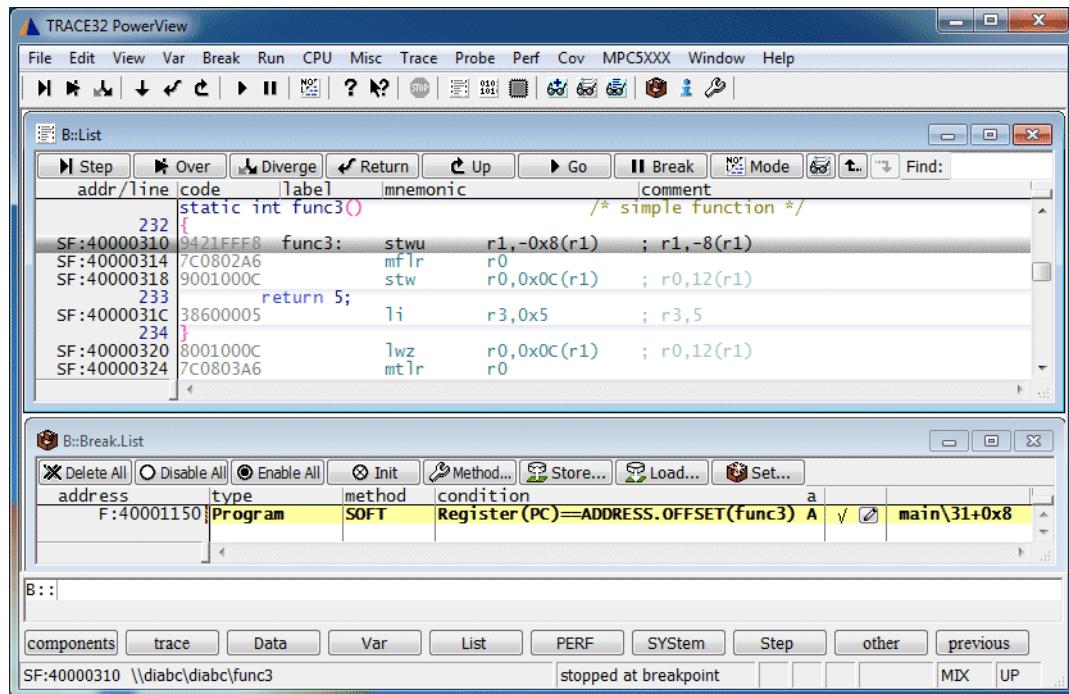
```
; stop the program execution at a write to the address flags if the
; register R11 is equal to 1
Break.Set flags /Write /CONDITION Register(R11)==0x1

; stop program execution at a write to the address flags if the long
; at address D:0x1000 is larger then 0x12345
Break.Set flags /Write /CONDITION Data.Long(D:0x1000)>0x12345
```

**Example:** Stop the program execution if an register-indirect call calls the function func3.

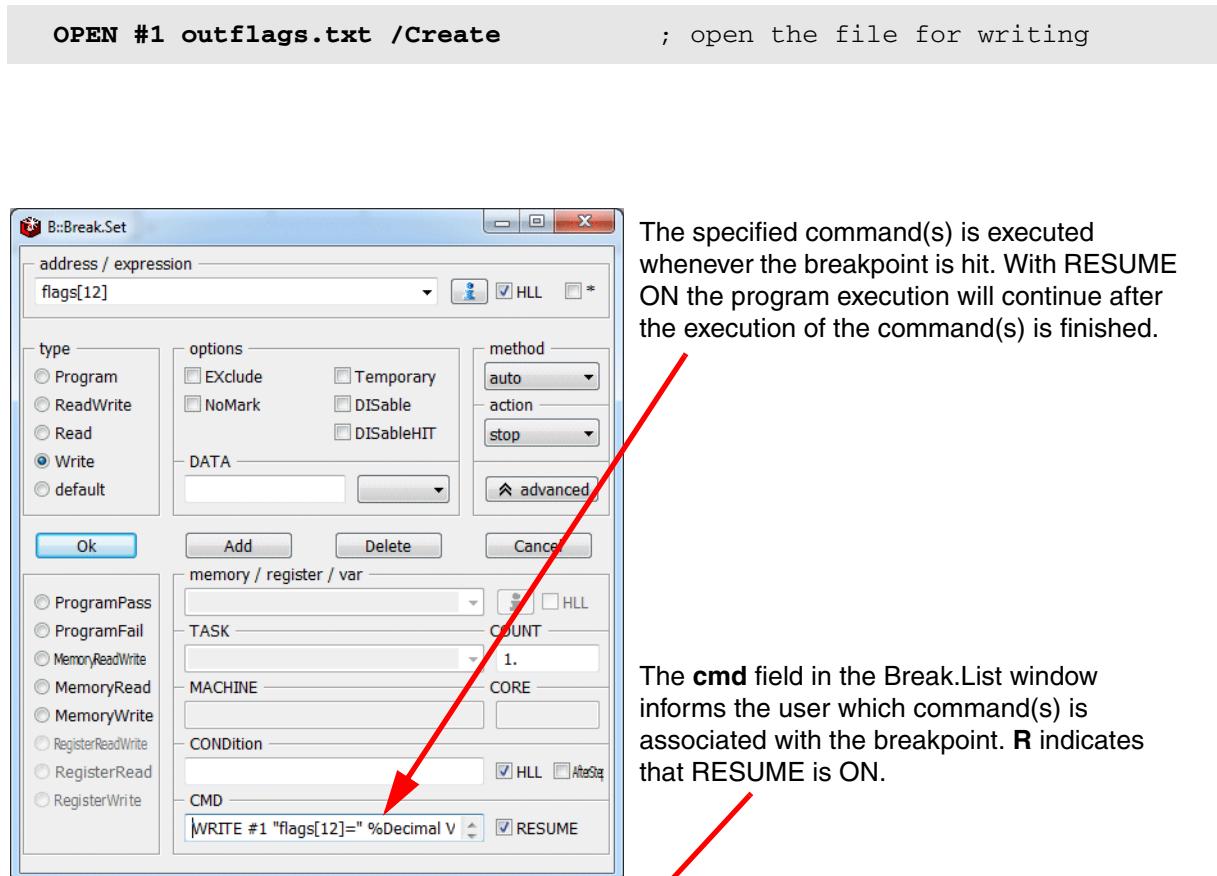


```
Break.Set main\31+0x8 /CONDITION Register(PC)==ADDRESS.OFFSET(func3)
/AfterStep
```



The field CMD allows to specify one or more commands that are executed when the breakpoint is hit.

**Example:** Write the contents of flags[12] to a file whenever the write breakpoint at the variable flags[12] is hit.



The **cmd** field in the Break.List window informs the user which command(s) is associated with the breakpoint. **R** indicates that RESUME is ON.

B::Break.List						
<input type="checkbox"/> Delete All	<input type="checkbox"/> Disable All	<input checked="" type="checkbox"/> Enable All	<input type="checkbox"/> Init	<input type="checkbox"/> Method...	<input type="checkbox"/> Store...	<input type="checkbox"/> Load...
address	type	method	cmd			
C:40004134	Write	ONCHIP	\\WRITE #1 \"fFlags[12]=\" %Decimal Var.VALUE(fFlags[12])	r	/	<input checked="" type="checkbox"/> fFlags[12]

```
Var.Break.Set flags[12] /Write /CMD "WRITE #1 \"fFlags[12]=\" %Decimal
Var.VALUE(flags[12])" /RESUME
```



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 completed.

The screenshot shows the TRACE32 PowerView interface. The main window displays assembly code for a file named 'diabc.c'. The code includes instructions like stbx, addi, b, li, cmpwi, and bgt. A red arrow points to the status bar at the bottom, which shows the message 'stopped by r/w breakpoint'.

The bottom window shows a 'Break.List' table with one entry:

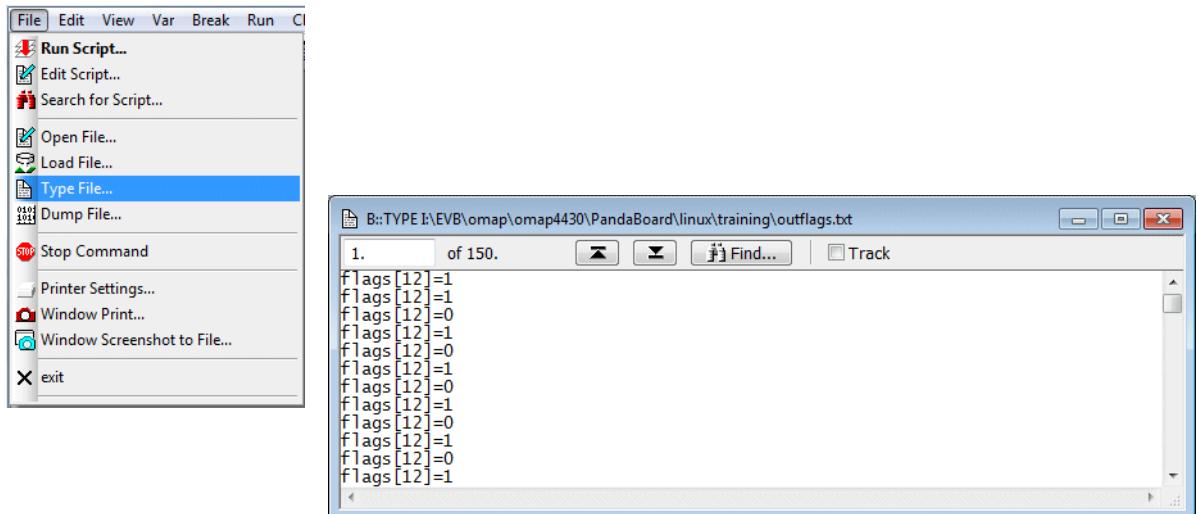
address	type	method	cmd
C:40004134	write	ONCHIP	WRITE #1 "Flags[12] = "%Decimal Var.VALUE(Flags[12]) R ✓ Flags[12]

The state of the debugger toggles between running and stopped

CLOSE #1

; close the file when you are done

## Display the result:



The on-chip break logic of some cores allows to combine data accesses and instructions to form a complex breakpoint (e.g. ARM or PowerArchitecture).

### Preconditions

- Harvard architecture.
- The on-chip break logic supports a logical AND between Program and Read/Write breakpoints.

### Advantageous

- Program breakpoints on address ranges are possible.
- Read/Write breakpoints on address ranges are possible.

**Example:** Stop the program execution when the function sieve writes a 1 to variable flags[3]. (If your core does not support this feature, the **radio buttons** (MemoryWrite, MemoryRead etc.) are grey.)

The screenshot shows the 'B::Break.Set' dialog box. A red box highlights the 'address / expression' field containing 'sieve'. A blue box highlights the 'type' section where 'MemoryWrite' is selected. A green box highlights the 'memory / register / var' field containing 'flags[3]'. A dashed blue arrow points from the 'DATA' field (containing '1.') to the 'memory / register / var' field. A red arrow points from the 'MemoryWrite' radio button to the 'MemoryWrite' section of the 'memory / register / var' field.

1. Define the address (range) of the instructions here

2. Select MemoryWrite

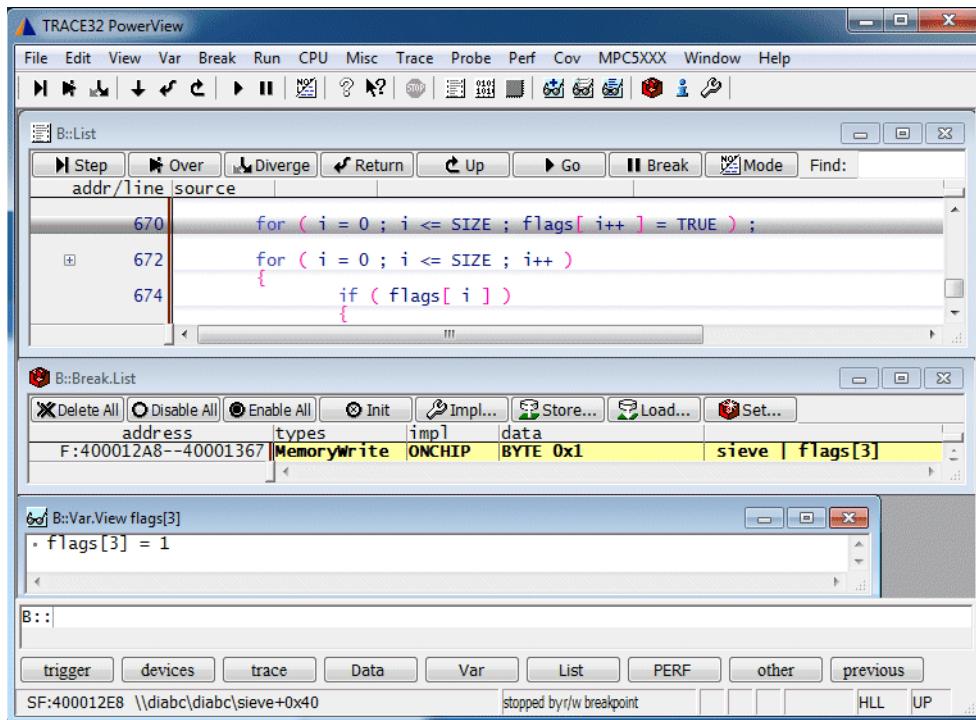
3. Define the address (range) for the MemoryWrite accesses

4. Define the data value for the MemoryWrite accesses

B::Break.List

address	types	impl	data	
F:400012A8--40001367	Memorywrite	ONCHIP	BYTE 0x1	sieve   flags[3]

```
Var.Break.Set sieve /VarWrite flags[3] /DATA.auto 1.
```



## Exclude

**(Advanced users only, not available on all cores)**

The breakpoint is inverted.

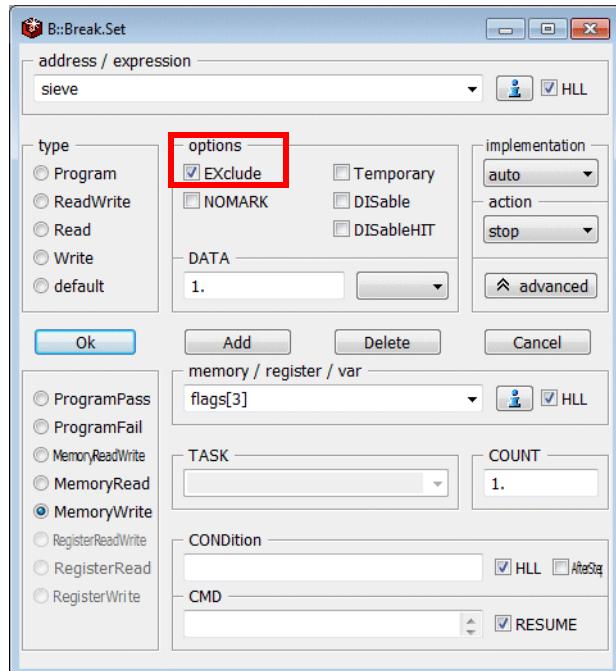
- by the inverting logic of the on-chip break logic
- by setting the specified breakpoint type to the following 2 address ranges  
0x0--(start\_of\_breakpoint\_range-1)  
(end\_of\_breakpoint\_range+1)--end\_of\_memory

The EXclude option applies only to Onchip breakpoints.

If the on-chip breakpoint logic does not provide an inverting logic, the core has to provide the facility to set the specified breakpoint type on 2 address ranges.

## Example for the Option EXclude

Stop the program execution when code outside of the function sieve writes 1 to the variable flags[3].



B::Break.List						
<input type="button" value="Delete All"/>	<input type="button" value="Disable All"/>	<input type="button" value="Enable All"/>	<input type="button" value="Init"/>	<input type="button" value="Impl..."/>	<input type="button" value="Store..."/>	<input type="button" value="Load..."/>
address	types	impl	options	data		
F:400012A8--40001367	MemoryWrite	ONCHIP	EXclude	BYTE 0x1	sieve	flags[3]

```
Var.Break.Set sieve /VarWrite flags[3] /DATA.auto 1. /EXclude
```

```

[B::List]
Step Over Diverge Return Up Go Break Mode Find:
breakpoint addr/line source
649     j = func25();
651     p = func26();
653     for (j = 0; j < 10; j++)
654     {
655         sieve();
656     }
658 }

char flags[SIZE+1];
int sieve() /* sieve of er
E mw   664 {
register int i, primz, k;
int anzahl;
E mw   668     anzahl = 0;
E mw   670     for (i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE
E mw   672     for (i = 0 ; i <= SIZE ; i++)
{

```

The function sieve is marked with **Exclude memoryWrite** breakpoints

The following command allows to check how the option EXclude is implemented.

```
Break.List /Onchip
```

Inverting logic of on-chip break logic:

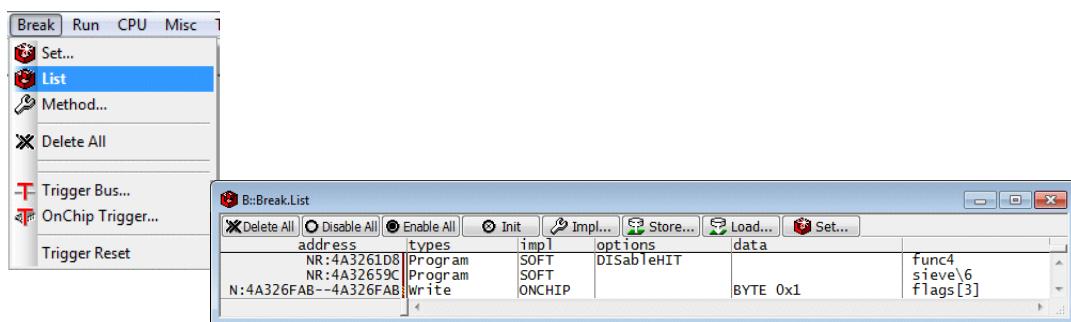
B::Break.List /Onchip					
<input checked="" type="checkbox"/> Delete All	<input type="radio"/> Disable All	<input checked="" type="radio"/> Enable All	<input checked="" type="checkbox"/> Init	<input checked="" type="checkbox"/> Impl...	<input checked="" type="checkbox"/> Store...
address	types	impl	options	data	onchip resource
R:00001800--00001BFF	Memorywrite	ONCHIP	Exclude	BYTE 0x1	(func23+0x4)--(main\26+0x23)   f1ags[3]

Two address range breakpoints:

B::Break.List /Onchip					
<input checked="" type="checkbox"/> Delete All	<input type="radio"/> Disable All	<input checked="" type="radio"/> Enable All	<input checked="" type="checkbox"/> Init	<input checked="" type="checkbox"/> Impl...	<input checked="" type="checkbox"/> Store...
address	types	impl	options	data	onchip resource
C:00000000--700013E9	Memorywrite	ONCHIP		BYTE 0x1	C:0x0--0x700013E9   f1ags[3]
C:70001450--FFFFFFFFFF	Memorywrite	ONCHIP		BYTE 0x1	C:0x70001450--0xFFFFFFFF   f1ags[3]

If your TRACE32 PowerView does not accept the option EXclude, delete all other Onchip breakpoints, to make sure that enough resources are available.

# Display a List of all Set Breakpoints

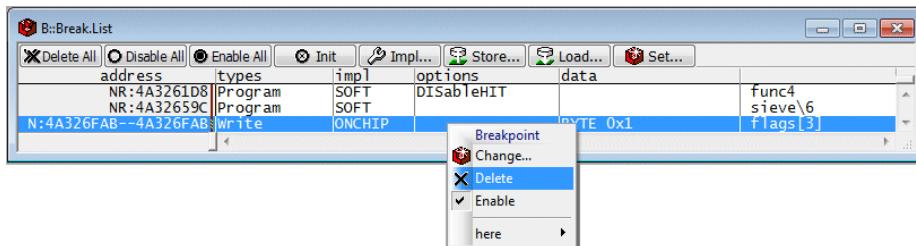


<b>address</b>	Address of the breakpoint
<b>types</b>	Type of the breakpoint
<b>impl</b>	Implementation of the breakpoint or disabled
<b>action</b>	Action selected for the breakpoint (if not stop)
<b>options</b>	Option defined for the breakpoint
<b>data</b>	Data value that has to be read/written to stop the program execution by the breakpoint
<b>count</b>	Current value/final value of the counter that is combined with a breakpoint
<b>condition</b> <b>A</b> (AfterStep)	Condition that has to be true to stop the program execution by the breakpoint A ON: Perform an assembler single step before condition is evaluated
<b>cmd</b> (command) <b>R</b> (resume)	Commands that are executed after the breakpoint hit R ON: continue the program execution after the specified commands were executed
<b>task</b>	Name of the task for a task-aware breakpoint
	Symbolic address of the breakpoint

**Break.List** [/<option>]

List all breakpoints

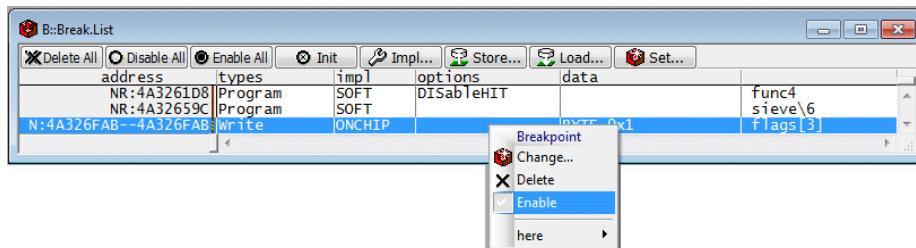
# Delete Breakpoints



**Break.Delete** <address>|<address\_range> [/<type>] [/<implem.>] [/<option>] Delete breakpoint

**Var.Break.Delete** <hl\_expression> [/<type>] [/<implem.>] [/<option>] Delete HLL breakpoint

# Enable/Disable Breakpoints



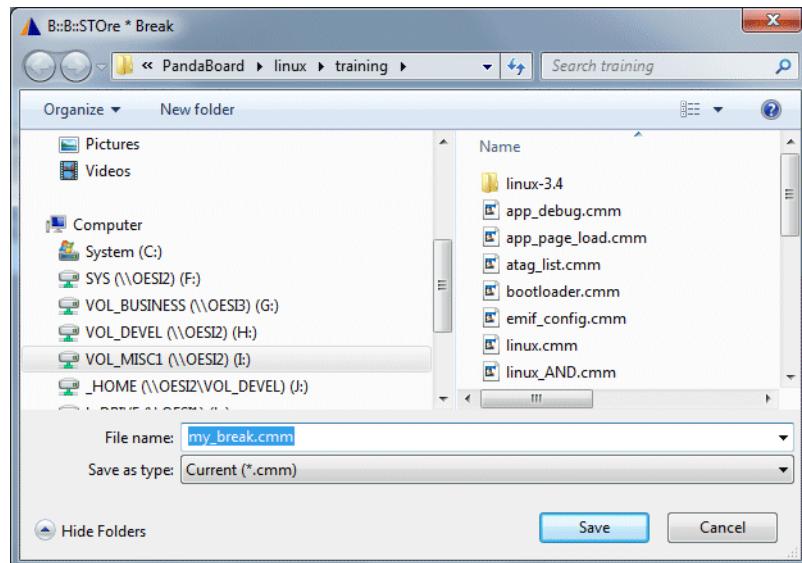
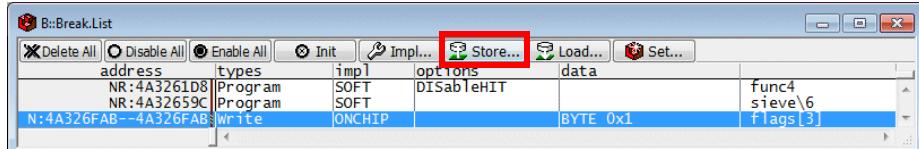
**Break.ENABLE** <address>|<address\_range> [/<option>]

Enable breakpoint

**Break.DISABLE** <address>|<address\_range> [/<option>]

Disable breakpoint

# Store Breakpoint Settings



```
// AndT32 Fri Jul 04 13:17:41 2003

B:::

Break.RESet
Break.Set    func4 /Program /DISableHIT
Break.Set    sieve /Program
Var.Break.Set \\diabp555\Global\flags[3]; /Write /DATA.Byte 0x1;

ENDDO
```

**STOre <filename> Break**

Generate a script for breakpoint settings

## Debugging of Optimized Code

A video tutorial about debugging optimized code can be found here:

[https://www.lauterbach.com/tut\\_optimized.html](https://www.lauterbach.com/tut_optimized.html)

HLL mode and MIX mode debugging is simple, if the compiler generates a continuous block of assembler code for each HLL code line.

If compiler optimization flags are turned on, it is highly likely that two or more detached blocks of assembler code are generated for individual HLL code lines. This makes debugging laboriously.

TRACE32 PowerView displays a drill-down button, whenever two or more detached blocks of assembler code are generated for an HLL code line.

```
B:List.HII
Step Over Next Return Up Go Break Mode Find: diabc.c
addr/line source
664 char flags[SIZE+1];
int sieve() /* sieve of erathostenes */
{
    register int i, primz, k;
    int anzahl;
    anzahl = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
    for ( i = 0 ; i <= SIZE ; i++ )
    {
        if ( flags[ i ] )
    }
```

Drill-down button

The following background information is fundamental if you want to debug optimized code:

- In HLL debug mode, the HLL code lines are displayed as written in the compiled program (source line order).
- In MIX debug mode, the target code is disassembled and the HLL code lines are displayed together with their assembler code blocks (target line order). This means if two or more detached blocks of assembler code are generated for an HLL code line, this HLL code line is displayed more than once in a MIX mode source listing.

The expansion of the drill-down button shows how many detached blocks of assembler code are generated for the HLL line (e.g. two in the example below).

#### List.HLL

Display source listing, display HLL code lines only.

#### List.Mix /Track

Display source listing, display disassembled code and the assigned HLL code lines.

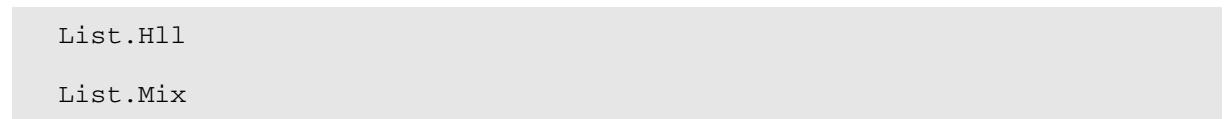
The blue cursor in the MIX mode display follows the cursor movement of the HLL mode display (Track option).

addr/line	source
	char flags[SIZE+1];
664 [	int sieve() /* sieve of erathostenes */
	register int i, primz, k;
	int anzahl;
668	anzahl = 0;
670	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
672 L	for ( i = 0 ; i <= SIZE ; i++ )
672	for ( i = 0 ; i <= SIZE ; i++ )
674	if ( flags[ i ] )

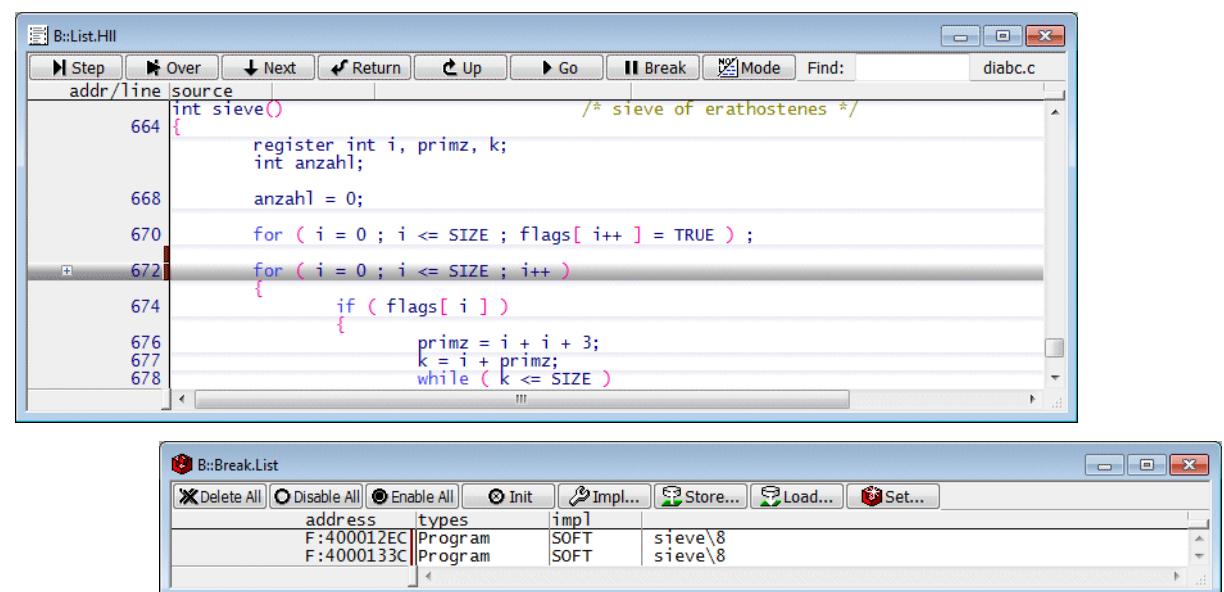
addr/line	code	label	mnemonic	comment
672	SF:400012EC	.L514:	li	r31,0x0 ; i,0
	SF:400012F0	.L522:	cmpwi	r31,0x12 ; i,18
	SF:400012F4		bgt	0x40001344 ; .L517 (-)
674	SF:400012F8		{	
	SF:400012FC		if ( flags[ i ] )	
	SF:40001300		lis	r12,0x4000 ; r12,16384
	SF:40001304		addi	r12,r12,0x4128 ; r12,r12,16680
	SF:40001308		lbzx	r12,r12,r31 ; r12,r12,i
	SF:40001320		cmpwi	r12,0x0 ; r12,0
	SF:40001324		beq	0x4000133C ; .L521 (-)
676	SF:4000132C		{	
	SF:40001330		primz = i + 3;	
	SF:40001331		add	r12,r31,r31 ; r12,i,i
	SF:40001334		addi	r30,r12,0x3 ; primz,r12,3
	SF:40001338		k = i + primz;	
	SF:40001339		add	r29,r31,r30 ; k,i,primz
678	SF:4000133B	.L520:	cmpwi	r29,0x12 ; k,18
	SF:4000133C		bgt	0x40001338 ; .L519 (-)
680	SF:40001340		{	
	SF:40001342		flags[ k ] = FALSE;	
	SF:40001344		lis	r12,0x4000 ; r12,16384
	SF:40001348		addi	r12,r12,0x4128 ; r12,r12,16680
	SF:40001352		li	r11,0x0 ; r11,0
	SF:40001356		stbx	r11,r12,r29 ; r11,r12,k
681	SF:40001358		k += primz;	
	SF:40001360		add	r29,r29,r30 ; k,k,primz
	SF:40001364		b	0x40001318 ; .L520
683	SF:40001368		}	
	SF:40001370	.L519:	anzahl++;	
	SF:40001374		addi	r28,r28,0x1 ; anzahl,anzahl,1
	SF:40001378		{	
672	SF:4000137C	.L521:	addi	r31,r31,0x1 ; i,i,1
	SF:40001380		b	0x400012F0 ; .L522

To keep track when debugging optimized code, it is recommended to work with an HLL mode and a MIX mode display of the source listing in parallel.



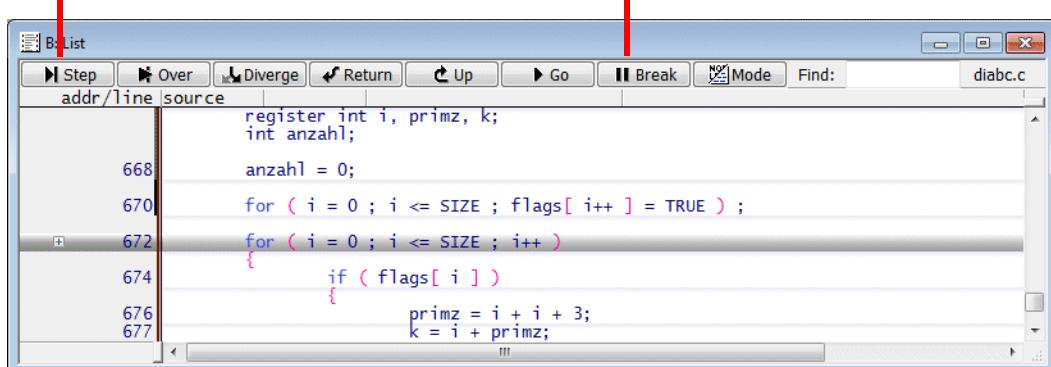
Please be aware of the following:

If a Program breakpoint is set to an HLL code line for which two or more detached blocks of assembler code are generated, a Program breakpoint is set to the start address of each assembler block.



# Basic Debug Control

There are local buttons in the **List** window for all basic debug commands



<b>Step</b>	Single stepping (command: <b>Step</b> )
<b>Over</b>	Step over call (command <b>Step.Over</b> ).
<b>Diverge</b>	Exit loops or fast forward to not yet stepped code lines. <b>Step.Over</b> is performed repeatedly.

## More details on Step.Diverge

TRACE32 maintains a list of all assembler/HLL lines which were already reached by a Step. These reached lines are marked with a slim grey line in the List window.

```
int sieve() /* sieve of eratosthenes */
664: register int i, primz, k;
      anzahl = 0;
664:   { or ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
668:     or ( i = 0 ; i <= SIZE ; i++ )
       if ( flags[ i ] )
668:   }
```

The following command allows you to get more details:

```
List.auto /DIVERGE
```

[B::List /DIVERGE]

s	state	i	addr/line	source
h stop			664	int sieve() /* sieve of erathostenes
h done			668	{ register int i, primz, k;
h done			670	int anzahl;
h done	[		672	anzahl = 0;
hit	672			for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
				for ( i = 0 ; i <= SIZE ; i++ )
				{ for ( i = 0 ; i <= SIZE ; i++ )
				{ if ( flags[ i ] )
				{ primz = i + i + 3;
				k = i + primz;
				while ( k <= SIZE )
				{ flags[ k ] = FALSE;
				k += primz;
				}
target			683	anzahl++;
			687	}
			688	return anzahl;

Drag this handle to see the DIVERGE details

[B::List /DIVERGE]

s	state	i	addr/line	code	label	mnemonic	comment
a stop			602	j = (*funcptr)();			
a stop			SF:40001148	l1wz r11,-0x7F3C(r13)	r11	funcptr(r	
a done			SF:4000114C	mtlr r11			
a done	i		SF:40001150	b1rl			
a stop			SF:40001154	mr r31,r3		; j,r3	
a done			604	j = func5( (int) j, (char) 2, (long) 3 );			
a done			SF:40001158	mr r3,r31		; r3,j	
a done			SF:4000115C	li r4,0x2		; r4,2	
a done			SF:40001160	li r5,0x3		; r5,3	
a done			SF:40001164	b1 0x400003A0		; func5	
stop			SF:40001168	mr r31,r3		; j,r3	

### Column layout

<b>s</b>	Step type performed on this line <b>a:</b> Step on assembler level was started from this code line <b>h:</b> Step on HLL level was started from this code line
<b>state</b>	<b>done:</b> code line was reached by a Step and a Step was started from this code line. <b>hit:</b> code line was reached by a Step. <b>target:</b> code line is a possible destination of an already started Step, but was not reached yet (mostly caused by conditional branches).  <b>stop:</b> program execution stopped at code line.
<b>i</b>	indirect branch taken (return instructions are not marked).

## Example 1: Diverge through function sieve.

### 1. Run program execution until entry to function sieve.

stop indicates that the program execution was stopped at this code line

[B::List /DIVERGE]

s	state	i	addr/line	source
				char flags[SIZE+1];
			664	int sieve()
				{ register int i, primz, k;
			668	int anzahl;
			670	anzahl = 0;
			672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
			674	for ( i = 0 ; i <= SIZE ; i++ )
			676	{ if ( flags[ i ] )
			677	{ primz = i + i + 3;
			678	k = i + primz;
				while ( k <= SIZE )

[B::List /DIVERGE]

s	state	i	addr/line	source
				char flags[SIZE+1];
			stop	int sieve()
				{ register int i, primz, k;
			668	int anzahl;
			670	anzahl = 0;
			672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
			674	for ( i = 0 ; i <= SIZE ; i++ )
			676	{ if ( flags[ i ] )
			677	{ primz = i + i + 3;
			678	k = i + primz;
				while ( k <= SIZE )

### 2. Start a Step.Diverge command.

h indicates that a Step command in HLL mode was started in this line

hit indicates that this code line was reached by Step command

[B::List /DIVERGE]

s	state	i	addr/line	source
				char flags[SIZE+1];
			h stop	int sieve()
				{ register int i, primz, k;
			668	int anzahl;
			670	anzahl = 0;
			672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
			674	for ( i = 0 ; i <= SIZE ; i++ )
			676	{ if ( flags[ i ] )
			677	{ primz = i + i + 3;
			678	k = i + primz;
				while ( k <= SIZE )

### 3. Continue with Step.Diverge.



s	state	i	addr/line	source
				char flags[SIZE+1];
			664	int sieve() /* sieve of eratho
	h stop			register int i, primz, k;
	h done		668	int anzahl;
	hit		670	anzahl = 0;
			672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
			674	{
			676	if ( flags[ i ] )
			677	primz = i + i + 3;
			678	k = i + primz;
				while ( k <= SIZE ) {

**done** indicates that the code line was reached by a Step command and that a Step command was started from this code line

The drill-down button indicates that two or more detached blocks of assembler code are generated for an HLL code line

```

B::List / DIVERGE
Step Over Diverge Return Up Go Break Mode Find: sieve
s state i addr/line source
h stop 664
h done 668
hit 670
h done 672
hit 672
h done 674

```

```

char flags[SIZE+1];
int sieve() /* sieve of erathos
register int i, primz, k;
int anzah];
anzahl = 0;
for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
for ( i = 0 ; i <= SIZE ; i++ )
{
    if ( flags[ i ] )
    {
        primz = i + i + 3;
        k = i + primz;
        while ( k <= SIZE )

```

#### 4. Continue with Step.Diverge.

The drill-down tree is expanded and the HLL code line representing the reached block of assembler code is marked as hit

```

B::List / DIVERGE
Step Over Diverge Return Up Go Break Mode Find: sieve
s state i addr/line source
h stop 664
h done 668
h done 670
hit [ 672
h done 672
hit 674
h done 676
h done 677
h done 678

```

```

char flags[SIZE+1];
int sieve() /* sieve of erathos
register int i, primz, k;
int anzah];
anzahl = 0;
for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
for ( i = 0 ; i <= SIZE ; i++ )
{
    if ( flags[ i ] )
    {
        primz = i + i + 3;
        k = i + primz;
        while ( k <= SIZE )

```

This HLL code line includes a conditional branch



The screenshot shows two windows side-by-side. The left window is titled 'B::List / DIVERGE' and displays assembly code with labels like 'h\_stop', 'h\_done', 'hit', and 'target'. The right window is titled 'B::List.Mix / Track' and shows the assembly code with its corresponding HLL source code. A red arrow points from the text 'This HLL code line includes a conditional branch' to the 'hit' line in the assembly dump.

s state	i	addr/line	source
			char flags[SIZE+1];
h stop	664		int sieve()
			{ register int i, primz, k;
			int anzahl;
			anzahl = 0;
			for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE
			for ( j = 0 ; j <= SIZE ; j++ )
			for ( i = 0 ; i <= SIZE ; i++ )

addr/line	code	label	mnemonic
672			for ( i = 0 ; i <= SIZE ; i++ )
SF:400012EC	3BE00000	L514:	li r31,0x0
SF:400012F0	2C1F0012	L522:	cmpwi r31,0x12
SF:400012F4	41810050		bgt 0x40001344

## 5. Continue with Step.Diverge.

The reached code line is marked as hit



The not-reached code line is marked as target



The screenshot shows the same two windows as the previous one. The left window now highlights the 'hit' line, while the right window highlights the 'target' line. Red arrows point from the text descriptions to their respective highlighted lines in the assembly dump.

s state	i	addr/line	source
h stop	664		register int i, primz, k;
h done	668		int anzahl;
h done	670	[	anzahl = 0;
h done	672	]	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE
hit	672		for ( i = 0 ; i <= SIZE ; i++ )
	674		if ( flags[ i ] )
	676		primz = i + i + 3;
	677		k = i + primz;
	678		while ( k <= SIZE )
	680		flags[ k ] = FALSE;
	681		k += primz;
	683		anzahl++;
	687		}
	688		return anzahl;

## 6. Continue with Step.Diverge (several times).

All code lines are now either marked as **done**, **hit** or **target**

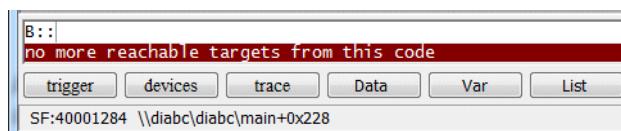
s state	i	addr/line	source
h stop		664	int sieve()
h done		668	register int i, primz, k;
h done		670	int anzahl;
h done		672	anzahl = 0;
h done		672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] =
h done		674	for ( i = 0 ; i <= SIZE ; i++ )
h done		676	{
h done		677	if ( flags[ i ] )
h done		678	{
h done		680	primz = i + i + 3;
h done		681	k = i + primz;
target		683	while ( k <= SIZE )
target		687	{
target		688	flags[ k ] = FALSE;
			k += primz;
			}
			anzahl++;
			}
			return anzahl;

## 7. Continue with Step.Diverge.

A code line former marked as **target** changes to **hit** when it is reached

s state	i	addr/line	source
char flags[SIZE+1];			
h stop		664	int sieve()
h done		668	register int i, primz, k;
h done		670	int anzahl;
h done		672	anzahl = 0;
h done		672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE
h done		674	for ( i = 0 ; i <= SIZE ; i++ )
h done		676	{
h done		677	if ( flags[ i ] )
h done		678	{
h done		680	primz = i + i + 3;
h done		681	k = i + primz;
hit		683	while ( k <= SIZE )
target		687	{
target		688	flags[ k ] = FALSE;
			k += primz;
			}
			anzahl++;
			}
			return anzahl;

When all reachable code lines are marked as **done**, the following message is displayed:



The **DIVERGE** marking is cleared when you use the **Go.direct** command without address or the **Break** command while the program execution is stopped.

## Example 2: Exit a loop.

DIVERGE marking is done whenever you single step.

If all code lines of a loop are marked as **done/hit**, a Step.Diverge will exit the loop

**B::List /DIVERGE**

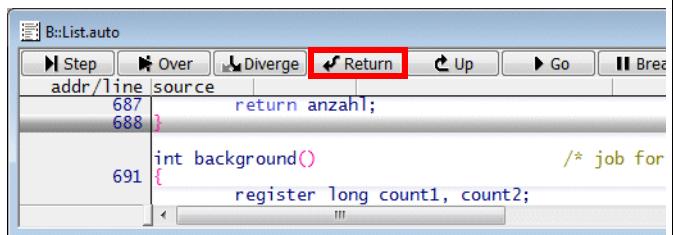
s	state	i	addr/line	source
				register int i, primz, k; int anzahl;
h done			668	anzahl = 0;
h done			670	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
h done target	[		672	for ( i = 0 ; i <= SIZE ; i++ ) for ( i = 0 ; i <= SIZE ; i++ )
h done			674	{ if ( flags[ i ] )
h done			676	primz = i + i + 3;
h done			677	k = i + primz;
h done			678	while ( k <= SIZE )
h done hit			680	{ flags[ k ] = FALSE;
			681	k += primz;
target			683	} anzahl++;
target			687	}
			688	return anzahl;
			691	int background() /* job for background */
				register long count1, count2;

**B::List /DIVERGE**

s	state	i	addr/line	source
				char flags[SIZE+1];
				int sieve() /* sieve of erathostenes */
h stop			664	{ register int i, primz, k;
h done			668	int anzahl;
h done			670	anzahl = 0;
h done target	[		672	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
h done			674	for ( i = 0 ; i <= SIZE ; i++ ) for ( i = 0 ; i <= SIZE ; i++ )
h done			676	{ if ( flags[ i ] )
h done			677	primz = i + i + 3;
h done			678	k = i + primz;
h done			680	while ( k <= SIZE )
h done			681	{ flags[ k ] = FALSE;
hit			683	k += primz;
target			687	} anzahl++;
				}
				return anzahl;

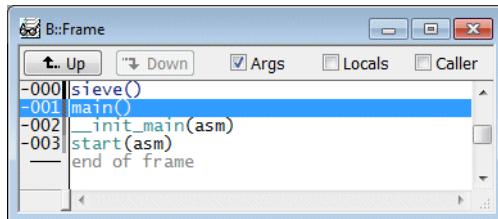
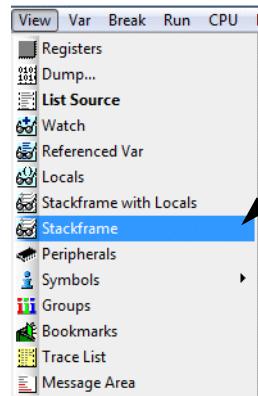
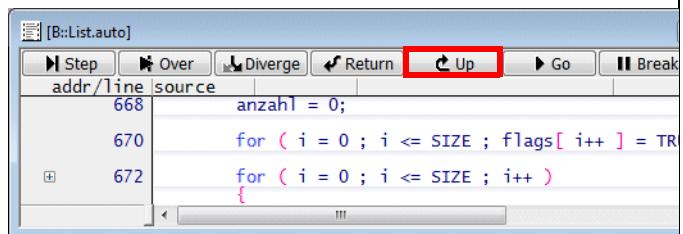
## Return

**Return** sets a temporary breakpoint to the last instruction of a function and then starts the program execution.



## Up

This command is used to return to the function that called the current function. For this a temporary breakpoint is set to the instruction directly after the function call. Afterwards the program execution is started.



Display the HLL stack to check the function nesting

<b>Step</b> [<count>]	Single step
<b>Step.Change</b> <expression>	Step until <expression> changes
<b>Step.Till</b> <condition>	Step until <condition> becomes true, <condition> written in TRACE32 syntax
<b>Var.Step.Change</b> <hl_expression>	Step until <hl_expression> changes
<b>Var.Step.Till</b> <hl_condition>	Step until <hl_condition> becomes true, <hl_condition> as allowed in used programming language

```
Step 10.

Step.Change Register(R11)

Step.Till Register(R11)>0xAA

Var.Step.Change flags[3]

Var.Step.Till flags[3]==1
```

## Step.Over

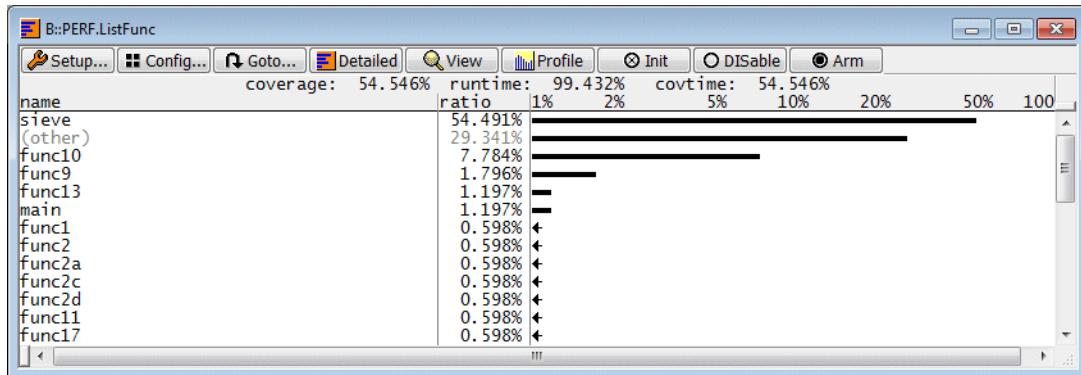
Step over call

<b>Go</b> [<address> <label>]	Start program execution
<b>Go.Next</b>	Set a temporary breakpoint to the next code line and start the program execution
<b>Go.Return</b>	Set a temporary breakpoint to the return instruction and start the program execution
<b>Go.Up</b> [<level> <address>]	Run program until it returns to the caller function

# Sample-based Profiling

## Program Counter Sampling

**Task:** get the percentage of time used by a high-level language function.



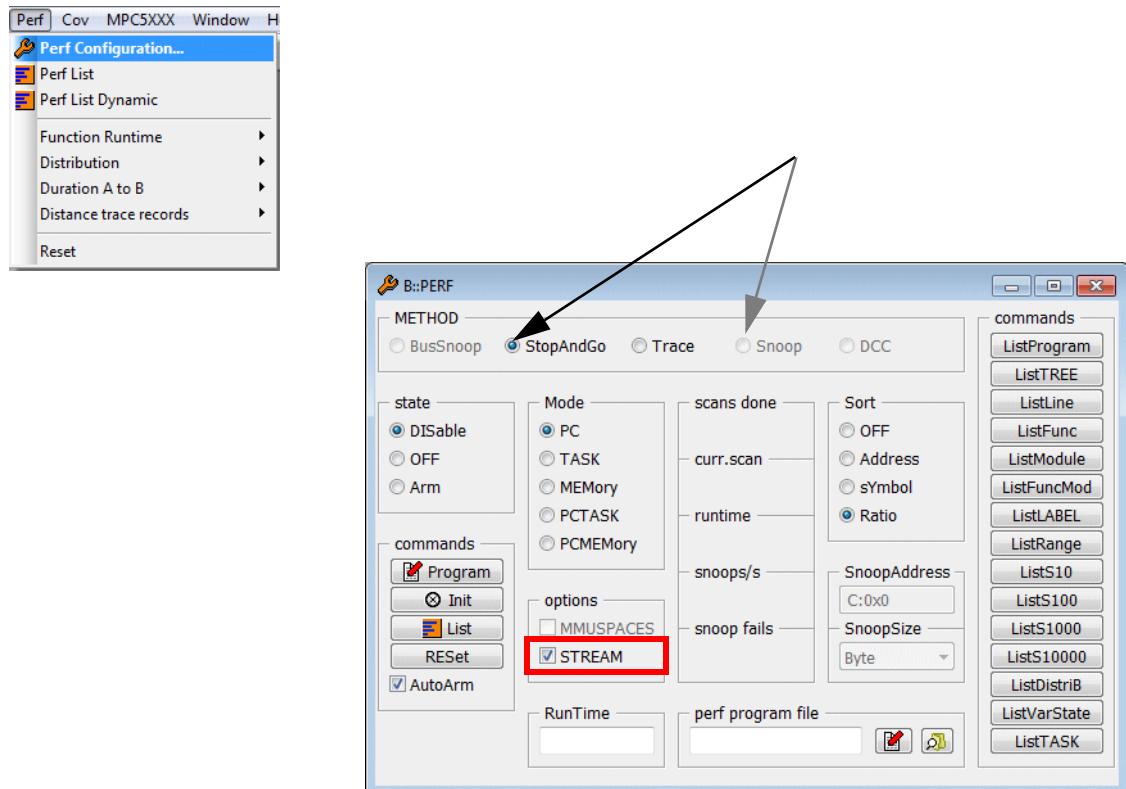
**Measurement procedure:** The Program Counter is sampled periodically. This is implemented in two ways.

- **Snoop:** Processor architecture allows to read the Program Counter while the program execution is running.
- **StopAndGo:** The program execution is stopped shortly in order to read the Program Counter.

# Standard Procedure

Steps to be taken:

## 1. Open the PERF configuration window.



### PERF.state

Display PERF configuration window

The PERF METHOD **Snoop** is automatically selected, if the processor architecture supports reading the Program Counter while the program execution is running.

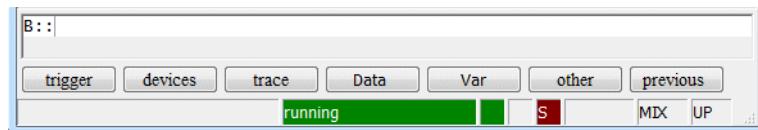
The default METHOD for all other processor architectures is **StopAndGo**.

## Remarks on the StopAndGo method

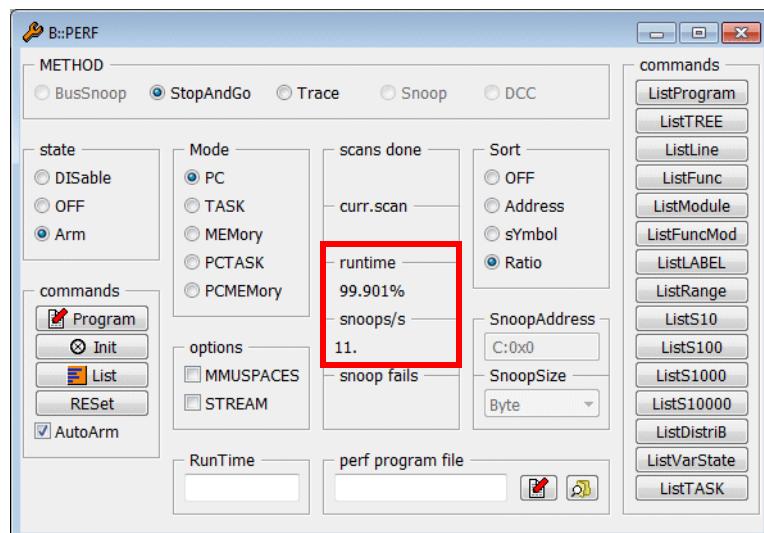
StopAnd Go means that the core is stopped periodically in order to get the actual Program Counter.

<b>STREAM ON</b>	The software running on the TRACE32 debug hardware initiates the periodic stops. This has the following advantages: <ul style="list-style-type: none"><li>• Low intrusive (approx. 50. to 100.us)</li><li>• More samples per second are possible</li></ul>
<b>STREAM OFF</b>	The software running on the host initiates the periodic stops. <ul style="list-style-type: none"><li>• More intrusive (1 ms in a worst case scenario)</li><li>• Less samples per second are possible</li></ul>

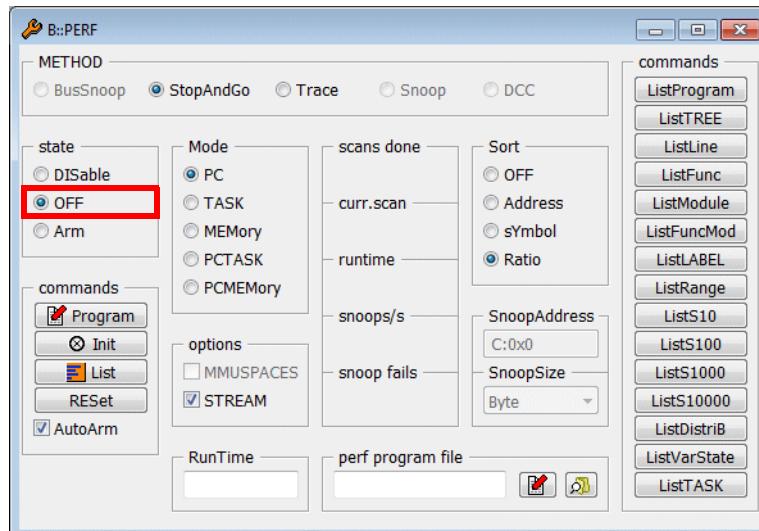
The display of a red **S** in the TRACE32 state line indicates that the program execution is periodically interrupted by the sample-based profiling.



TRACE32 tunes the sampling rate so that more than 99% of the run-time is retained for the actual program run (runtime). The smallest possible sampling rate is nevertheless 10 (snoops/s).



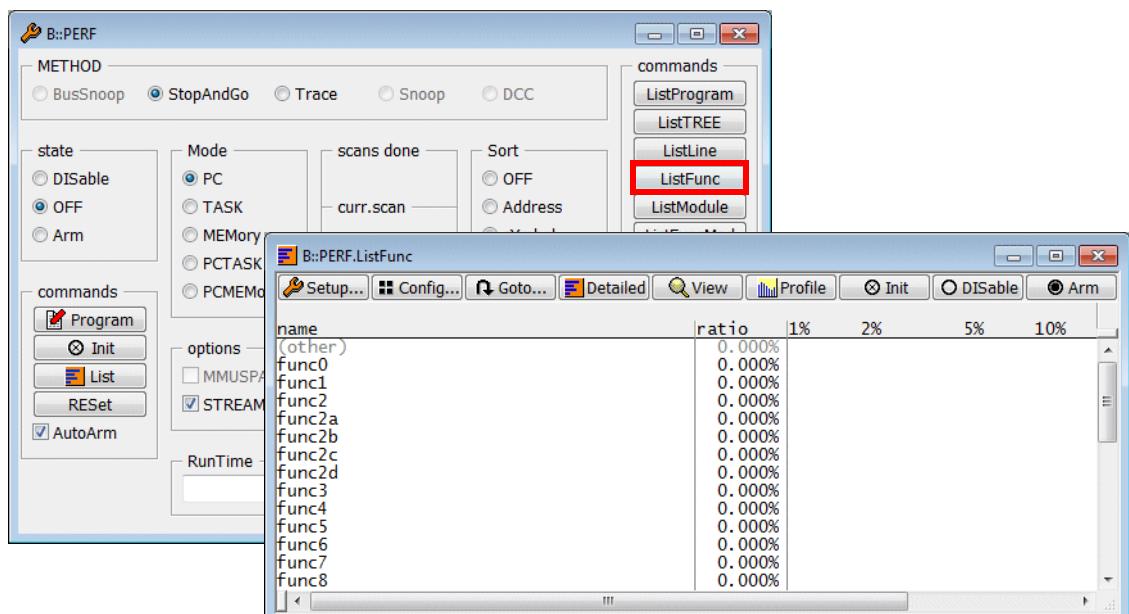
## 2. Enable the sample-based profiling by selecting the OFF state.



**PERF.OFF**

Enable the sample-based profiling

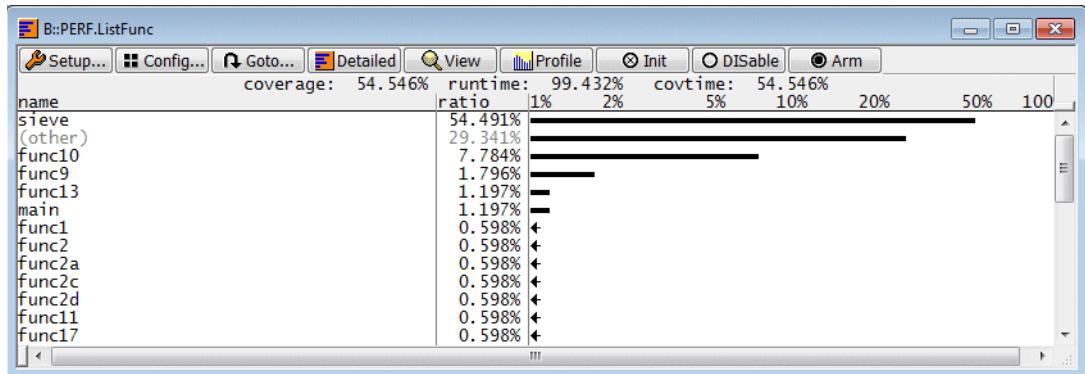
## 3. Open a result window by pushing the ListFunc button.



**PERF.ListFunc**

Open an HLL function profiling window

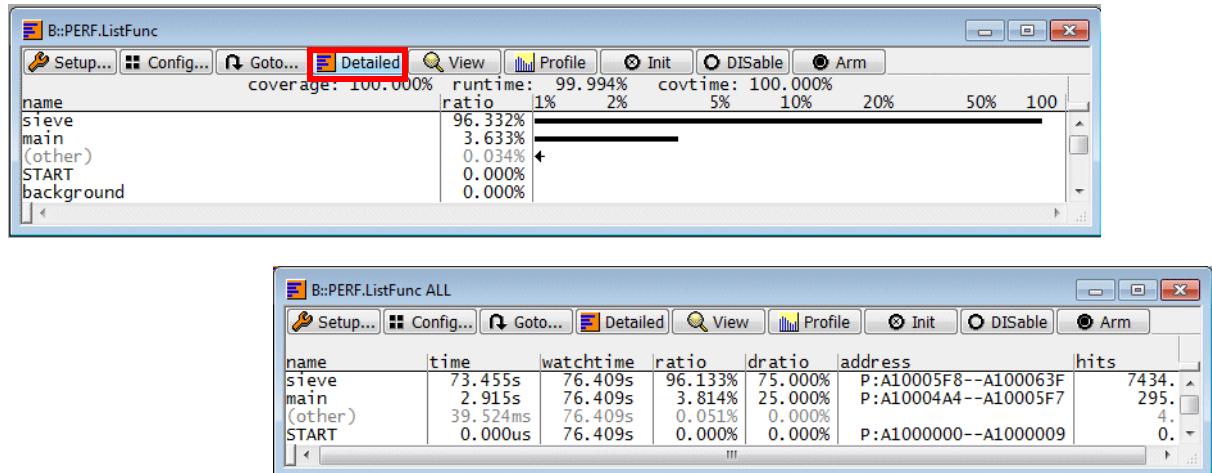
#### 4. Start the program execution and the sampling.



# Details

## In-depth Result

Push the Detailed button, to get more detailed information on the result.



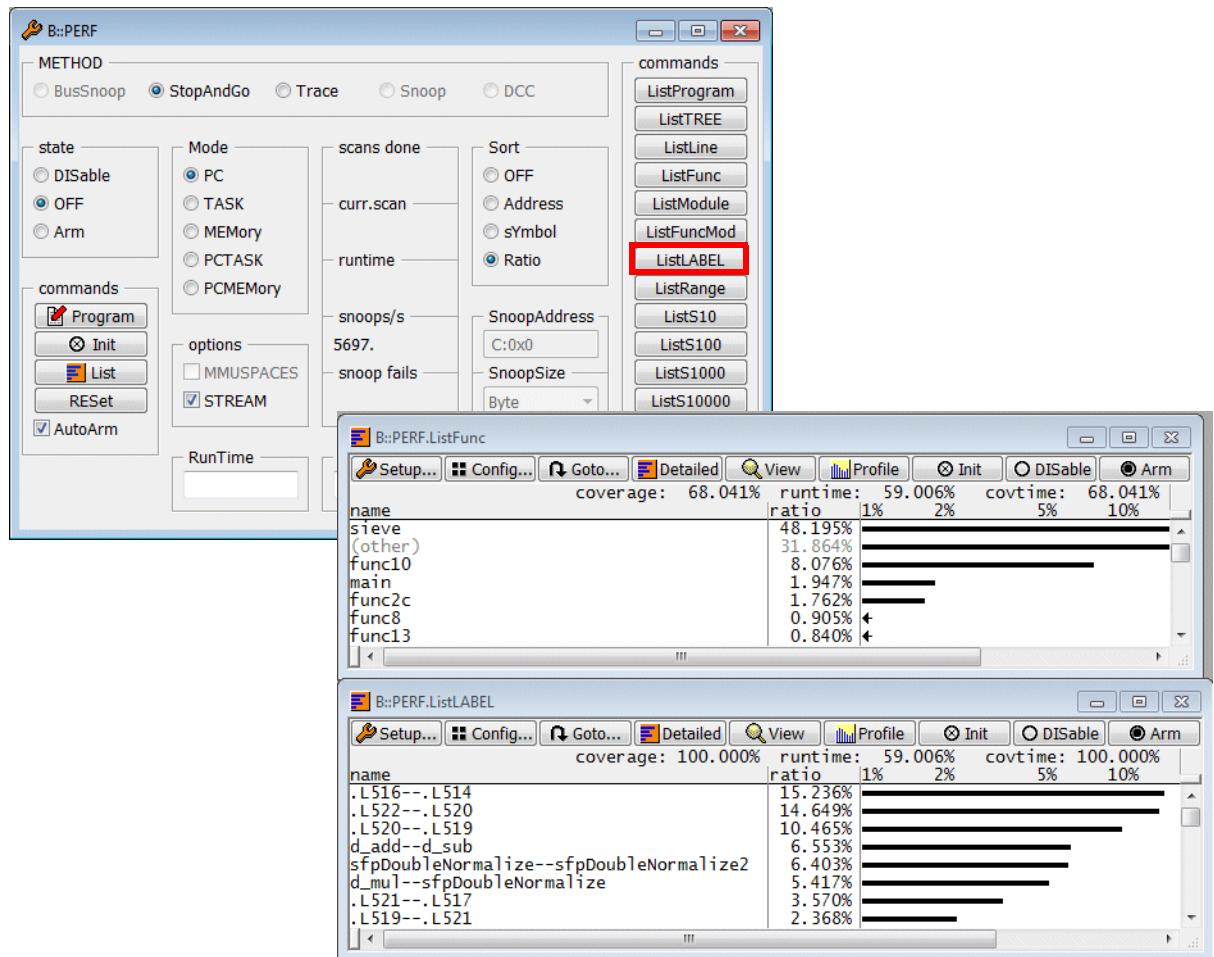
### PERF.ListFunc ALL

Open a detailed HLL function profiling window

<b>name</b>	Function name
<b>time</b>	Time in function
<b>watchtime</b>	Time the function is observed
<b>ratio</b>	Ratio of time spent by the function in percent
<b>dratio</b>	Similar to <b>Ratio</b> , but only for the last second
<b>address</b>	Function's address range
<b>hits</b>	Number of samples taken for the function

## (other)

TRACE32 assigns all samples that can not be assigned to a high-level language function to **(other)**. Especially if the ratio for (other) is quite high, it might be interesting what code is running there. In this case pushing the button **ListLABEL** is recommended.



**PERF.ListLABEL**

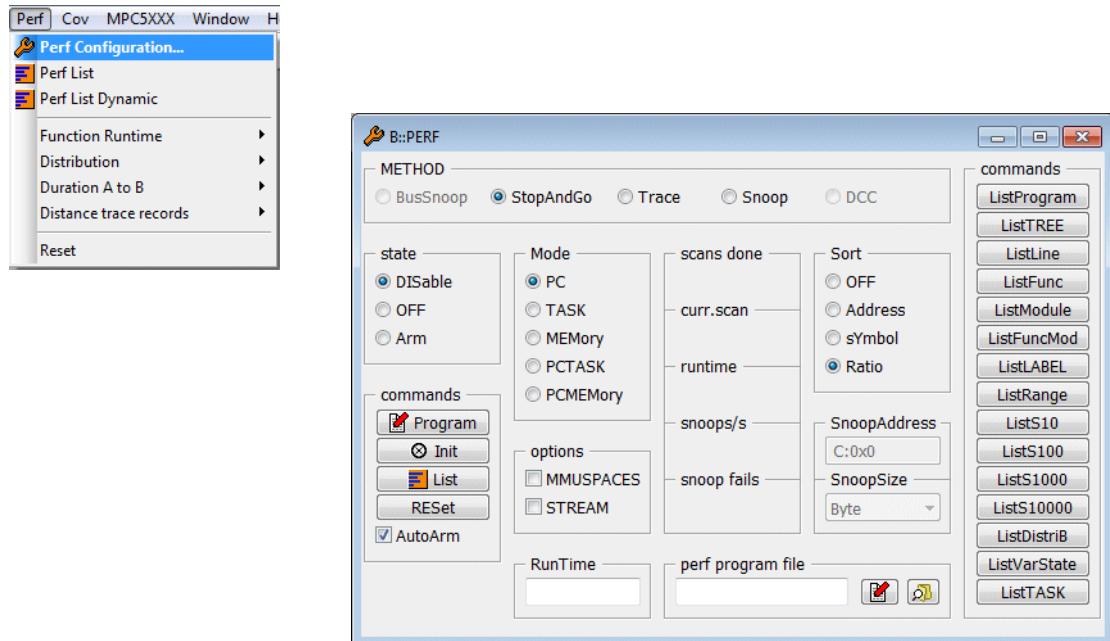
Open a window for label-based profiling

# TASK Sampling

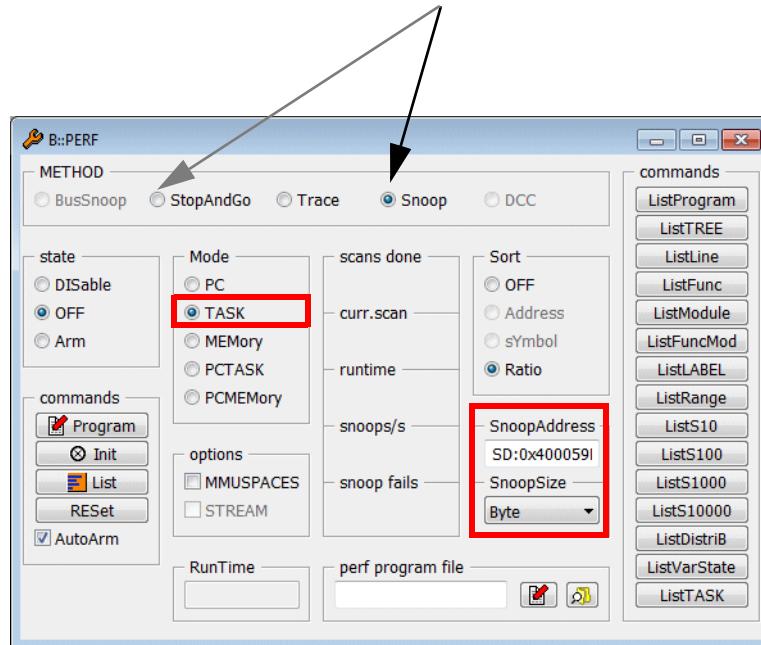
If OS-aware debugging is configured (refer to “[OS-aware Debugging](#)” in TRACE32 Glossary, page 31 (glossary.pdf)), TASK information can be sampled.

Steps to be taken:

## 1. Open the PERF configuration window.



## 2. Select Mode TASK.



Since every OS has a variable that contains the information which task/process is currently running, this variable has to be sampled while the program execution is running in order to perform TASK sampling.

TRACE32 fills the following fields when TASK mode is selected:

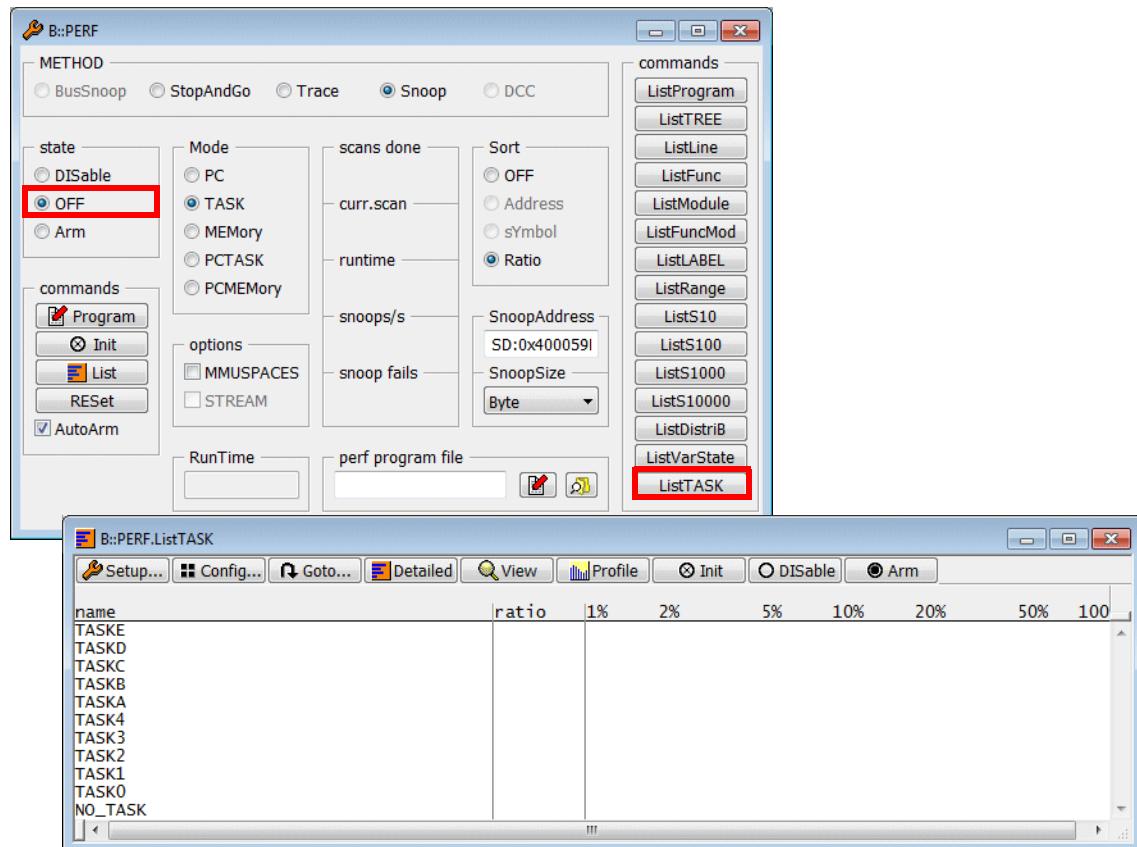
- the **SnoopAddress** field with the address of the variable.
- the **SnoopSize** field with the size of the variable.

The PERF METHOD **Snoop** is automatically selected, if the processor architecture supports reading physical memory while the program execution is running. For details refer to "["Run-time Memory Access"](#) (glossary.pdf)).

The default METHOD for all other processor architectures is **StopAndGo**.

### PERF.Mode TASK

3. Enable sample-based profiling by switching to OFF state and open the result window by pushing the ListTask button.



**PERF.OFF**

Enable the sample-based profiling

**PERF.ListTASK**

4. Start the program execution and the sampling.

