LAUTERBACH
DEVELOPMENT TOOLS

# Training - Debugger Basics

Release 09.2022

MANUAL

# Training - Debugger Basics

# Training - Debugger Basics

**Version 26-Oct-2022**

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

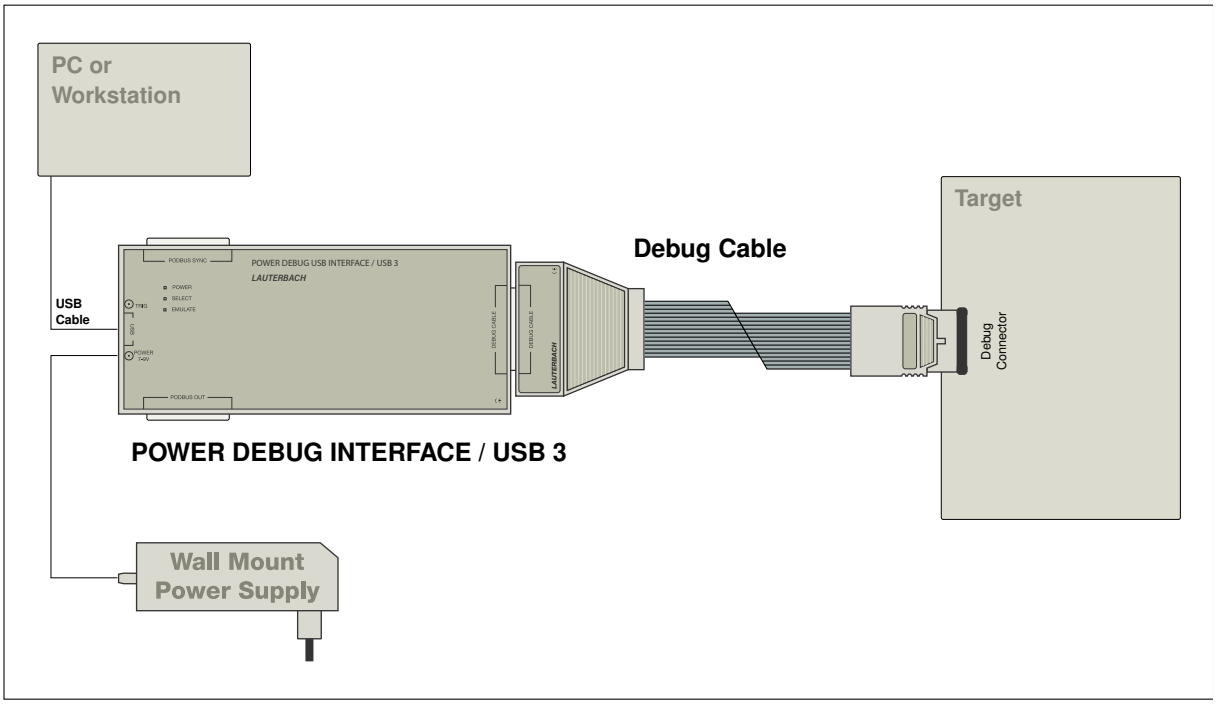
**Debug features specific for a processor architecture:**

- 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

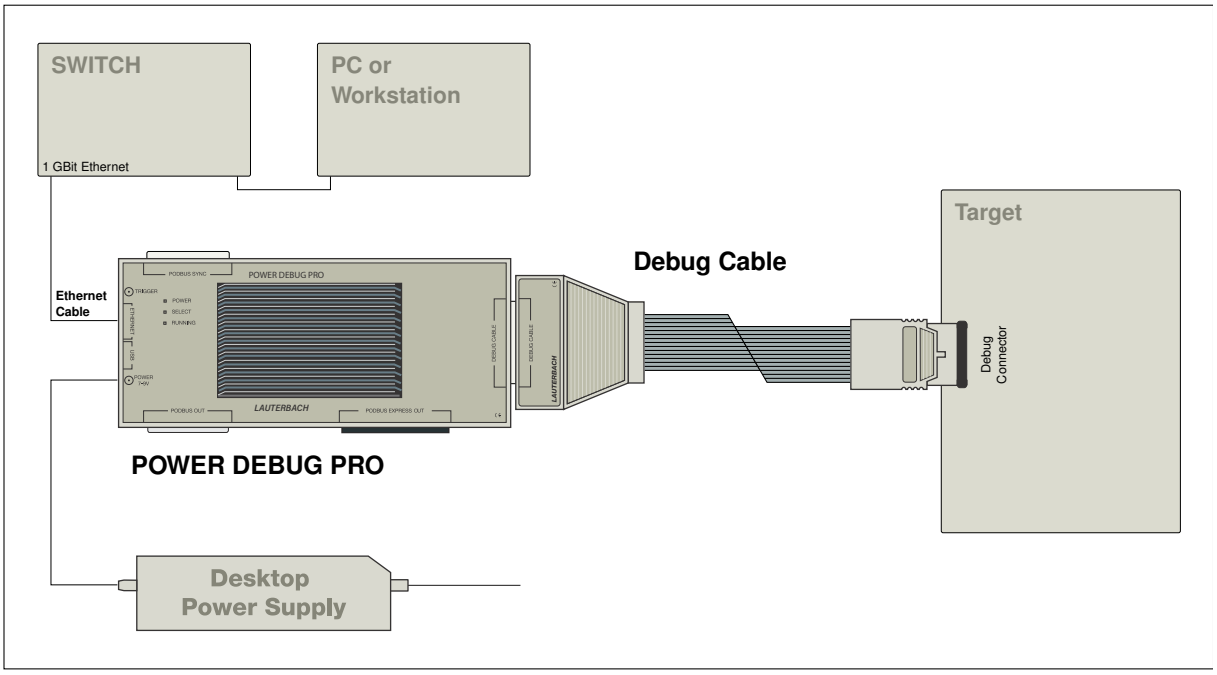



Current module:

- 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.

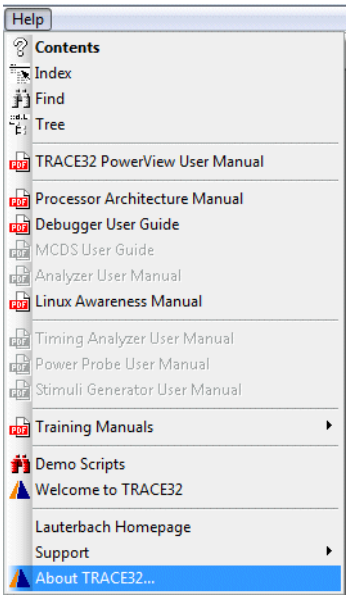
## On-chip Trace Features

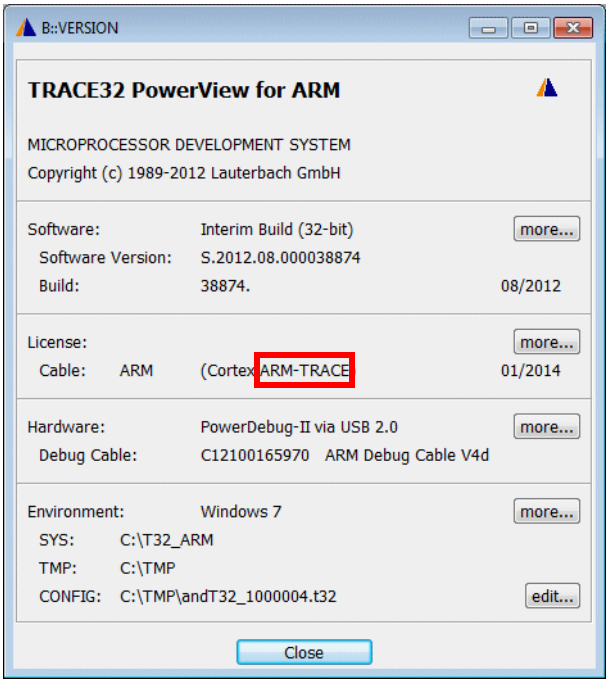
The on-chip trace buffer can store information:

- 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 Interface

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)**.**

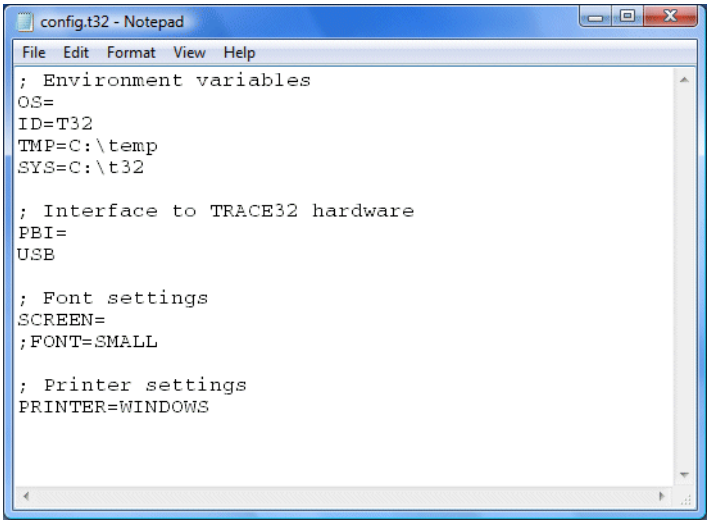
# Basic TRACE32 PowerView Parameters

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.




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.

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

### Configuration File for USB

Single debugger hardware module connected via USB:

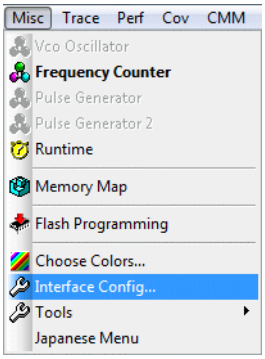
```
; Host interface
PBI=
USB

; 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
```

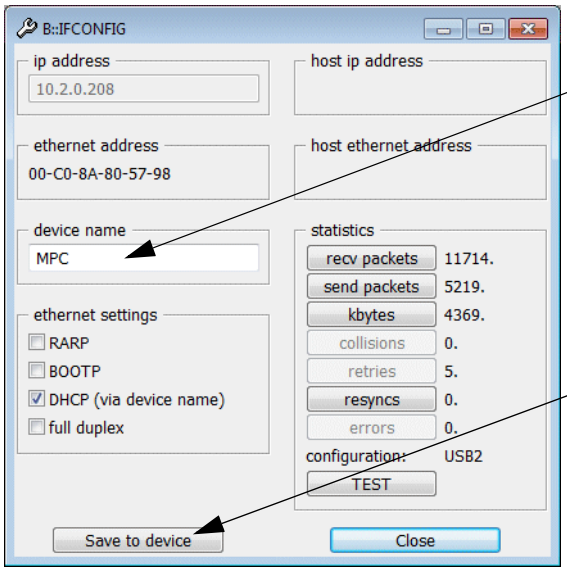
Multiple debugger hardware modules connected via USB:

```
; 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.



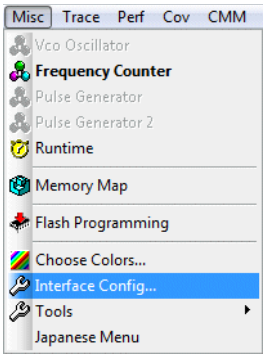

| | |
|---|---|
| **IFCONFIG** | Dialog to assign USB device name<br><br>Please be aware that USB device names are case-sensitive |

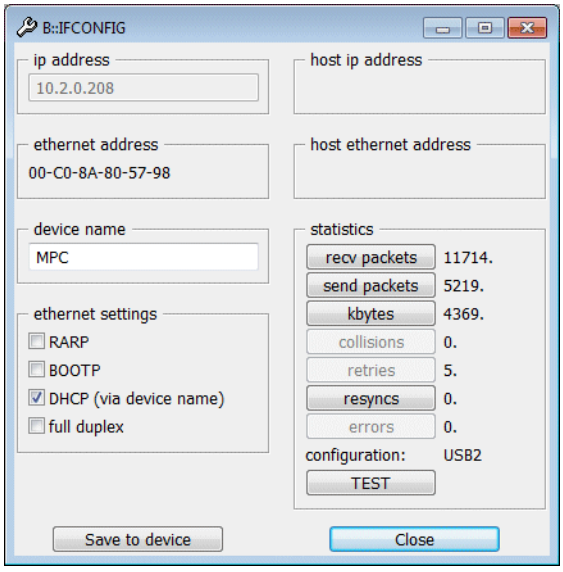
TRACE32 allows to communicate with a POWER DEBUG INTERFACE USB from a remote PC. For an example, see **"Example: Remote Control for POWER DEBUG INTERFACE / USB"** in TRACE32 Installation Guide, page 56 (installation.pdf).

## Configuration File for Ethernet

```
; 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



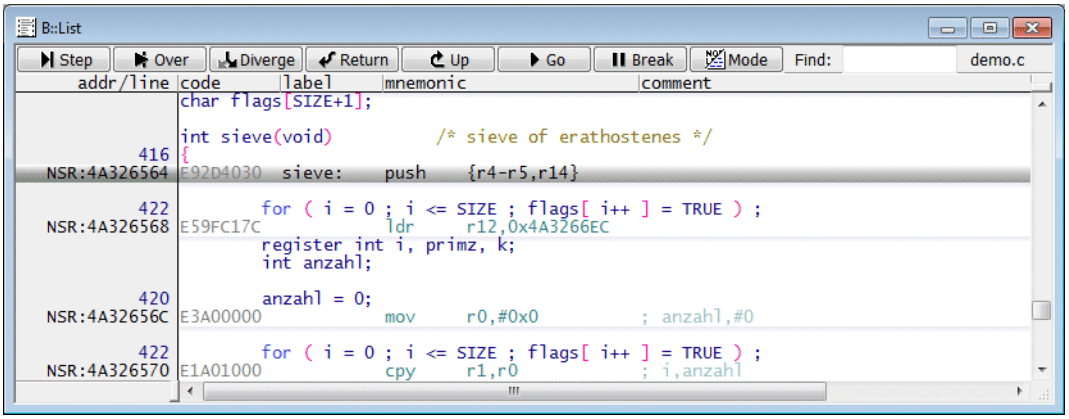




**IFCONFIG** Dialog to display and change information for the Ethernet interface

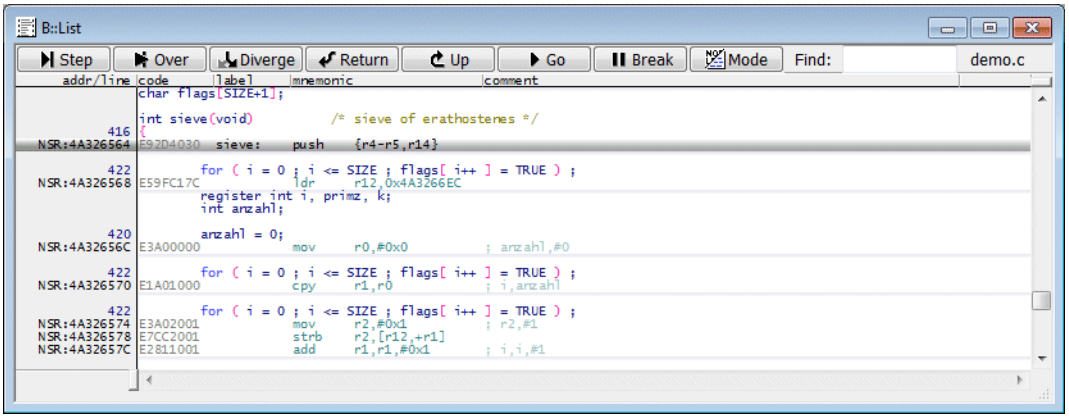
Changing the font size can be helpful for a more comfortable display of TRACE32 windows.

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

**Display with normal font:**



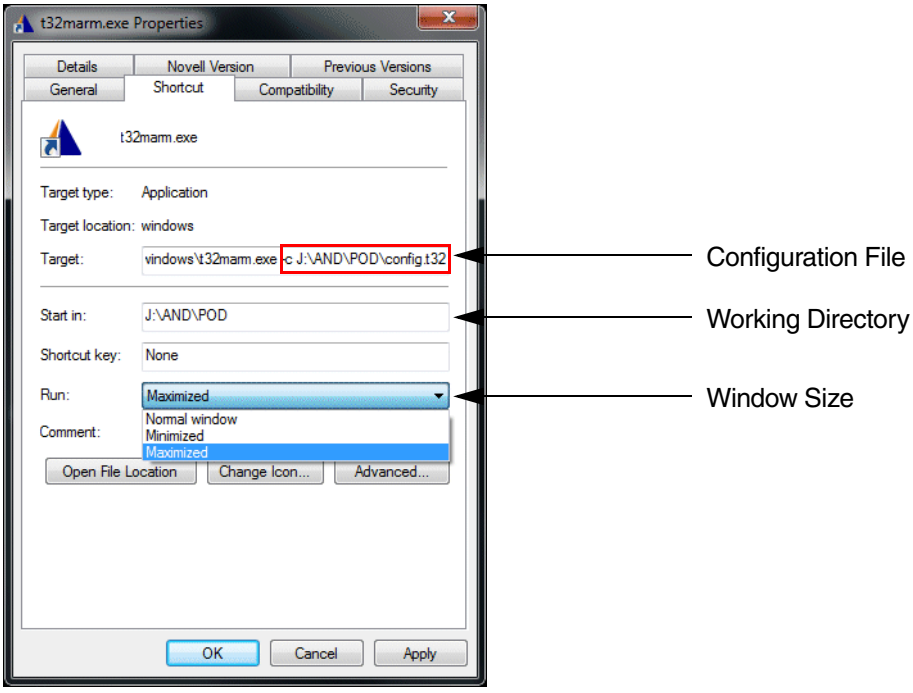

**Display with small font:**



# 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 |

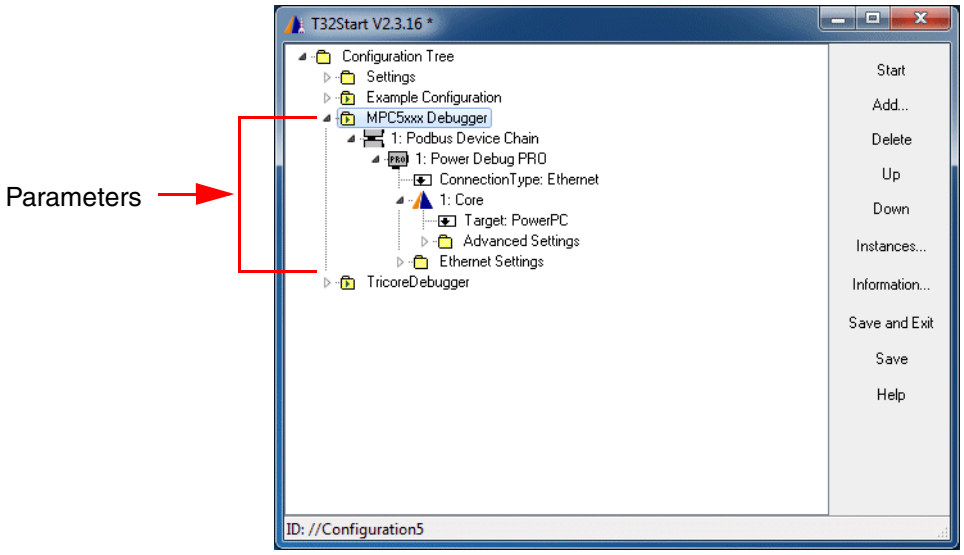
## Definition of the Window Size for TRACE32 PowerView

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**.

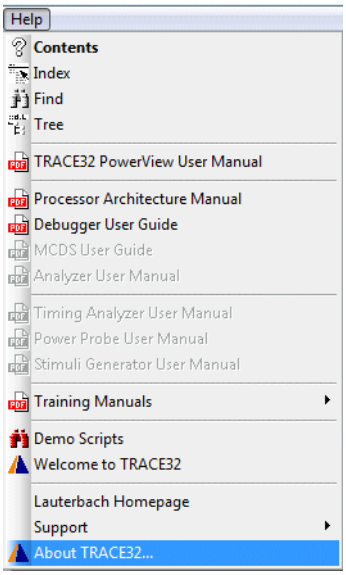
A detailed online help for **t32start.exe** is available via the **Help** button or in **"T32Start"** (app_t32start.pdf).

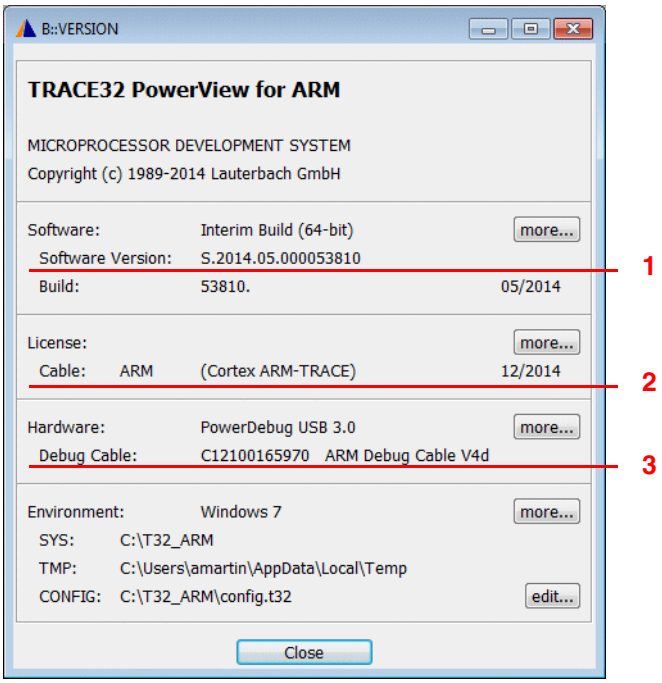


# About TRACE32

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:

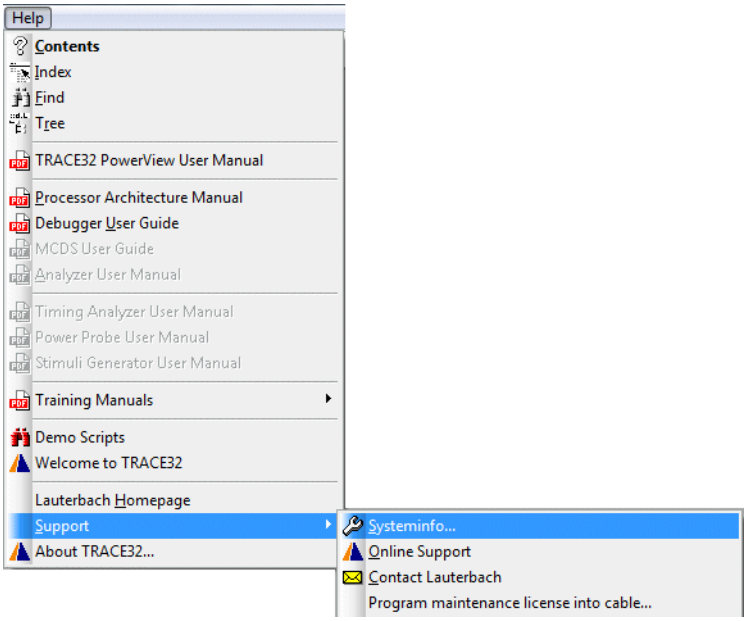
1. The version of the TRACE32 software.
2. The debug licenses programmed into the debug cable and the expiration date of your software warranty respectively the expiration date of your software maintenance.
3. The serial number of the debug cable.

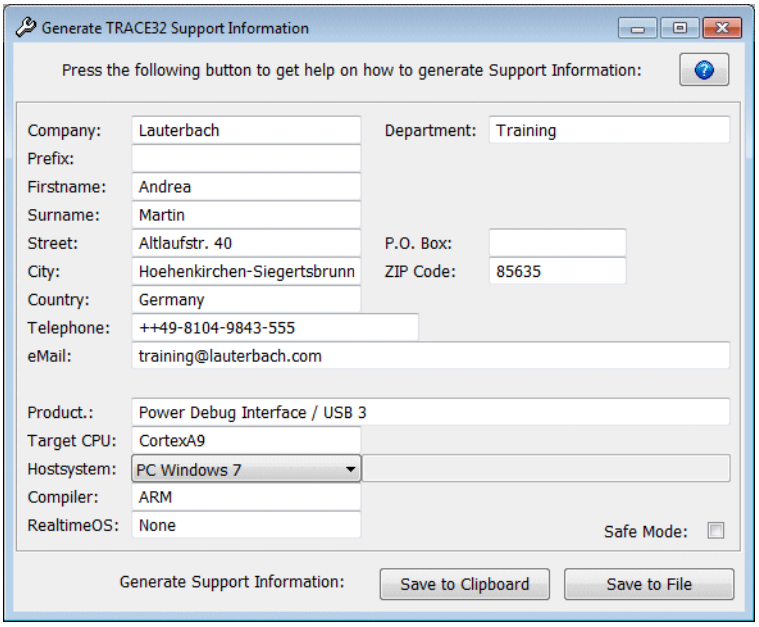
| | |
|---|---|
| **VERSION.view** | Display the VERSION window. |
| **VERSION.HARDWARE** | Display more details about the TRACE32 hardware modules. |
| **VERSION.SOFTWARE** | Display more details about the TRACE32 software. |

# 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**.



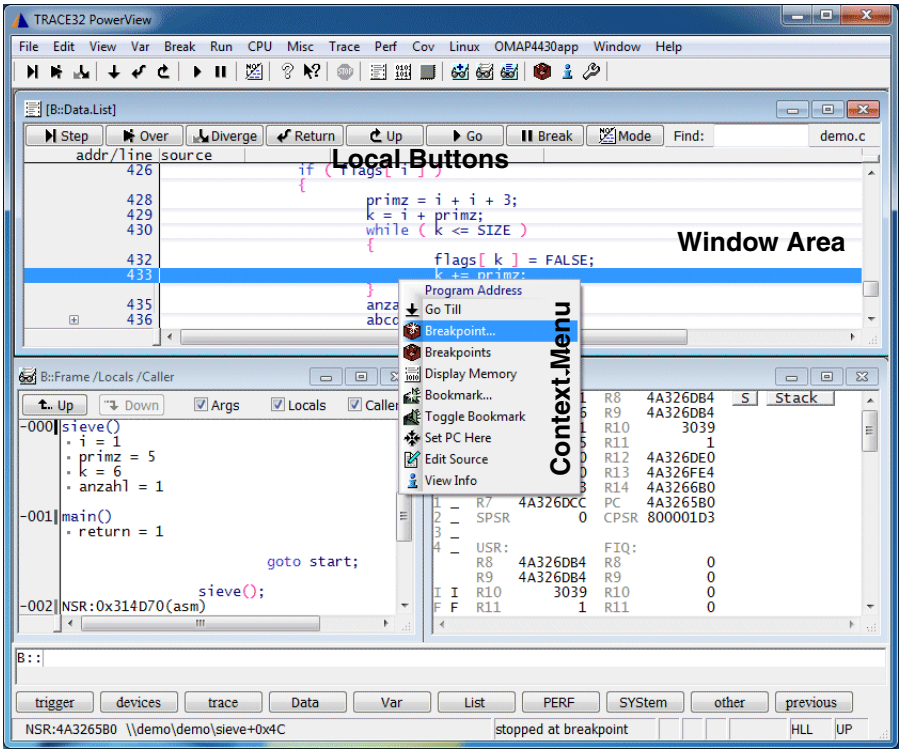




2. Preferred: click **Save to File**, and send the system information as an attachment to your e-mail.
3. Click **Save to Clipboard**, and then paste the system information into your e-mail.

# 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



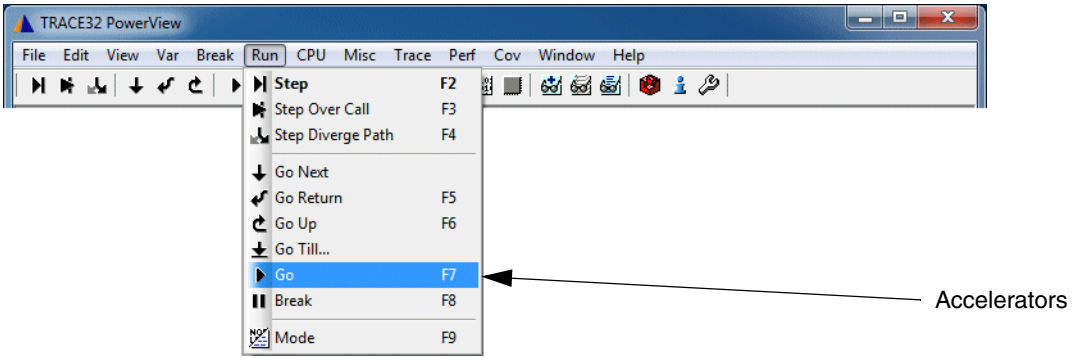

The structure of the menu bar and the tool bar are defined by the file **t32.men** which is located in the TRACE32 system directory.

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.

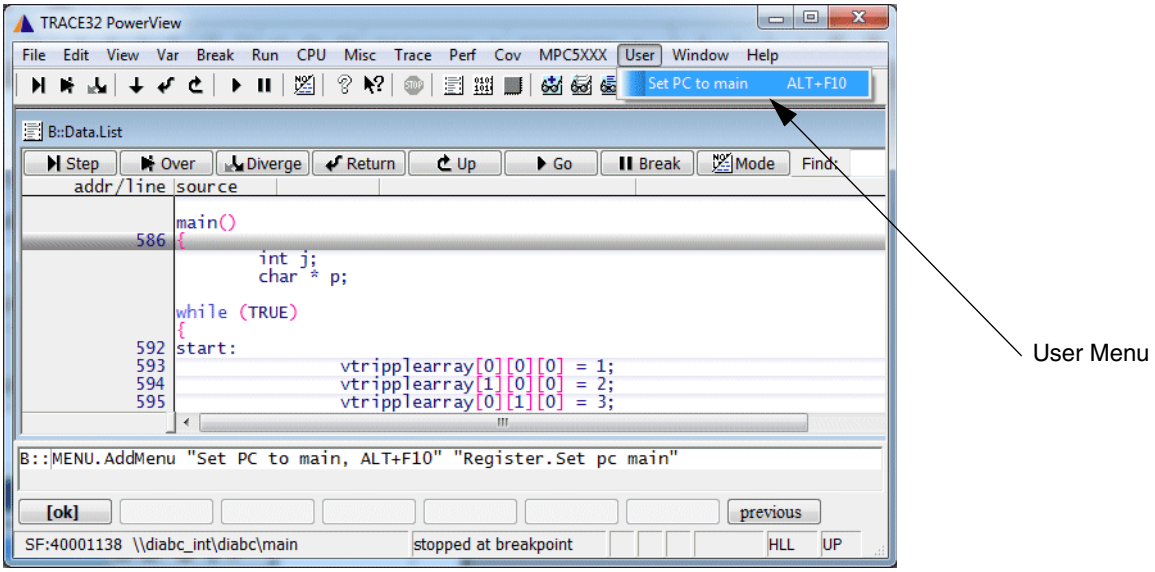
For often used commands accelerators are defined.



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).

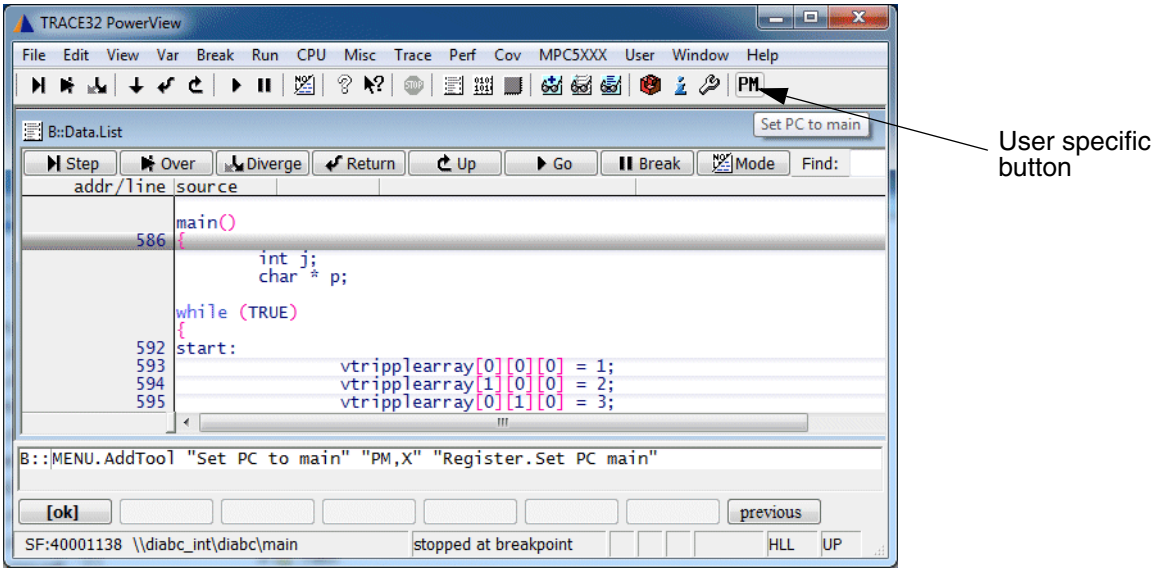
Videos about the menu programming can be found here:
**https://www.lauterbach.com/tut_customization.html**

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 |

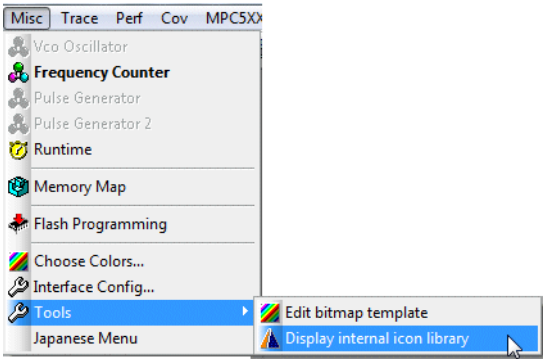
```
; <tooltip text> here:    Set PC to main
; <tool image> here:     button with capital letters PM in black
; <command> here:        Register.Set PC main

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

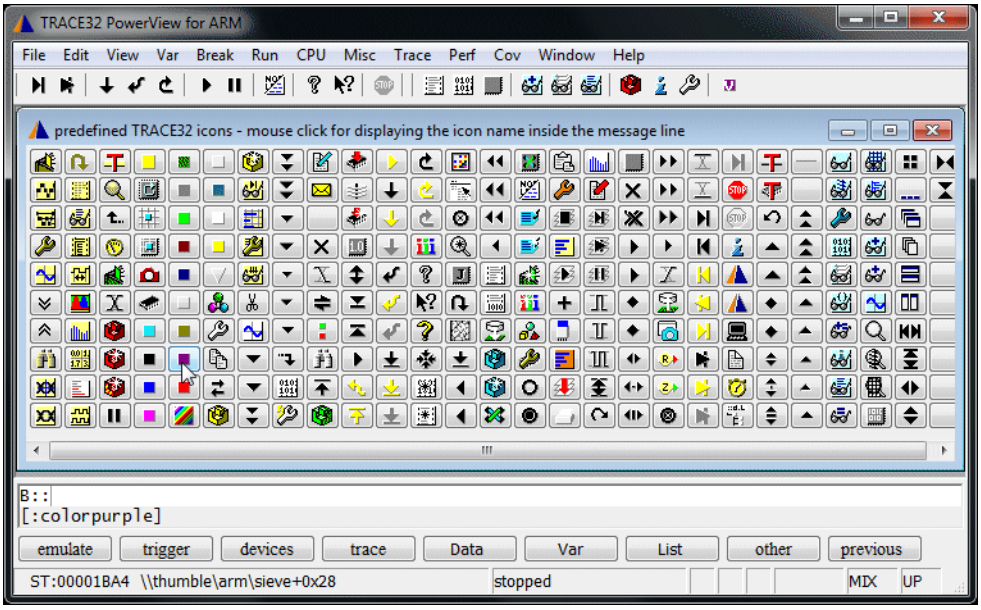



Information on the *<tool image>* can be found in ***Help -> Contents***

***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.

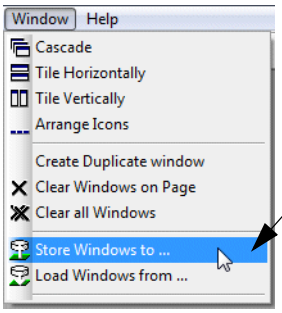
```
; overprint the icon colorpurple with the character v in White color
Menu.AddTool "Set PC to main" "v,W,colorpurple" "Register.Set PC main"
```

For more complex changes to the main tool bar refer to **"Training Menu"** (training_menu.pdf).

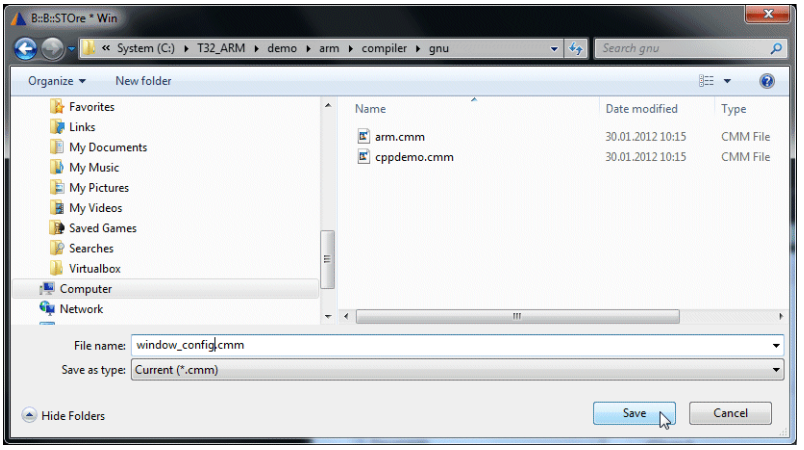
Videos about the menu programming can be found here:
**https://www.lauterbach.com/tut_customization.html**

## 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.

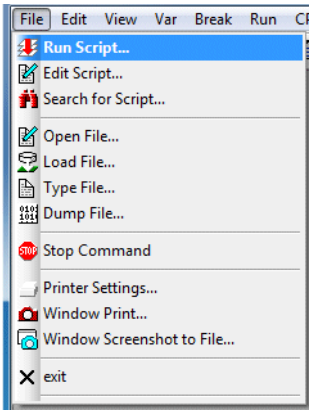


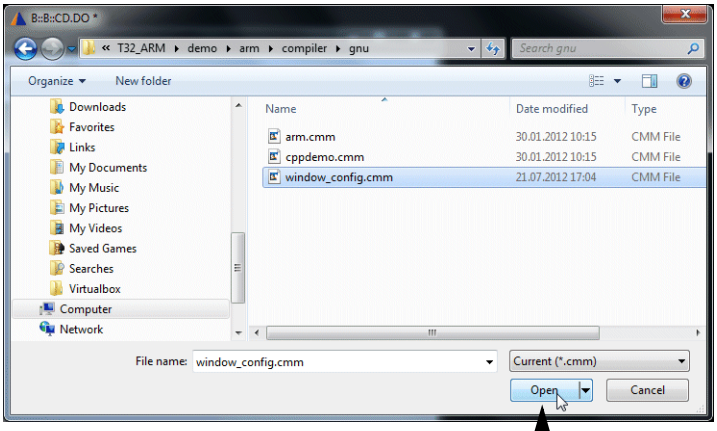

***Store Windows to …*** generates a script, that allows you to reactivate the window-configuration at any time.



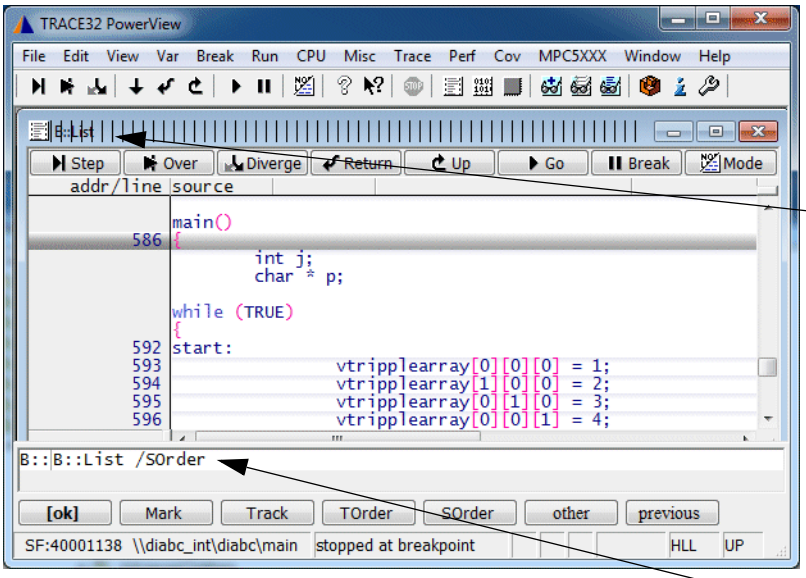

Script example:

```
// 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
```



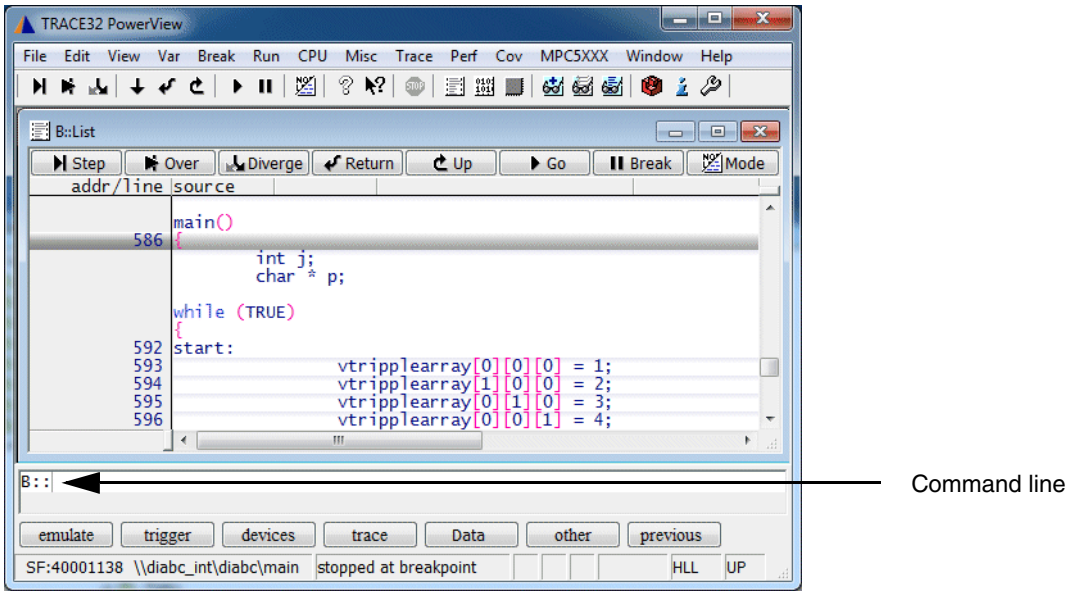




Run the script to reactivate the stored window-configuration




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

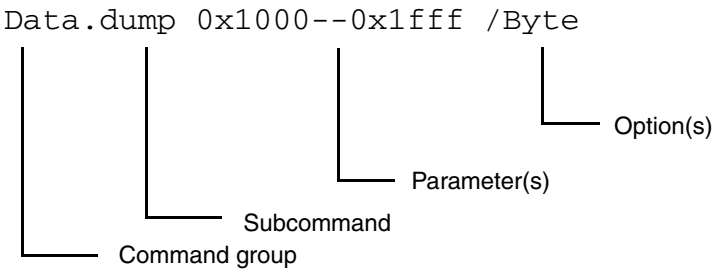
By clicking with the right mouse button to the window header, the command which was executed to open the window is re-displayed in the command line and can be modified there




## Command 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:

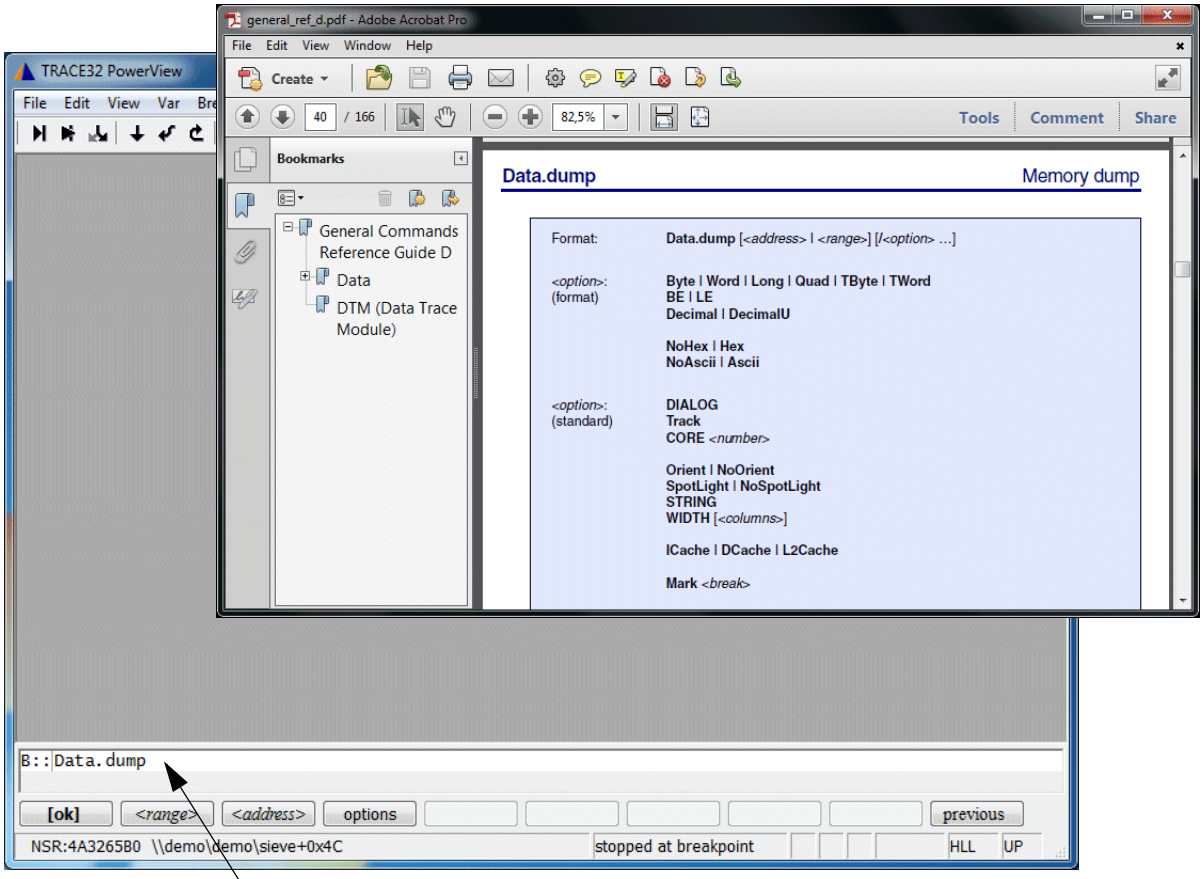


| Data | Command group to display, modify … memory |
|---|---|
| `Data.dump` | Displays a hex dump |
| `Data.Set` | Modify memory |
| `Data.LOAD.auto` | Loads code to the target memory |

| Break | Command group to set, list, delete … breakpoints |
|---|---|
| `Break.Set` | Sets a breakpoint |
| `Break.List` | Lists all set breakpoint |
| `Break.Delete` | Deletes a breakpoint |

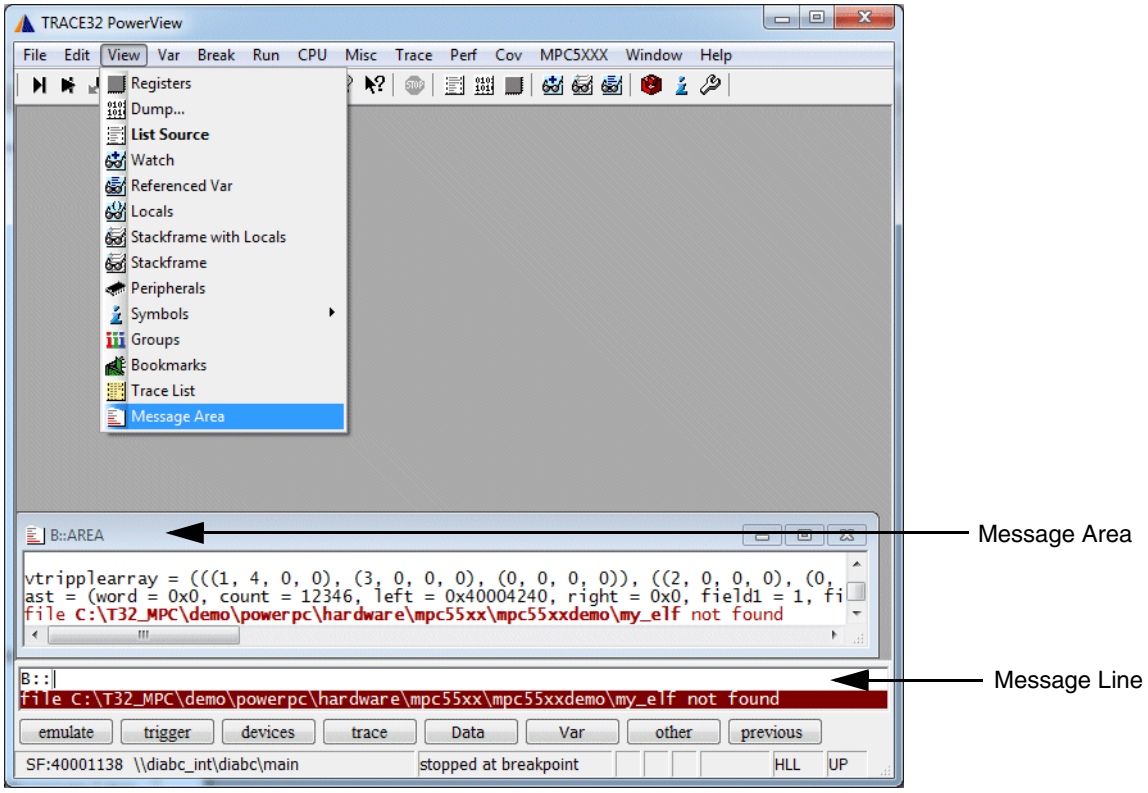
Each command can be abbreviated. The significant letters are always written in upper case letters.

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.

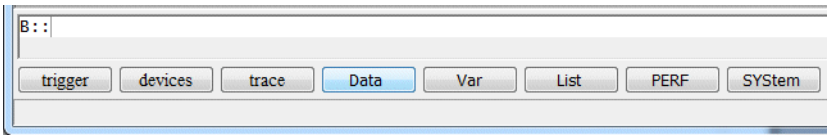
# Message Line



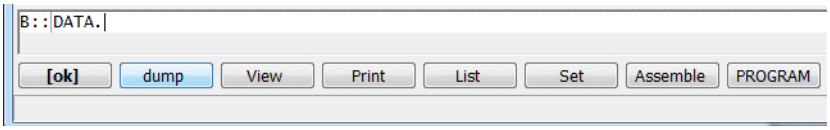

- **Message line** for system and error messages
- **Message Area window** for the display of the last system and error messages

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

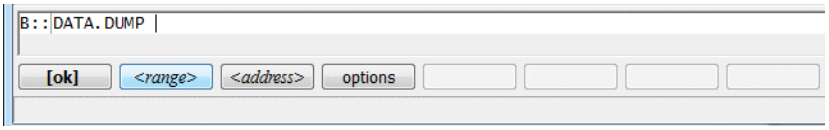
Select the command group, here **Data.**



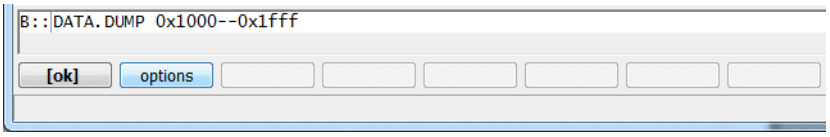

Select the subcommand, here **dump.**



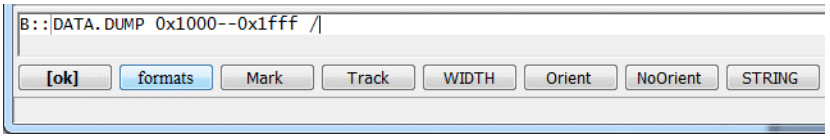

Angle brackets request an entry from the user,
here e.g. the entry of a <range> or an <address>.



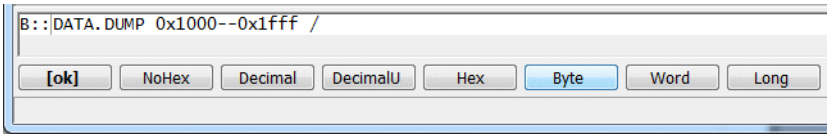

The display of the hex. dump can be adjusted to your needs by an option.



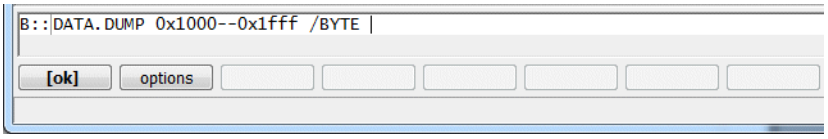

Select the option **formats** to get a list of all format options.

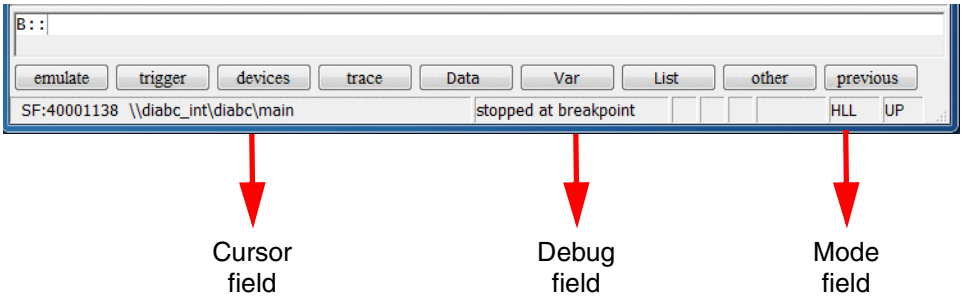



Select a format option, here **Byte**.




The command is complete now.






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.

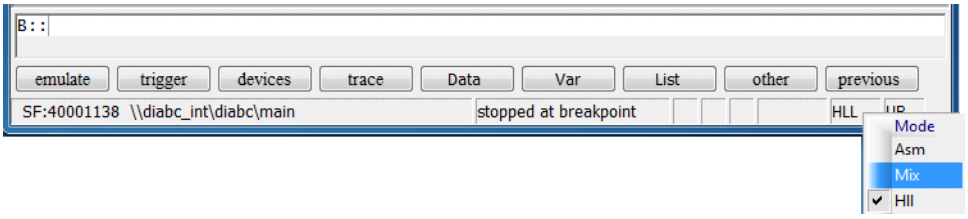
The **Debug** field of the state line provides:

- 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

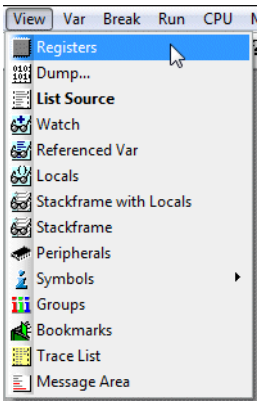
  It also defines how single stepping is performed (assembler line-wise or programming language line-wise).

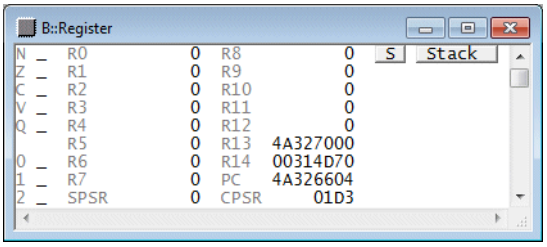



The debug mode can be changed by using the **Mode** pull-down.

# Core Registers

## Display the Core Registers



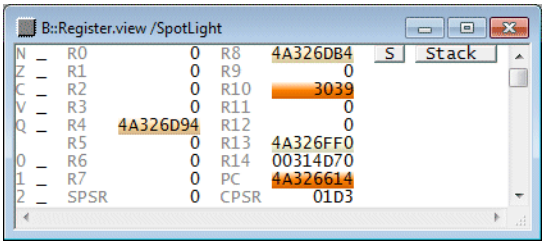




```
Register.view
```

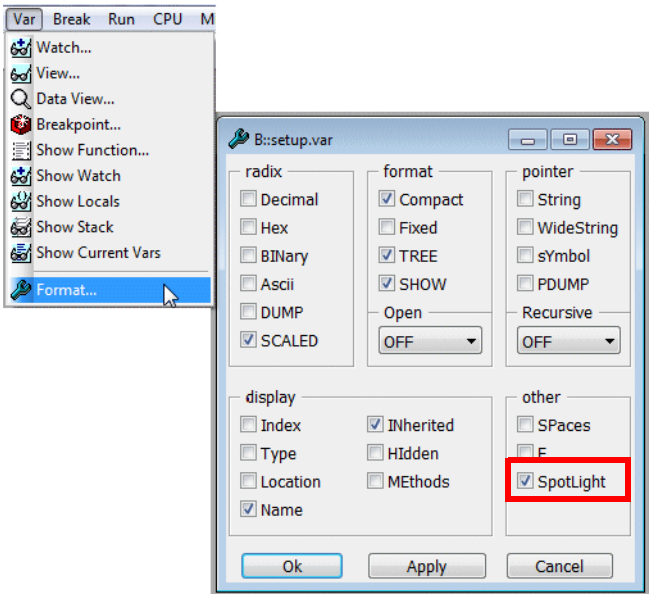
# Colored Display of Changed Registers

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.
```

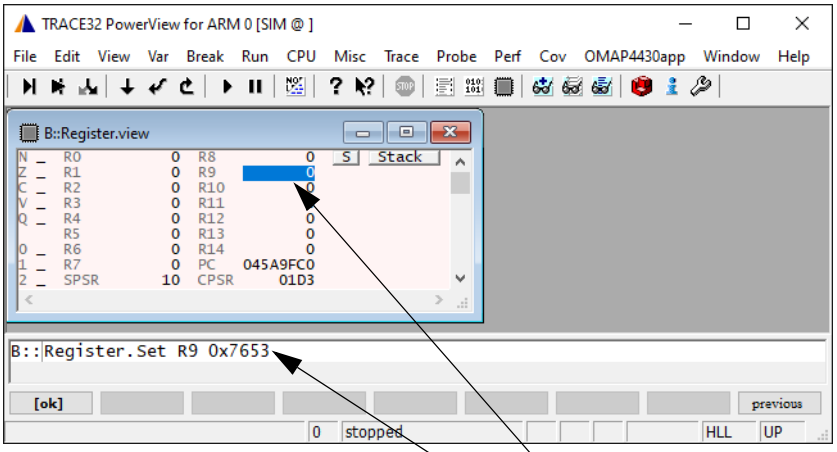



### Establish /SpotLight as default setting




| | |
|---|---|
| **SETUP.Var %SpotLight** | Establish the option SpotLight as default setting for<br>- all Variable windows<br>- Register window<br>- PERipheral window<br>- the HLL Stack Frame<br>- 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.

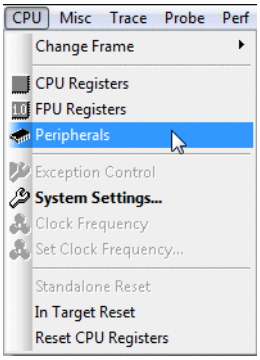
**Register.Set** *<register> <value>* Modify register

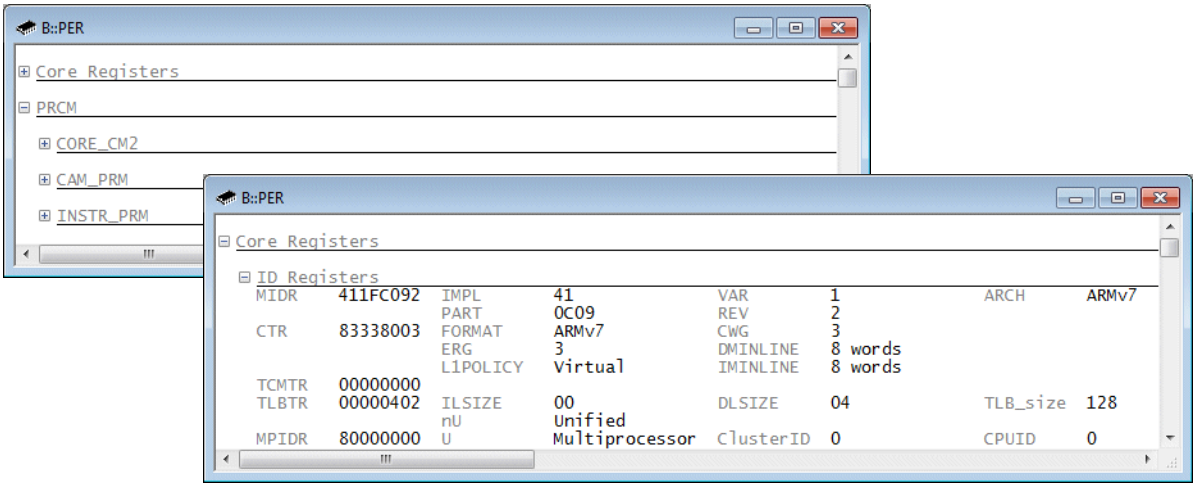
## Display the Special Function Registers

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

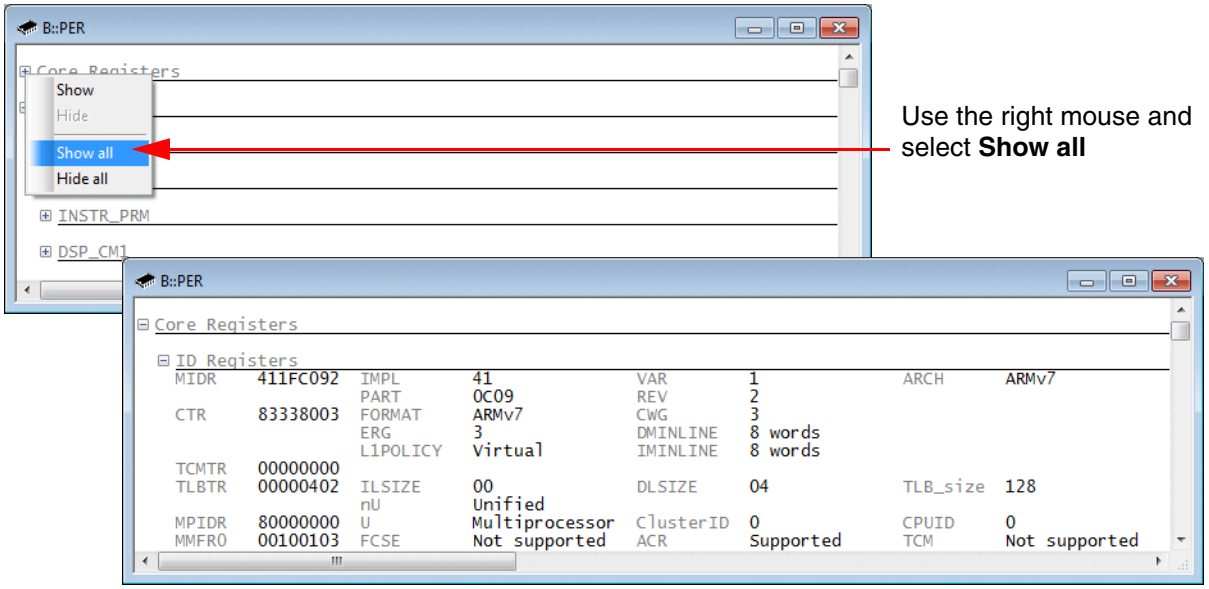
The individual configuration registers/on-chip peripherals are organized by TRACE32 PowerView in a tree structure. On demand, details about a selected register can be displayed.




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.

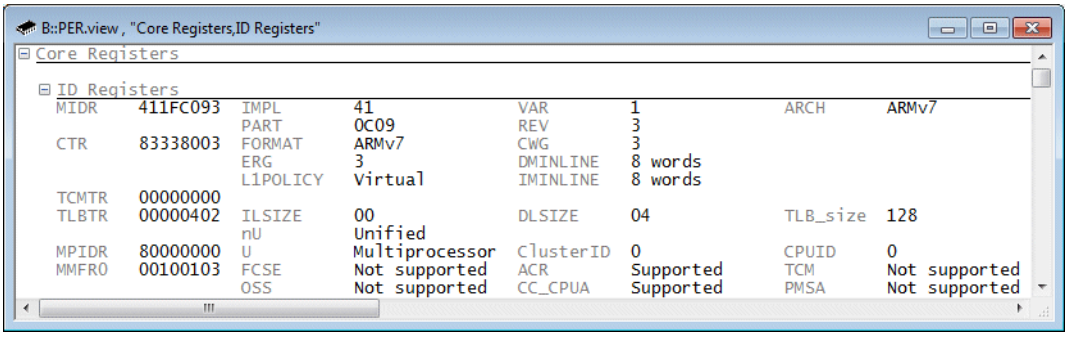



**Commands:**

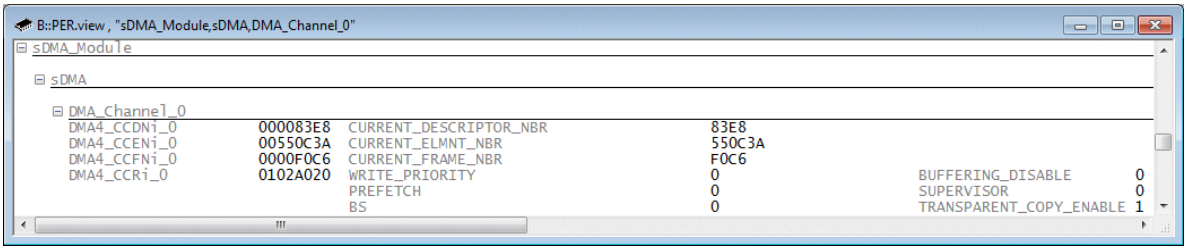
| | |
|---|---|
| **PER.view** *<filename>* [*<tree_item>*] | Display the configuration registers/on-chip peripherals |

```
; 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"
```

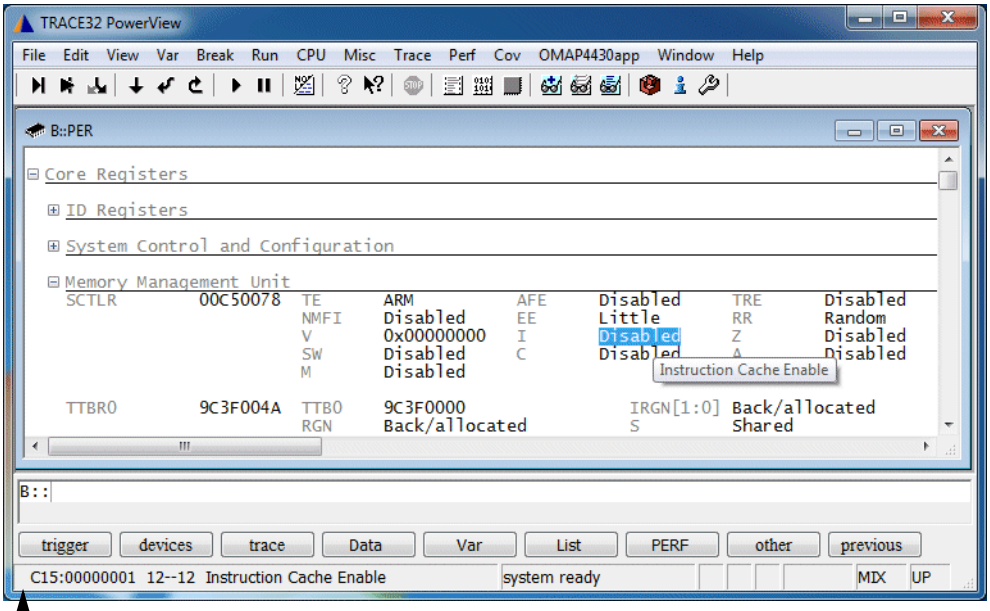



```
; Display the functional unit "DMA_Channel_0" within "sDMA_Module,sDMA"
; in expanded mode
PER.view , "sDMA_Module,sDMA,DMA_Channel_0"
```




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

```
; PRinTer.FileType ASCIIE              ; 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
```

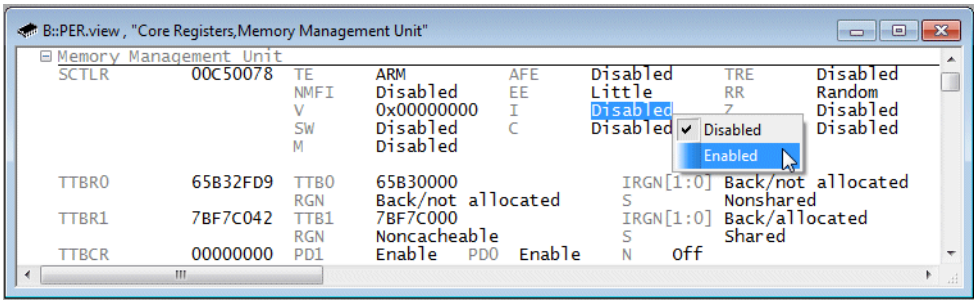



The access class, address, bit position and the full name of the selected item are displayed in the state line; the full name of the selected item is taken from the processor/chip manual.

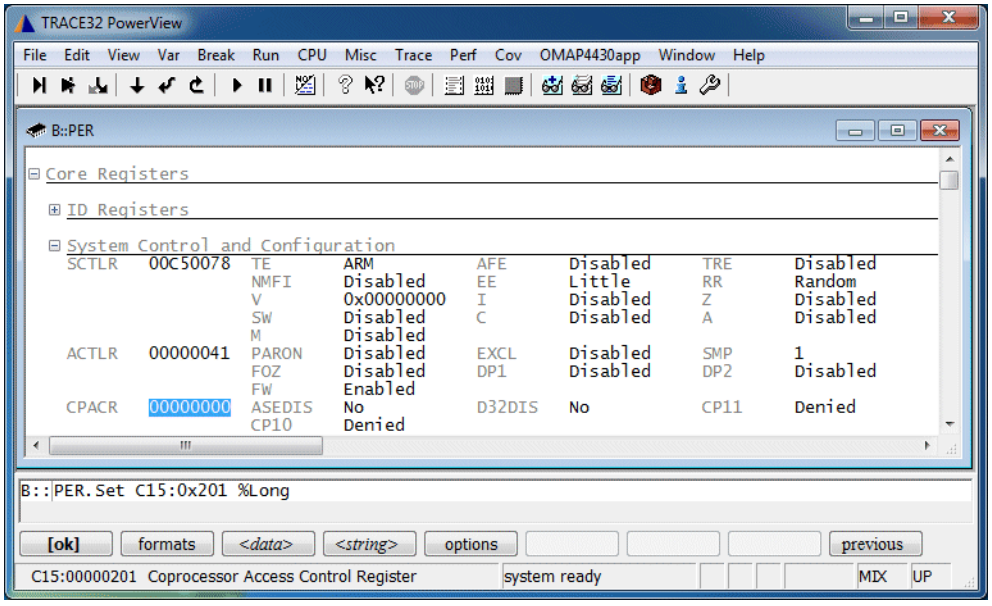
# Modify a Special Function Register

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.

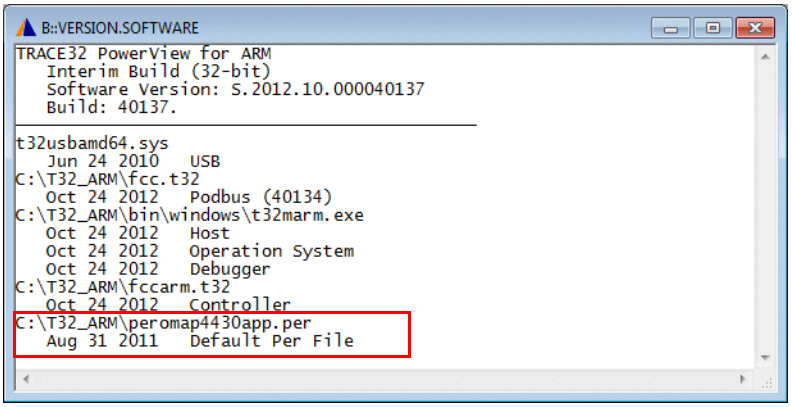
```
PER.Set.simple D:0xF87FFF10 %Long 0x00000b02
```

The 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:

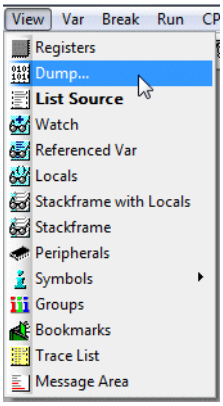
- The **Data.dump** command, that displays a hex dump of a memory area, and the **Data.Set** command that allows to modify the contents of a memory address.
- The **List** (former **Data.List)** command, that displays the memory contents as source code listing.

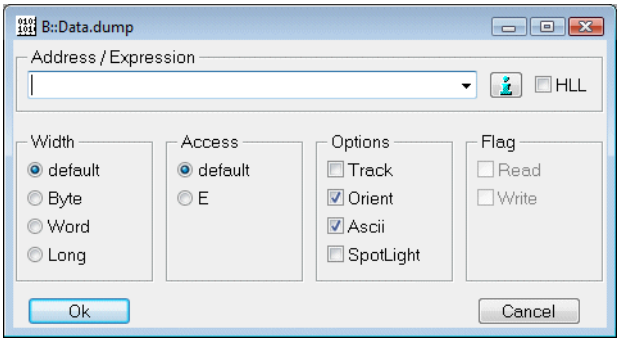
A so-called **access class** is always displayed together with a memory address. The following access classes are available for all processor architectures:

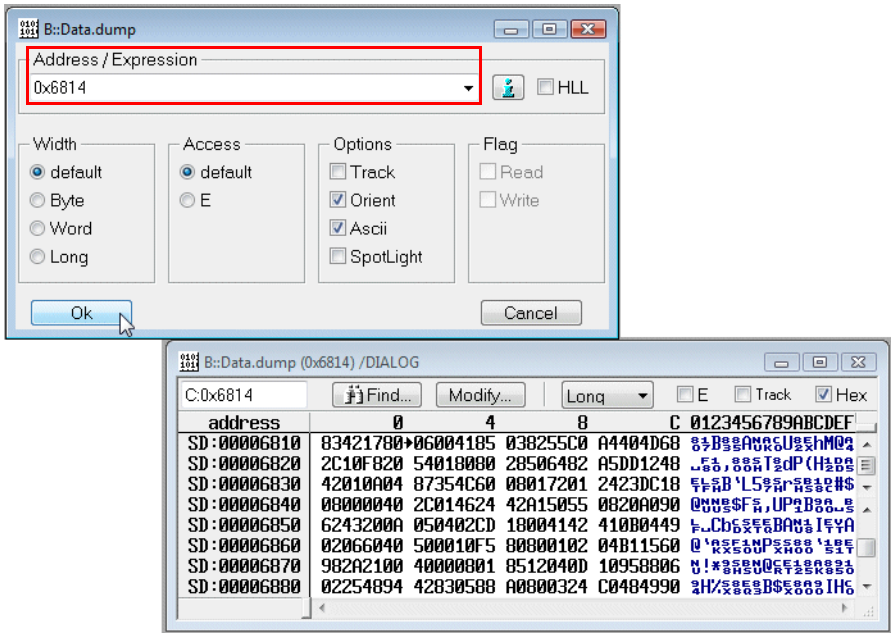
| | |
|---|---|
| **P:**1000 | **Program** address 0x1000 |
| **D:**6814 | **Data** address 0x6814 |

For additional access classes provided by your processor architecture refer to your **"Processor Architecture Manuals"**.

## Display the Memory Contents



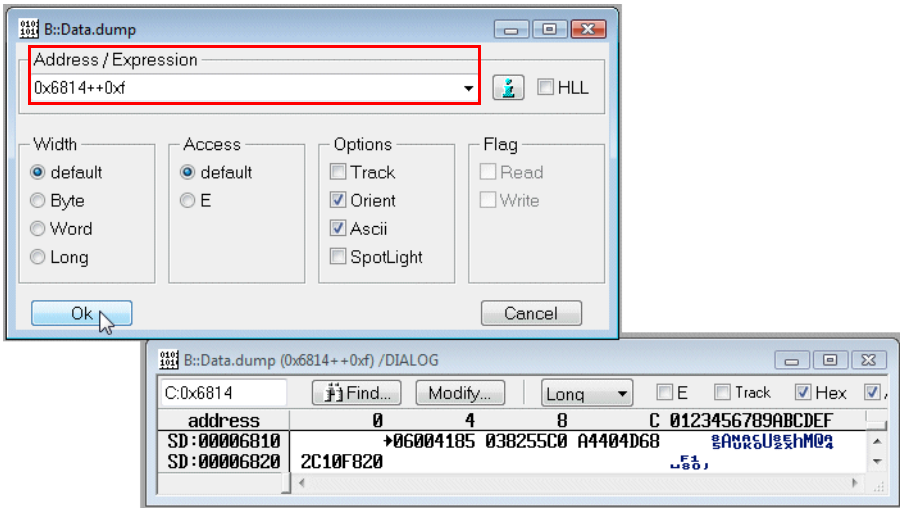



Please be aware, that TRACE32 permanently updates all windows. The default update rate is 10 times per second.

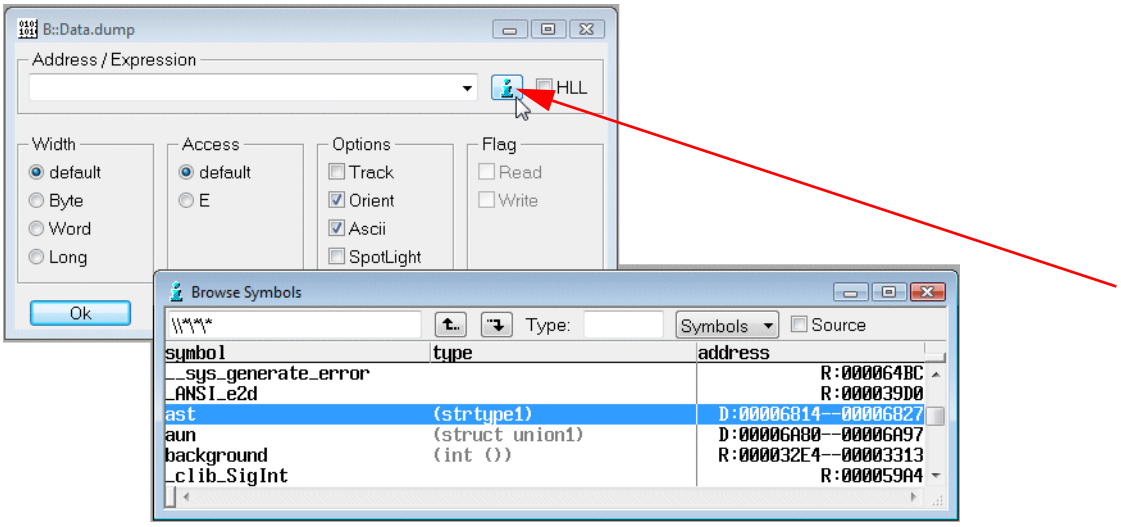
If you enter an address range, only data for the specified address range are displayed. This is useful if a memory area close to memory-mapped I/O registers should be displayed and you do not want TRACE32 PowerView to generate read cycles for the I/O registers.

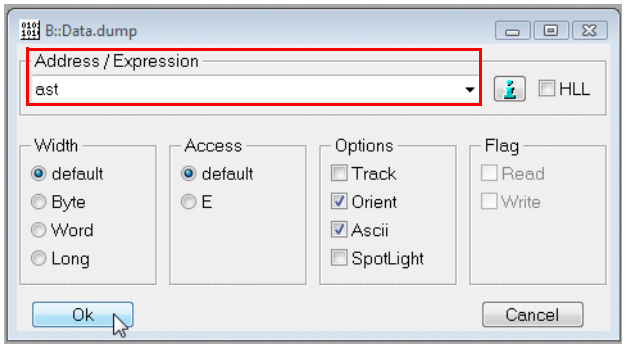
**Conventions for address ranges:**

- <start_address>--<end_address>
- <start_address>..<end_address>
- <start_address>++<offset_in_byte>
- <start_address>++<offset_in_word> (for DSPs)



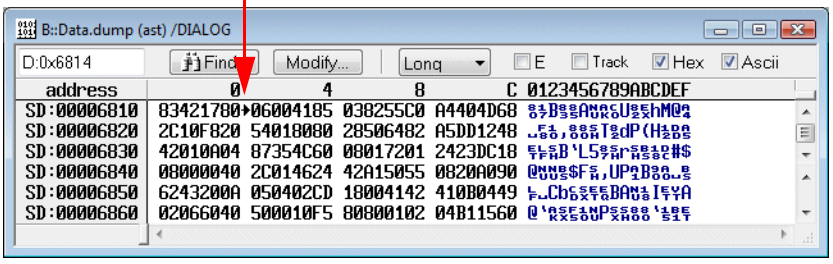
Use **i** to select any symbol name or label known to TRACE32 PowerView.



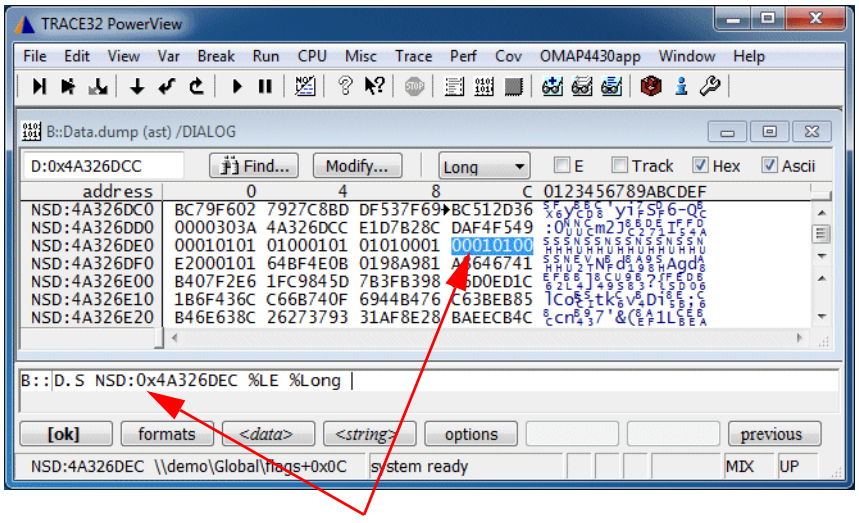

By default an oriented display is used (line break at $2^x$).

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
```

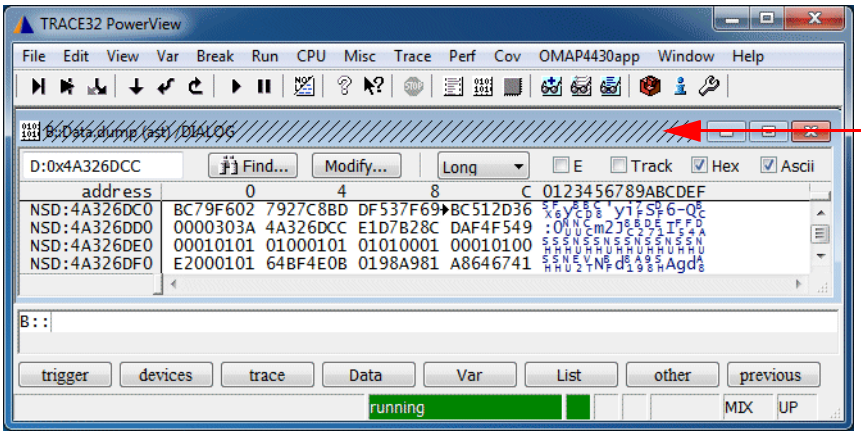



By a left mouse double-click to the memory contents a **Data.Set** command is automatically displayed in the command line, you can enter the new value and confirm it with return.

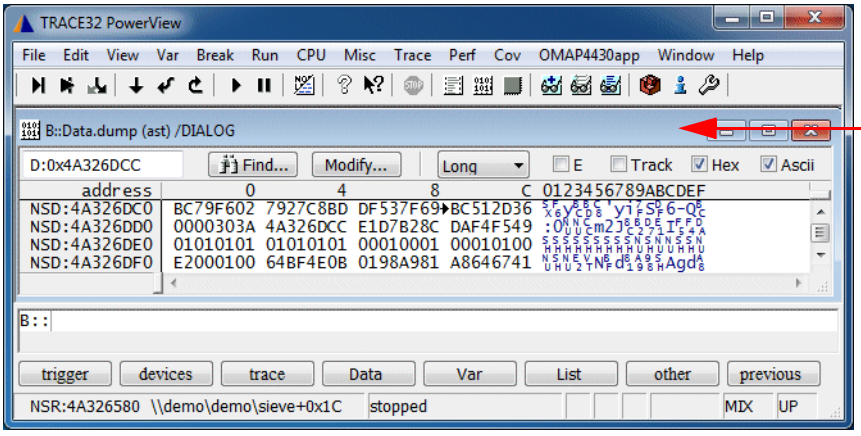
**Data.Set** *<address>*|*<range>* [**%***<format>*] *<value>* [**/***<option>*]

```
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
```

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



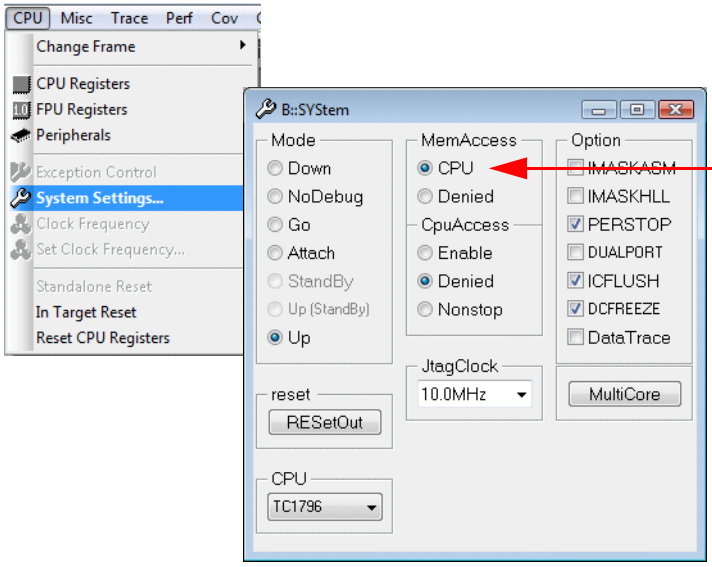

A hatched window frame indicates that the information display is brozen because the core is executing the program.




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**

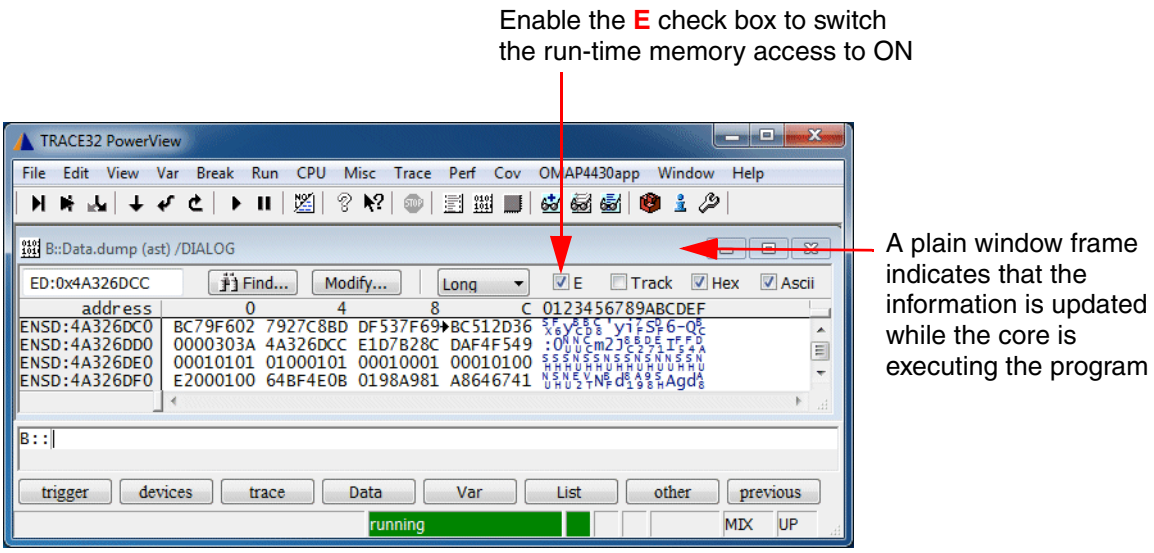
Tightly-coupled memory might not be accessible via the system memory bus.

## 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).

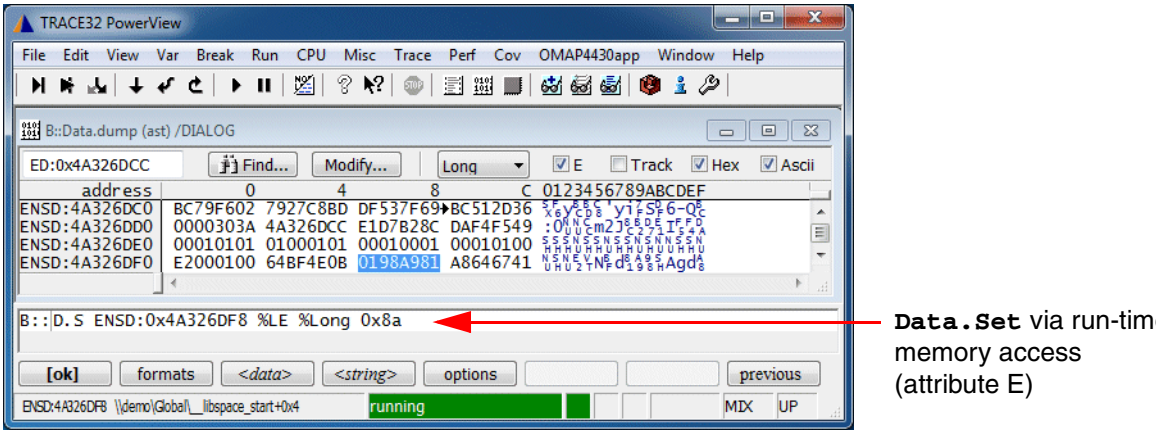
**Configure the run-time memory access for a specific memory area:**




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

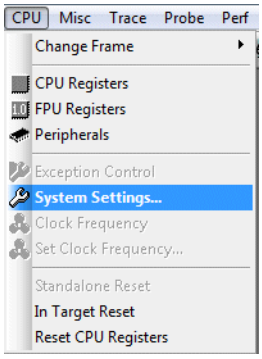
| | |
|---|---|
| **EP:**1000 | **Program** address 0x1000 with run-time memory access |
| **ED:**6814 | **Data** address 0x6814 with run-time memory access |

Write accesses to the memory work correspondingly:

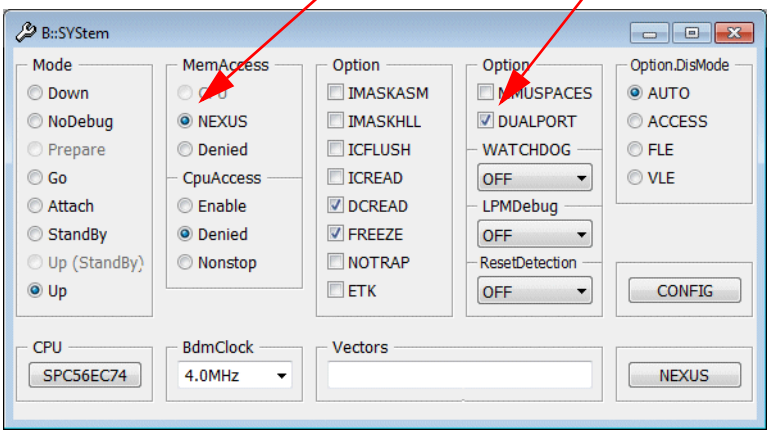


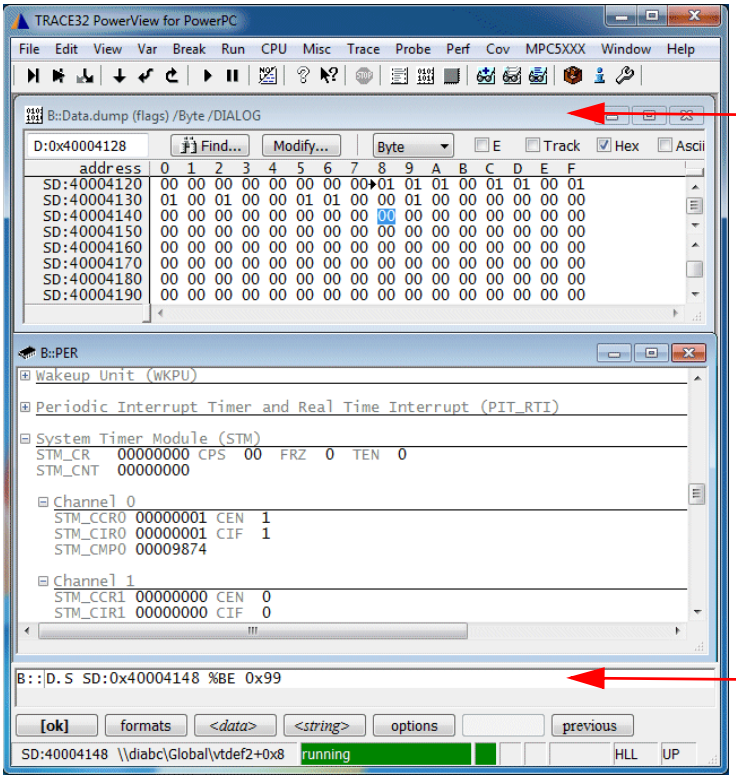
```
SYStem.MemAccess Enable                 ; Enable the non-intrusive
                                        ; run-time memory access

;…

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



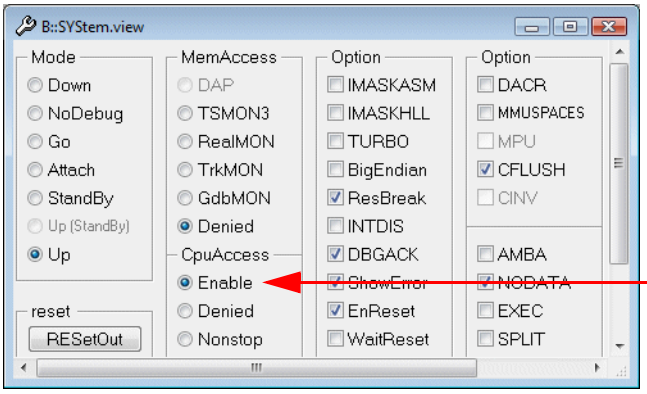




All windows that display memory have a plain window frame, because they are updated while the core is executing the program

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

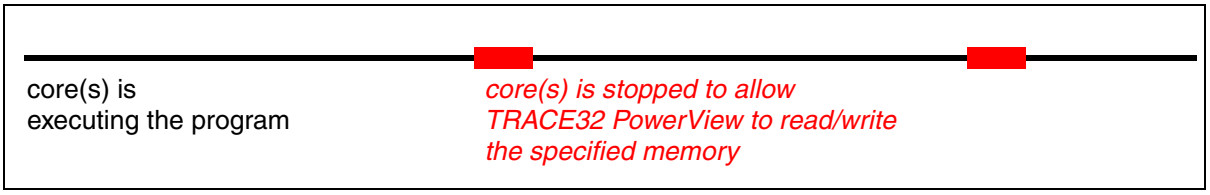
```
SYStem.MemAccess Enable                 ; Enable the non-intrusive
                                        ; run-time memory access

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**.

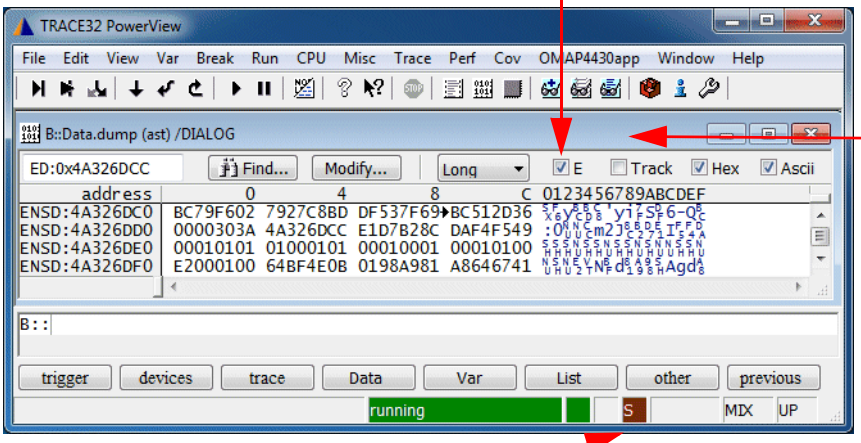



The time taken by a short stop depends on various factors:

- 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 assesses 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**.

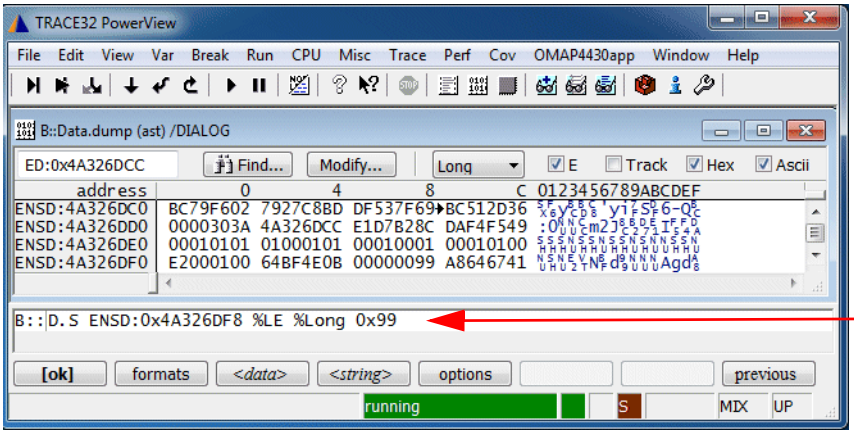
Enable the **E** check box to switch the run-time memory access to ON




A plain window frame indicates that the information is updated while the core(s) is executing the program

A red **S** in the state line indicates that a TRACE32 feature is activated that requires short-time stops of the program execution

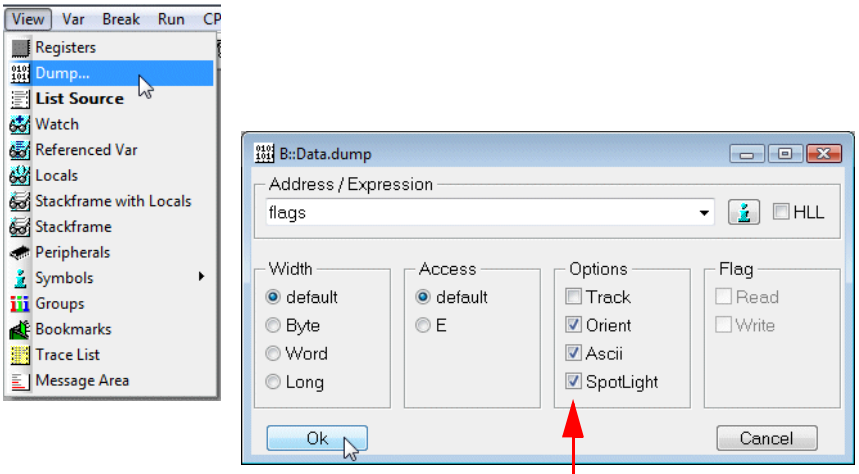
Write accesses to the memory work correspondingly:



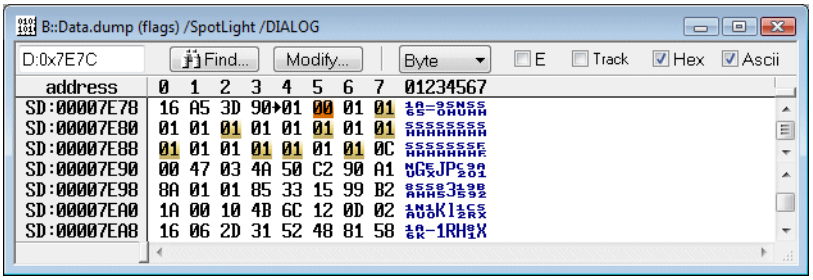

**Data.Set** via run-time memory access with short stop of the program execution

```
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

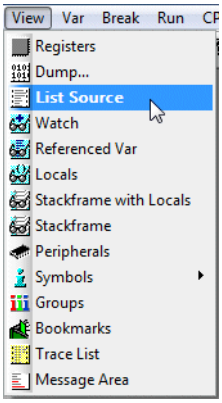


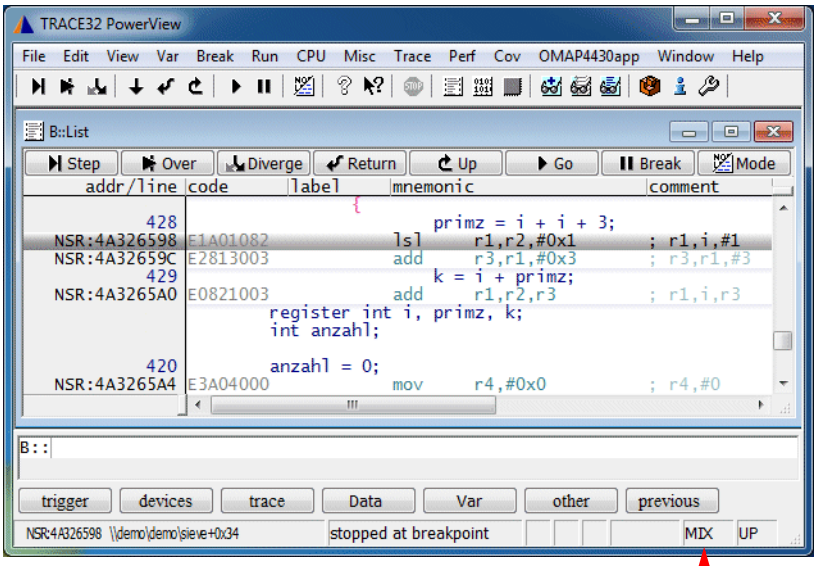

Enable the option **SpotLight** to mark the memory contents changed by the last 4 single steps in orange, older changes being lighter.



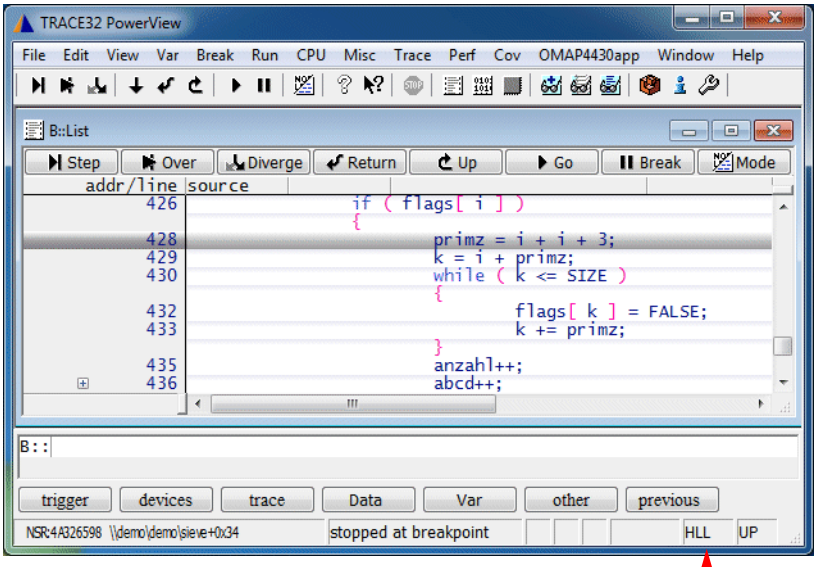

```
Data.dump flags /SpotLight            ; Display a hex dump starting at
                                      ; the address of the label flags

                                      ; Mark changes
```

## Displays the Source Listing Around the PC



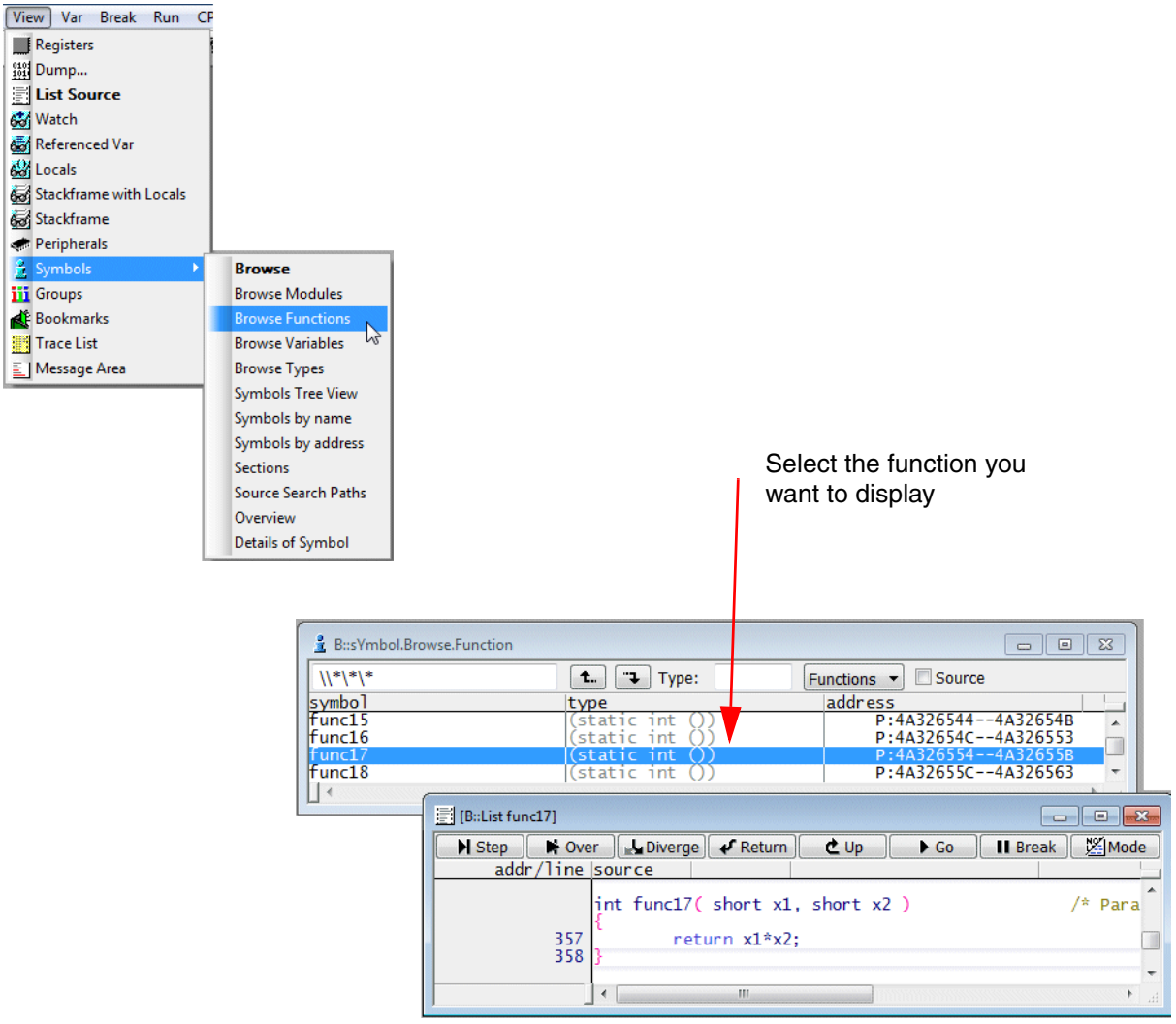




If MIX mode is selected for debugging, assembler and HLL information is displayed




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**

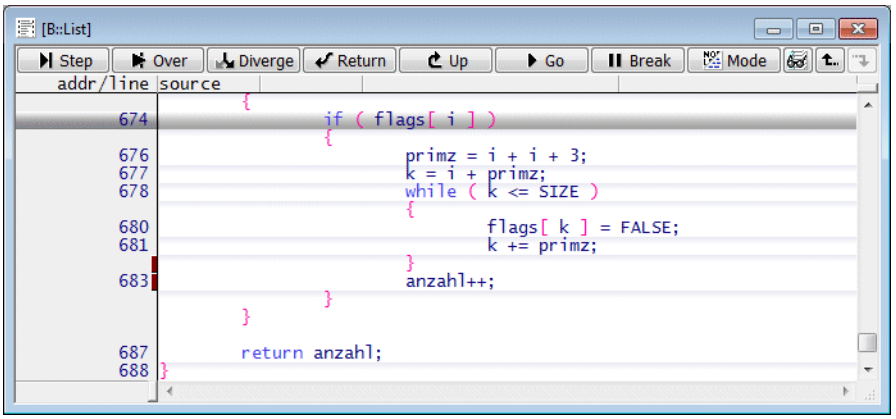
## Breakpoint Implementations

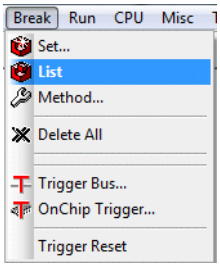
A debugger has two methods to realize breakpoints: Software breakpoints and Onchip breakpoints.

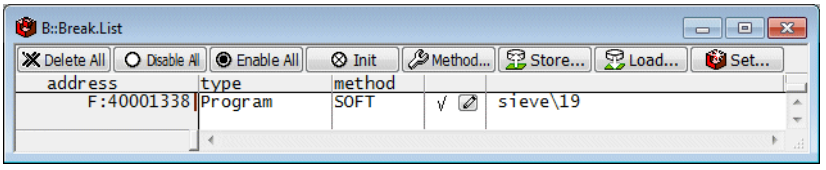
### Software Breakpoints in RAM

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.

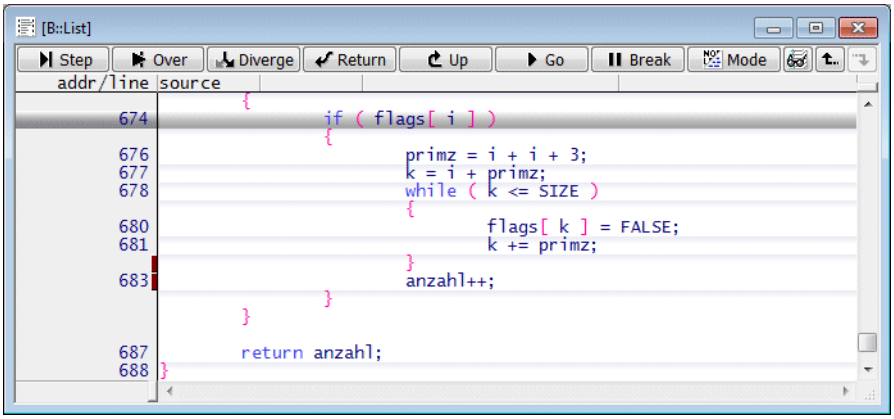
| | |
|---|---|
|  | Please be aware that TRACE32 PowerView always tries to set an Onchip breakpoint, when the setting of a Software Breakpoint fails. |

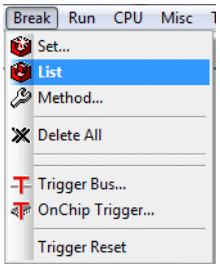
## Software Breakpoints in FLASH

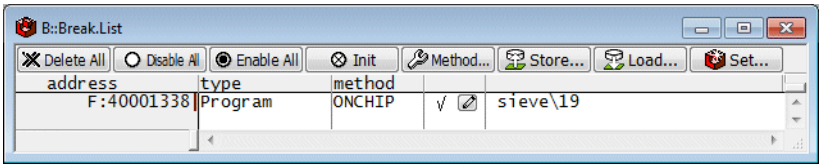
TRACE32 allows to set Software breakpoints to FLASH. Please be aware that the affected FLASH sector has to be erased and programmed in order to patch the break instruction used by the Software breakpoint. This usually takes some time and reduces the number of FLASH erase cycles. For details refer to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

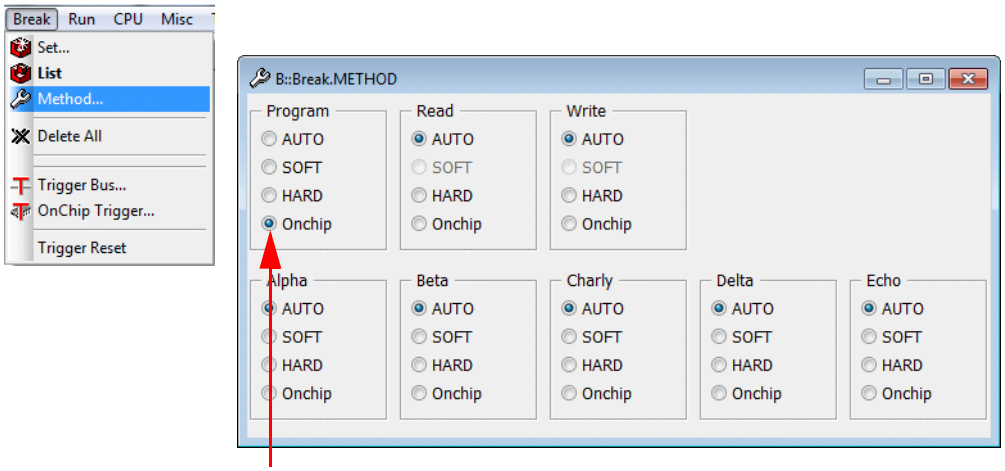


Since Software breakpoints are used by default for Program breakpoints, TRACE32 PowerView **can** be informed explicitly where to use Onchip breakpoints. Depending on your memory layout, the following methods are provided:

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**

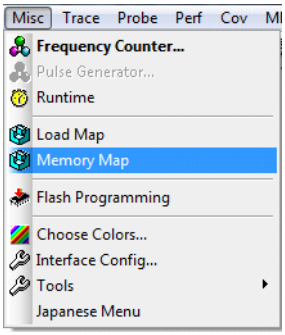
| | |
|---|---|
| **Break.METHOD Program Onchip** | Advise TRACE32 PowerView to implement Program breakpoints always as Onchip breakpoints |

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

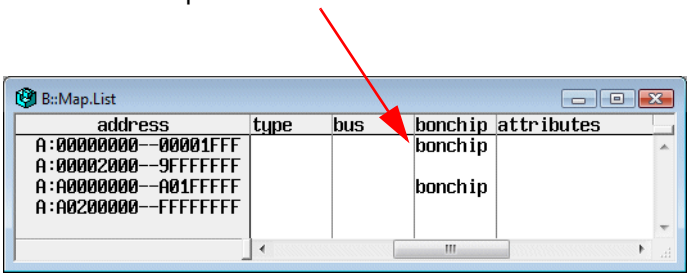
| | |
|---|---|
| **MAP.BOnchip** *<range>* | Advise TRACE32 PowerView to implement Program breakpoints as Onchip breakpoints within the defined address range |
| **MAP.List** | Check your settings |

```
MAP.BOnchip 0x0++0x1FFF

MAP.BOnchip 0xA0000000++0x1FFFFF
```

Check your settings as follows:

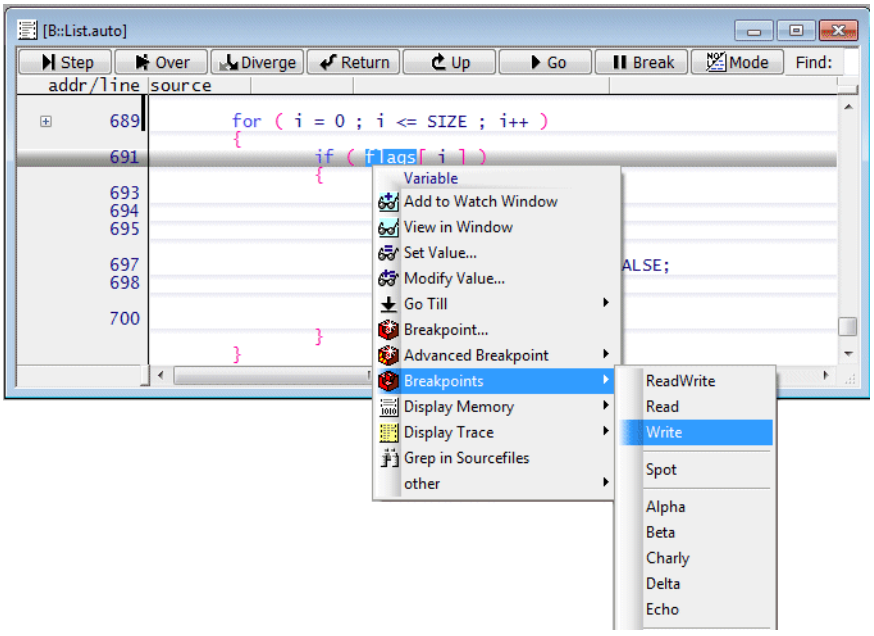


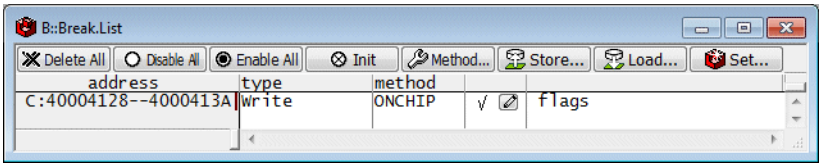

For the specified address ranges Program breakpoints are implemented as Onchip breakpoints. For all other memory areas Software breakpoints are used.



# 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.



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

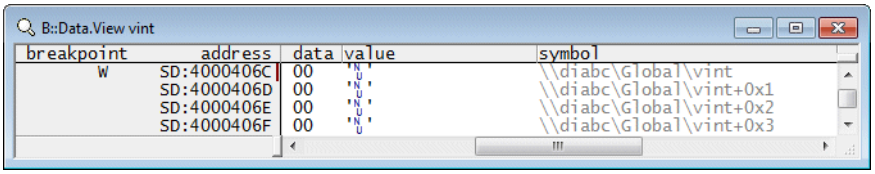
For some processor architectures Onchip breakpoints can only mark **single addresses** (e.g Cortex-A9). Most processor architectures, however, allow to mark **address ranges** with Onchip breakpoints. It is very common that one Onchip breakpoint marks the start address of the address range while the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

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

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

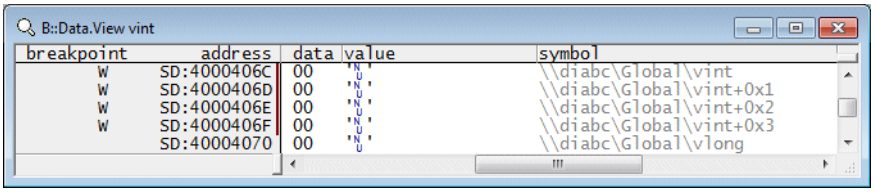
The current setting can be inspected and changed from the **Break.CONFIG** window.

**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
```

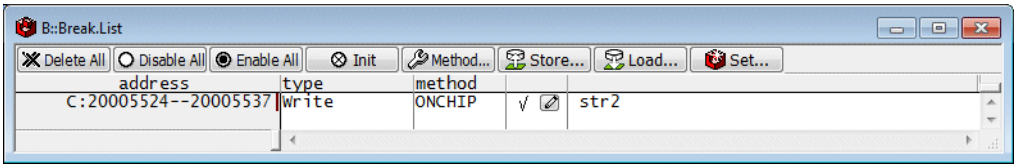



A number of processor architectures provide only **bit masks** or **fixed range sizes** to mark an address range with Onchip breakpoints. In this case the address range is always enlarged to the **smallest bit mask/next allowed range** that includes the address range.

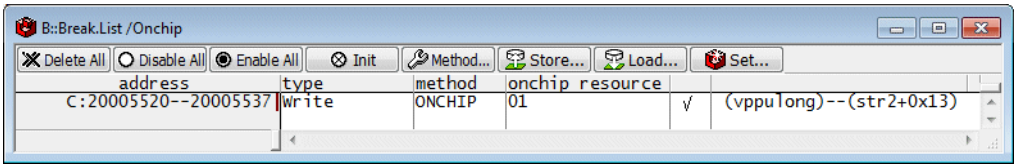
It is recommended to control which addresses are actually marked with breakpoints by using the **Break.List /Onchip** command:

Breakpoint setting:

```
Var.Break.Set str2

Break.List
```




```
Break.List /Onchip
```




## 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.

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

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

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

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

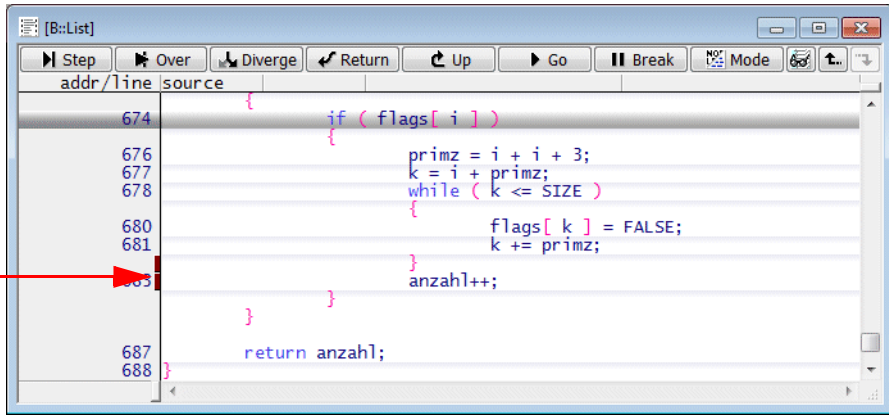
Please refer to the description of the **ETM.StoppingBreakPoints** command, if you want to use the ETM breakpoints.

# Breakpoint Types

TRACE32 PowerView provides the following breakpoint types for standard debugging.

| Breakpoint Types | Possible Implementations |
|---|---|
| **Program** | Software (Default)<br>Onchip |
| **Read, Write, ReadWrite** | Onchip (Default) |

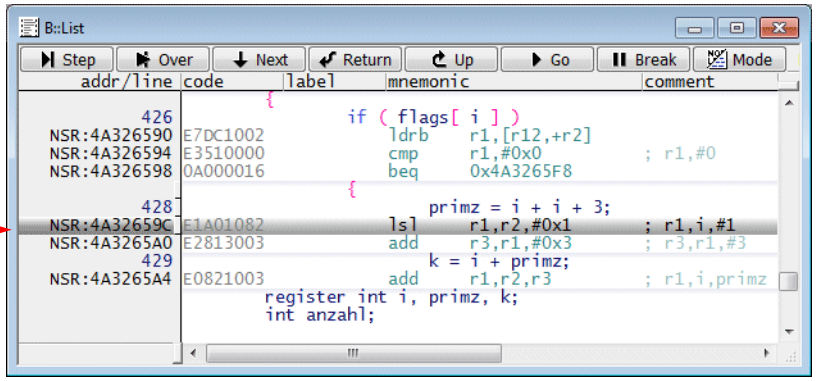
Set a Program breakpoint by a left mouse double-click to the instruction




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).

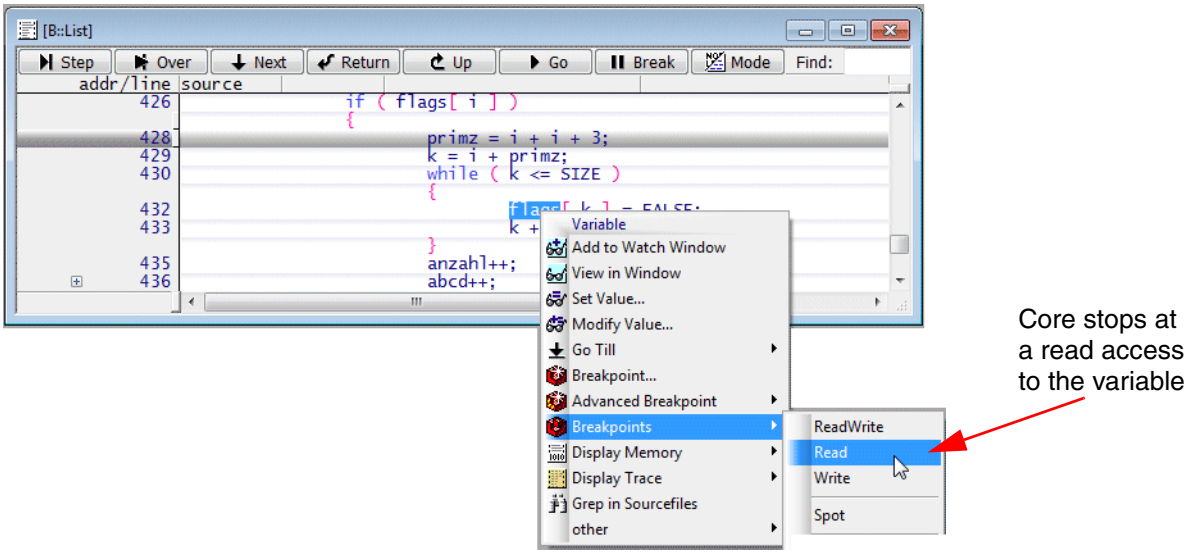
Disable the Program breakpoint by a left mouse double-click to the red program breakpoint indicator. The program breakpoint indicator becomes grey.

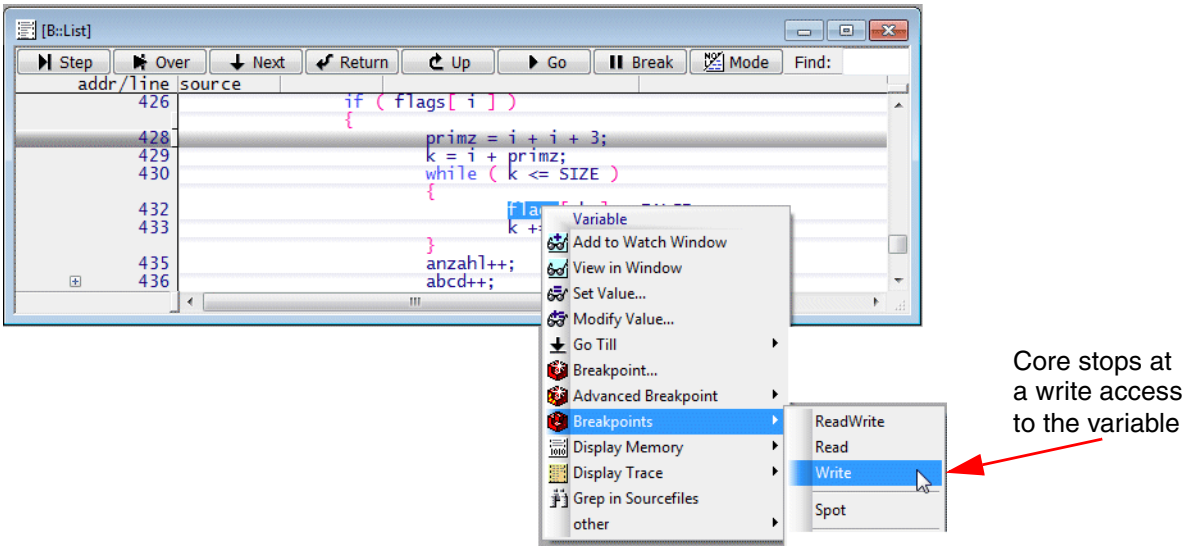



| | |
|---|---|
| **Break.Set** *<address>* **/Program** [**/DISable**] | Set a Program breakpoint to the specified address. The Program breakpoint can be disabled if required. |

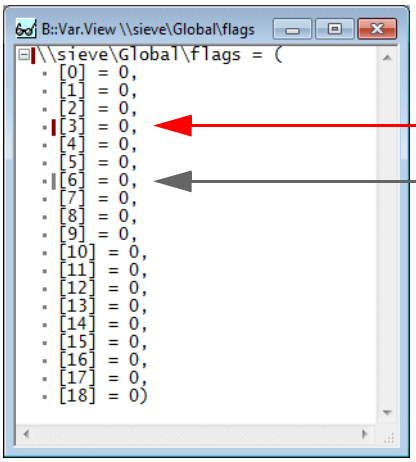
```
Break.Set 0xA34f /Program             ; set a Program breakpoint to
                                      ; address 0xA34f

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
                                      ; diable Program breakpoint

Break.List                            ; list all breakpoints
```

# Read/Write Breakpoints




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.

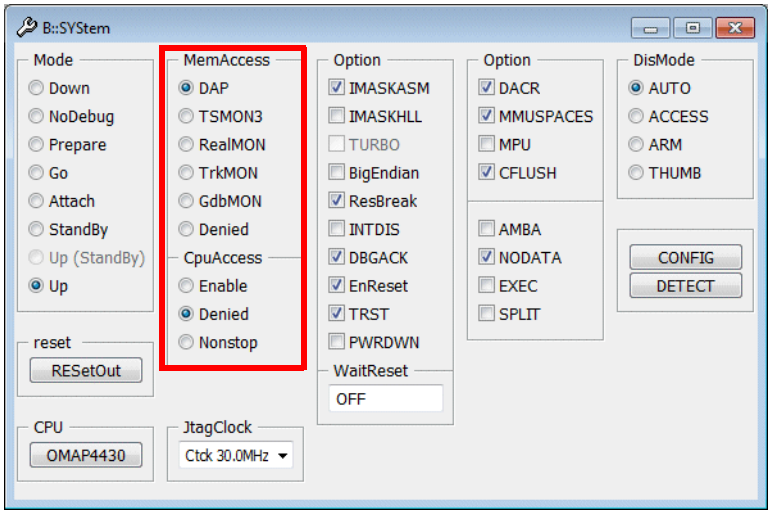
A small **grey breakpoint indicator**
marks a disabled Read/Write breakpoint.

**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 0x0B56 /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.

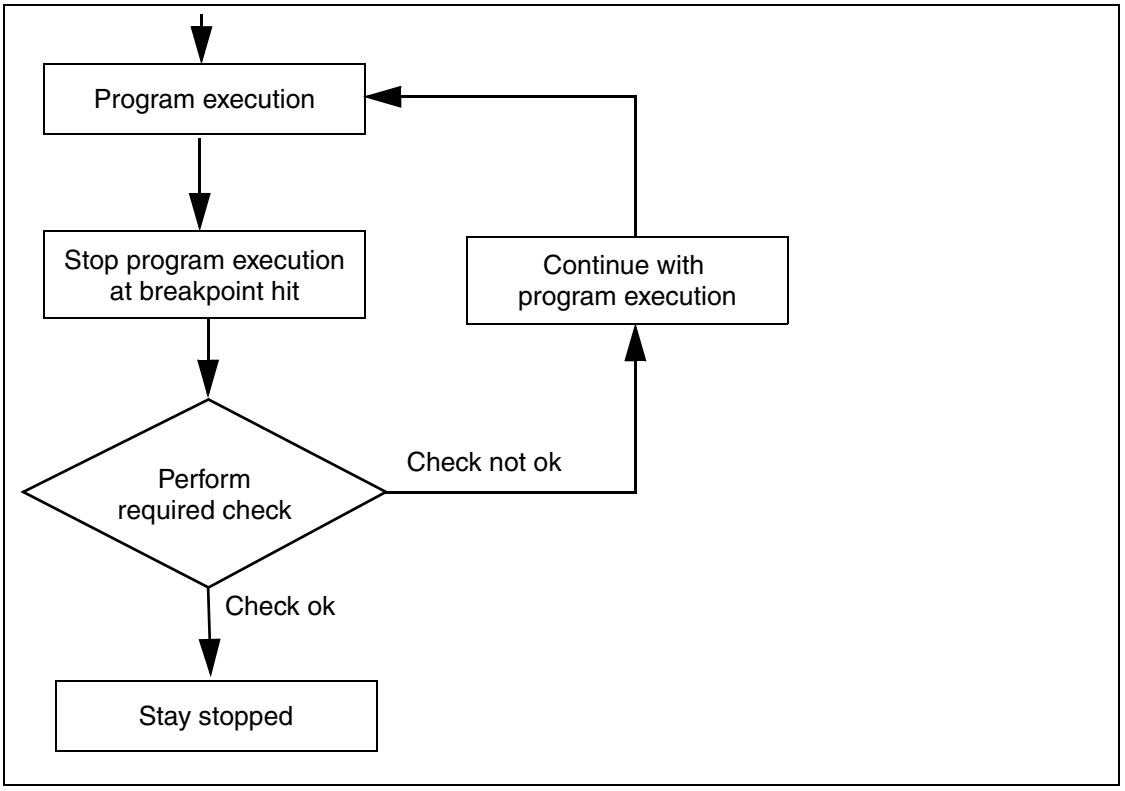
# Real-time Breakpoints vs. Intrusive Breakpoints

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.

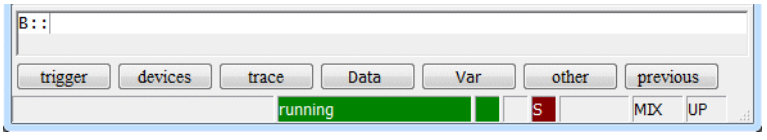
**Real-time breakpoints** do not disturb the real-time program execution on the core(s), but they require a complex on-chip break logic.

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)

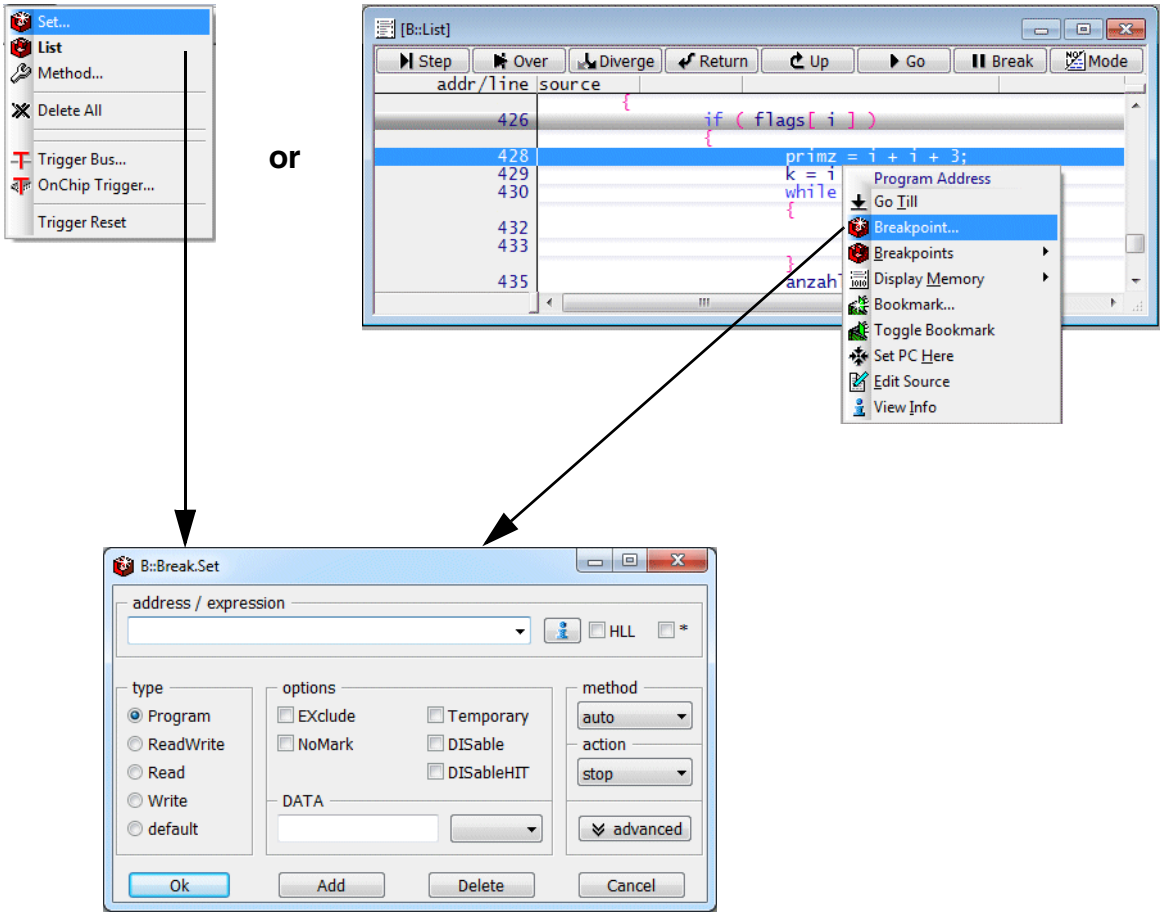



The (short-time) display of a red S in the state line indicates that an intrusive breakpoint was hit.

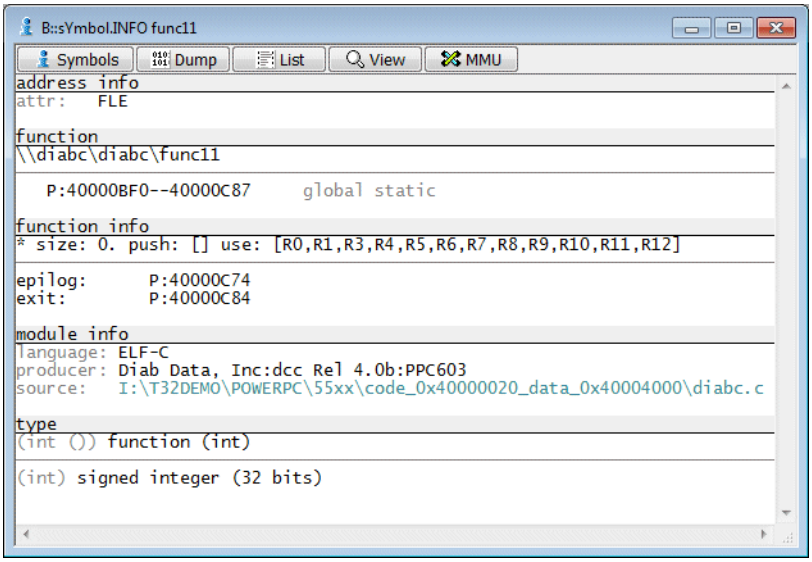
Intrusive breakpoints are marked with a special breakpoint indicator:

# Break.Set Dialog Box

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

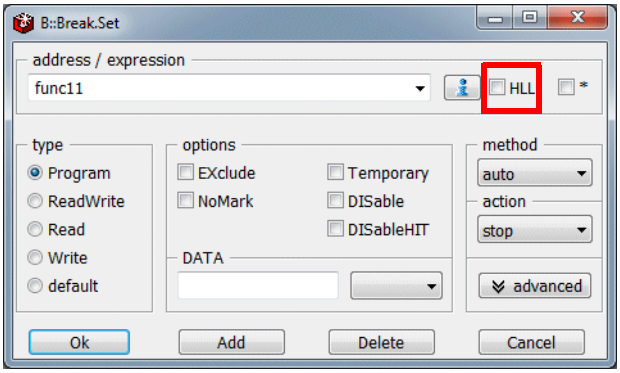


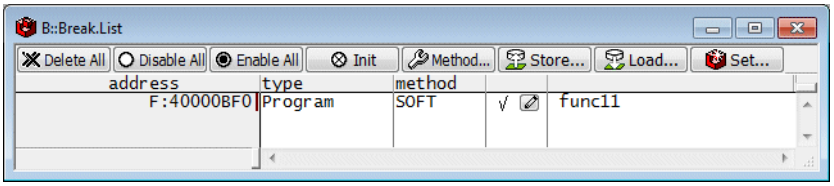
```
sYmbol.INFO func11                      ; display symbol information
                                        ; for function func11
```




## Function Name/HLL Check Box OFF

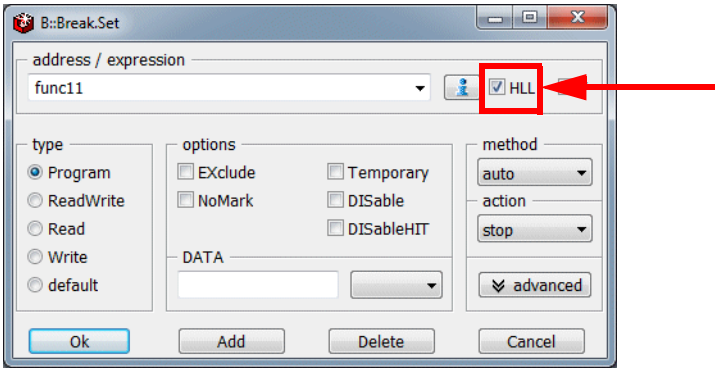
Program breakpoint is set to the function entry (first address of the function).

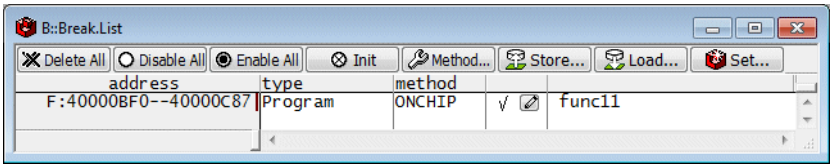


```
Break.Set func11
```

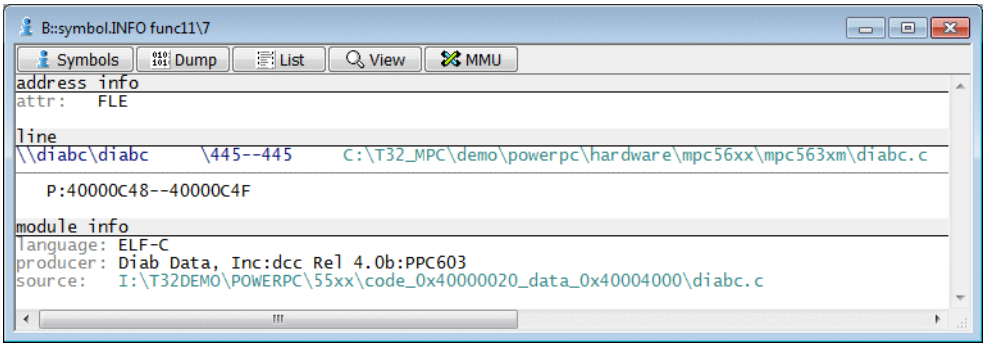
- If the on-chip break logic supports ranges for Program breakpoints, a Program breakpoint implemented as Onchip is set to the full address range covered by the function.
- If the on-chip break logic provides only bitmasks to realizes breakpoints on instruction ranges, a Program breakpoint implemented 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.



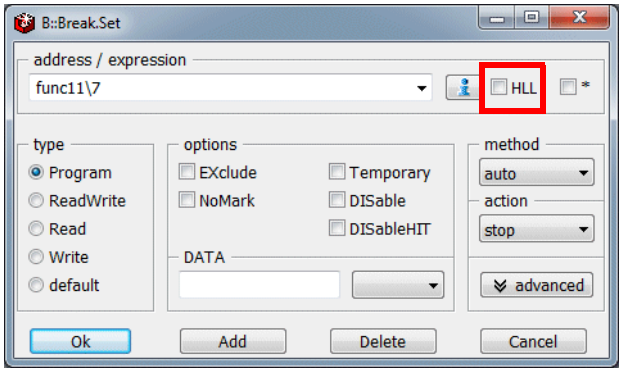

```
Var.Break.Set func11
```

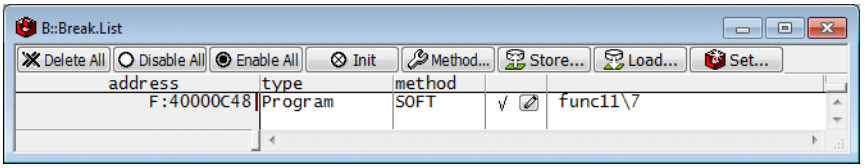
```
sYmbol.INFO func11\7                    ; display debug information
                                        ; for 7th program line in
                                        ; function func11
```




## Program Line Number/HLL Check Box OFF

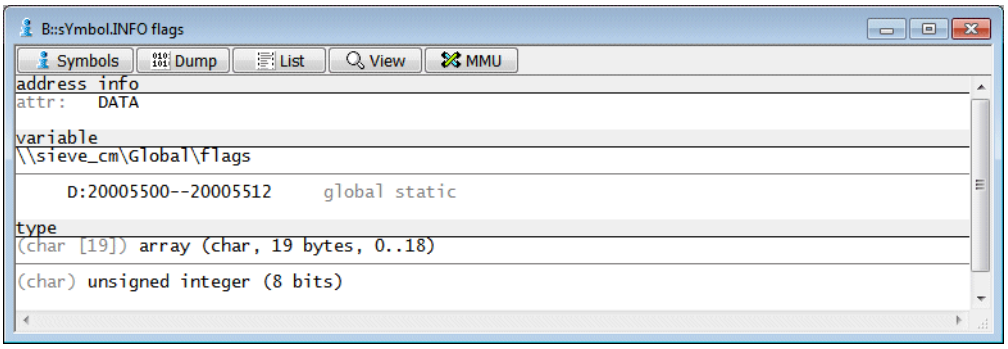
Program breakpoint is set to the first assembler instruction generated for the program line number.



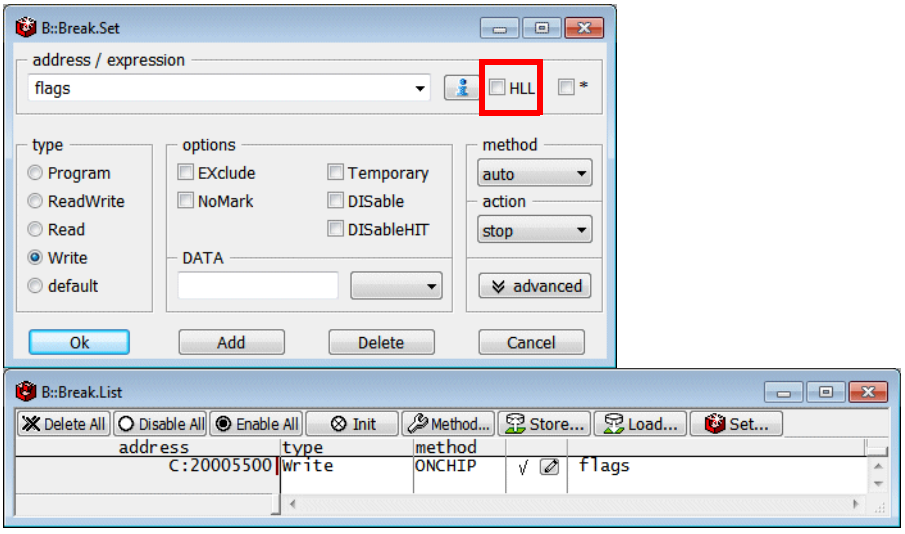

```
Break.Set func11\7
```

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



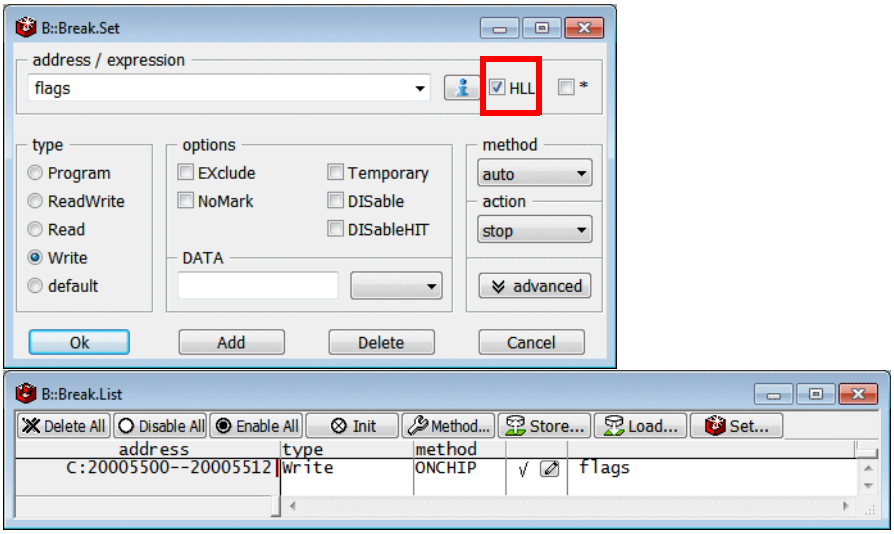

## Variable/HLL Check Box OFF

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



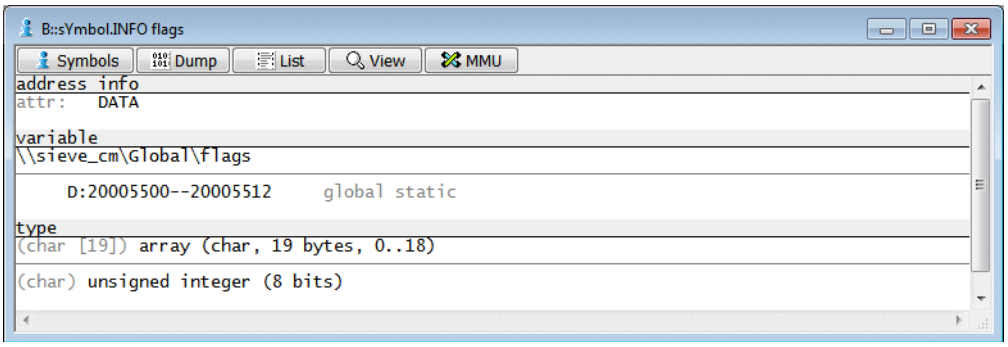

```
Break.Set flags
```

- 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
```

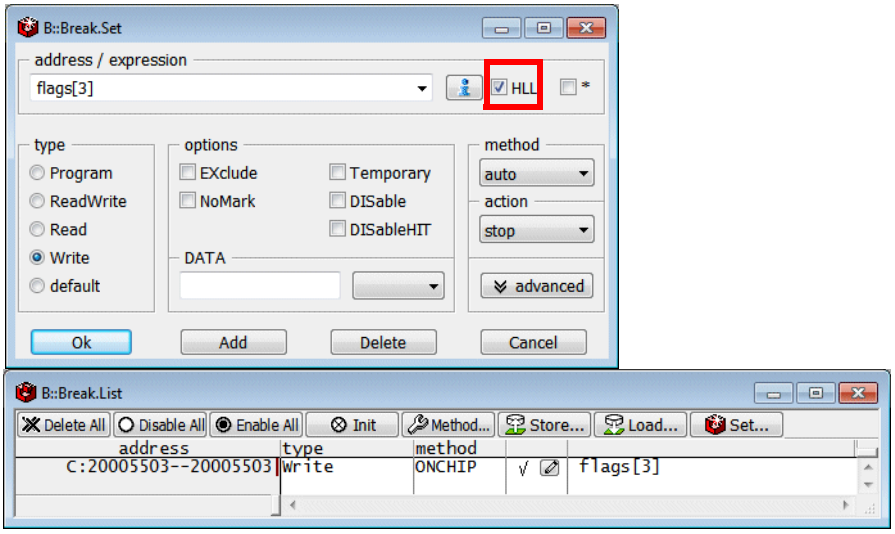
```
sYmbol.INFO flags                       ; display symbol information
                                        ; for variable flags
```




## Variable/HLL Check Box Must Be ON

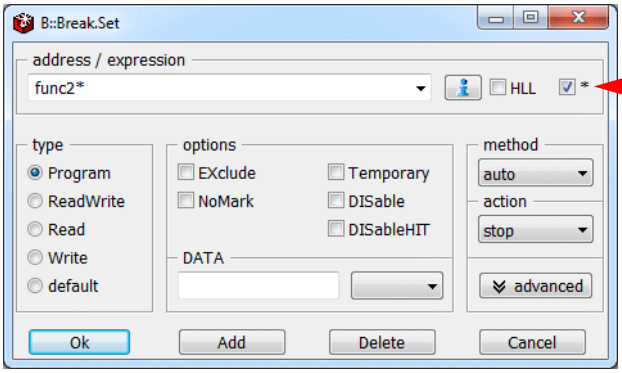
If you want to use an HLL expression to specify the address range for a Read/Write breakpoint, the HLL check box has to be checked.

- 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.

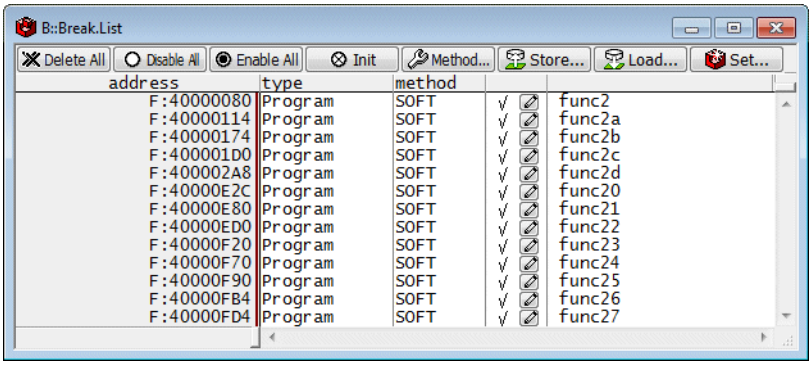


```
Var.Break.Set flags[3]
```

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

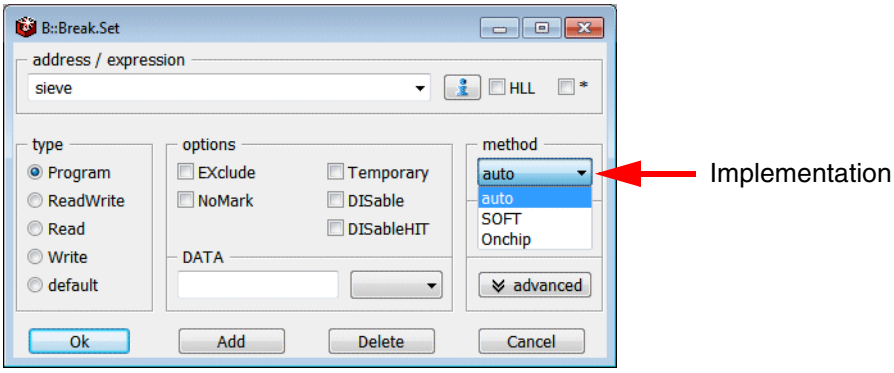



Check * to enable wildcard usage

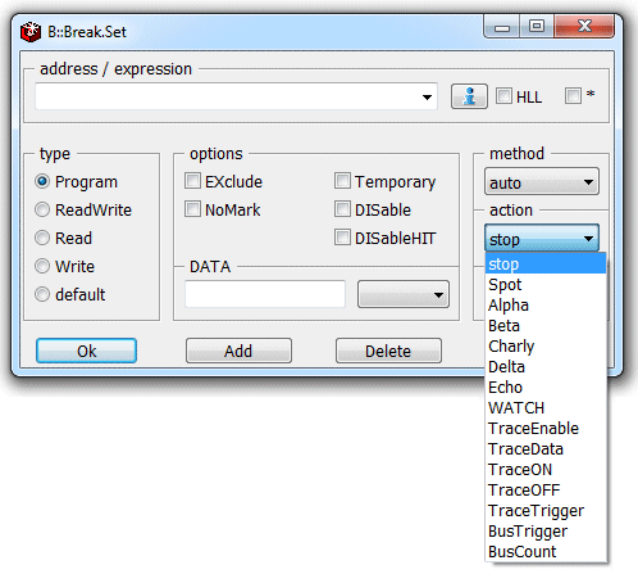



Requires sufficient resources if Onchip breakpoints are used.

```
Break.SetPATtern func2*
```




| Implementation | |
|---|---|
| **auto** | Use breakpoint implementation as predefined in TRACE32 PowerView. |
| **SOFT** | Implement breakpoint as Software breakpoint. |
| **Onchip** | 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)** | |
|---|---|
| **Spot** | 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. |
| **Alpha** | Set an Alpha breakpoint. |
| **Beta** | Set a Beta breakpoint. |
| **Charly** | Set a Charly breakpoint. |
| **Delta** | Set a Delta breakpoint. |
| **Echo** | Set an Echo breakpoint. |
| **WATCH** | 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.

| Action (on-chip or off-chip trace) | |
|---|---|
| **TraceEnable** | Advise on-chip trace logic to generate trace information on the specified event. |
| **TraceON** | Advise on-chip trace logic to start with the generation of trace information at the specified event. |
| **TraceOFF** | Advise on-chip trace logic to stop with the generation of trace information at the specified event. |
| **TraceTrigger** | 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).

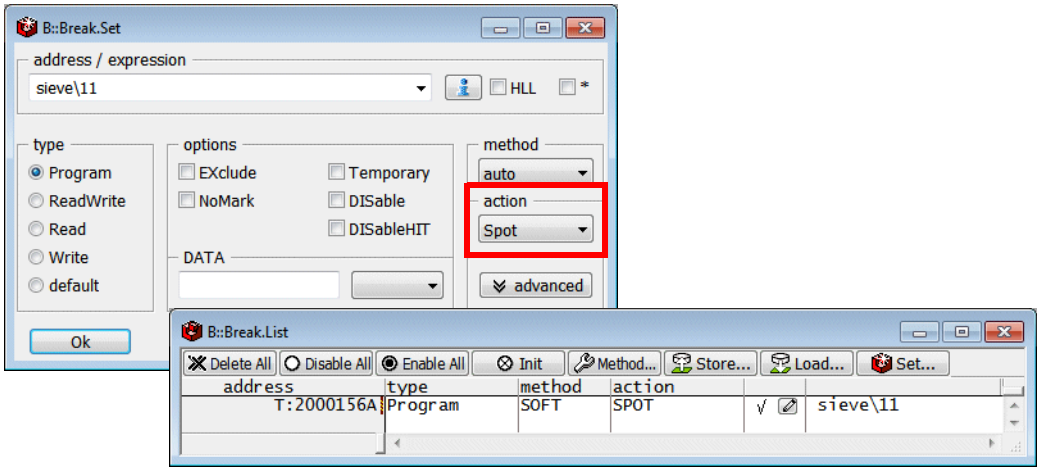
or with the description of the **Break.Set** command.

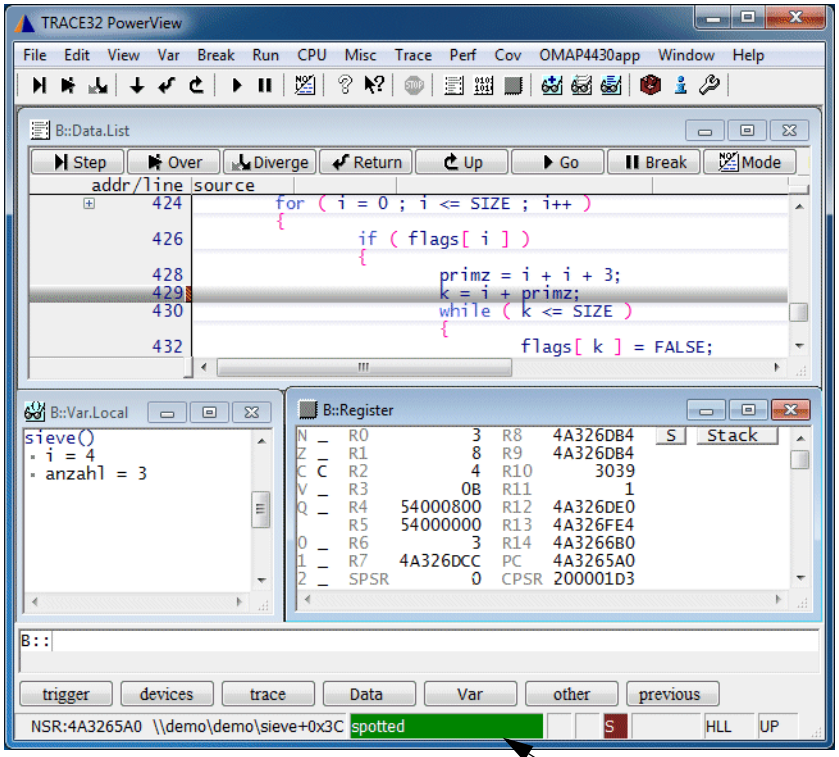
# Example for the Action Spot

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.

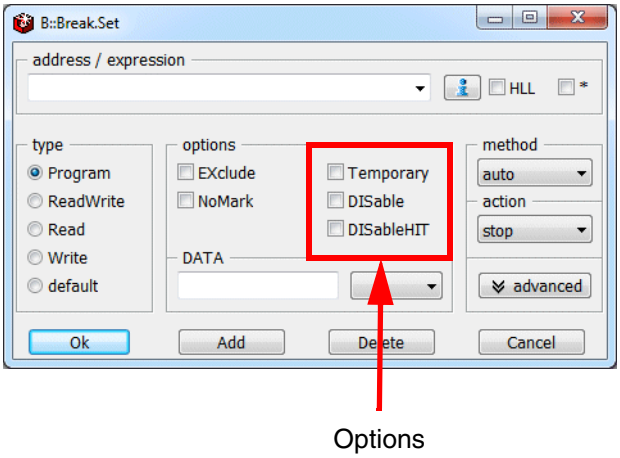
**Example:** Update the screen whenever the program executes the instruction sieve\11.






**spotted** indicates a breakpoint with the action Spot

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

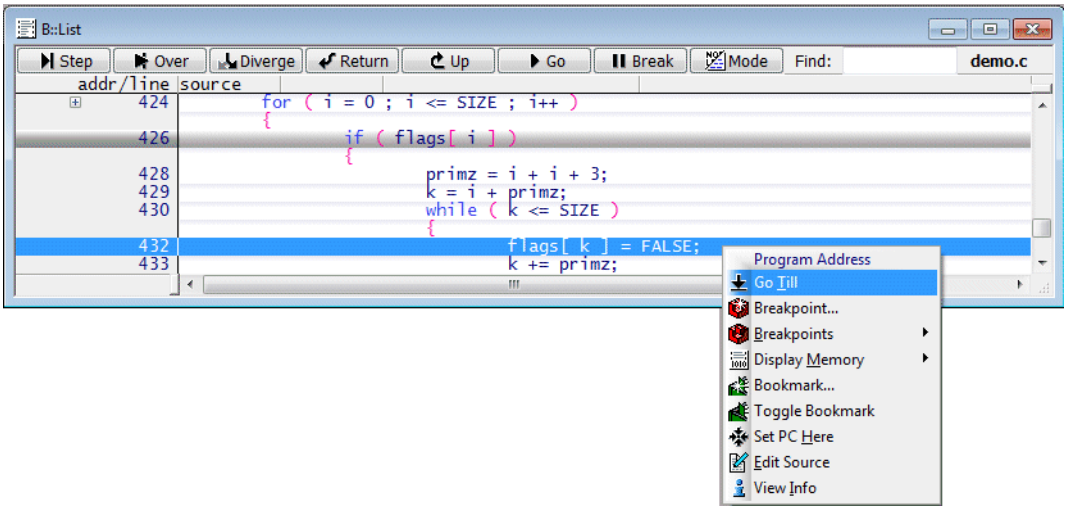



| | |
|---|---|
| **Temporary** | **OFF:** Set a permanent breakpoint (default).<br>**ON:** Set a temporary breakpoint. All temporary breakpoints are deleted the next time the core(s) stops the program execution. |
| **DISable** | **OFF:** Breakpoint is enabled (default).<br>**ON:** Set breakpoint, but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

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

**Examples:**

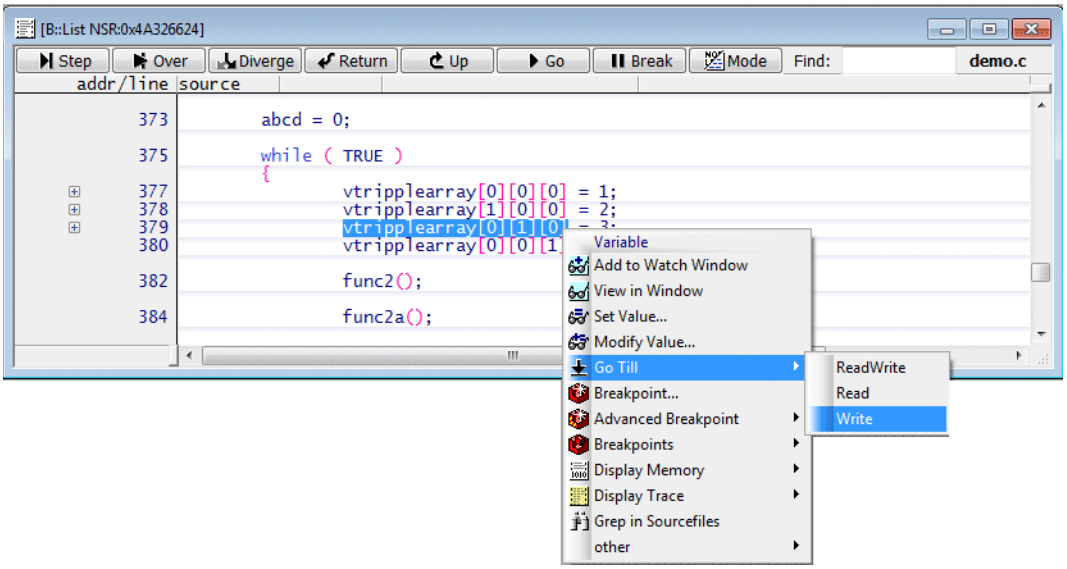
- **Go Till**




**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
```

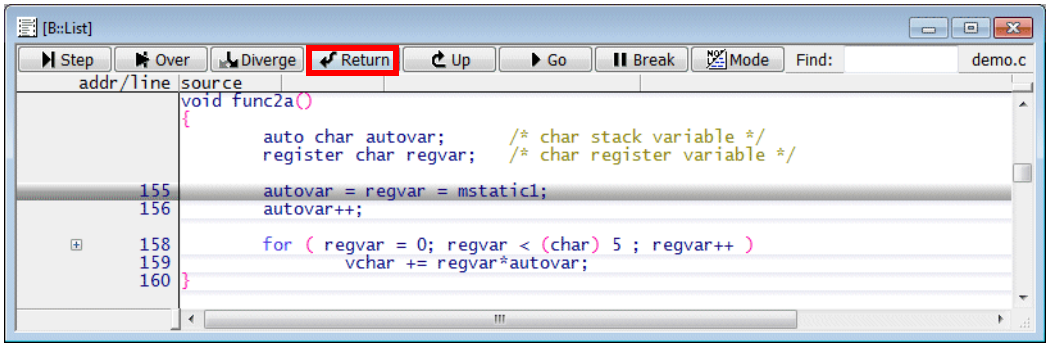
- **Go Till -> Write**




**Var.Go** *<hll_expression>* **[/Write]**

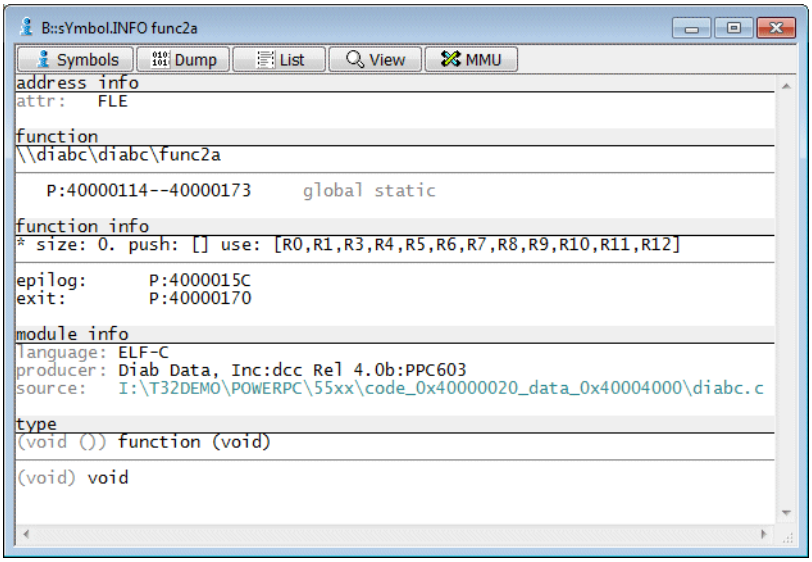
```
; set a temporary write breakpoint to the variable
; vtripplearray[0][1][0] and start the program execution
Var.Go vtripplearray[0][1][0] /Write
```

- **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
```



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.

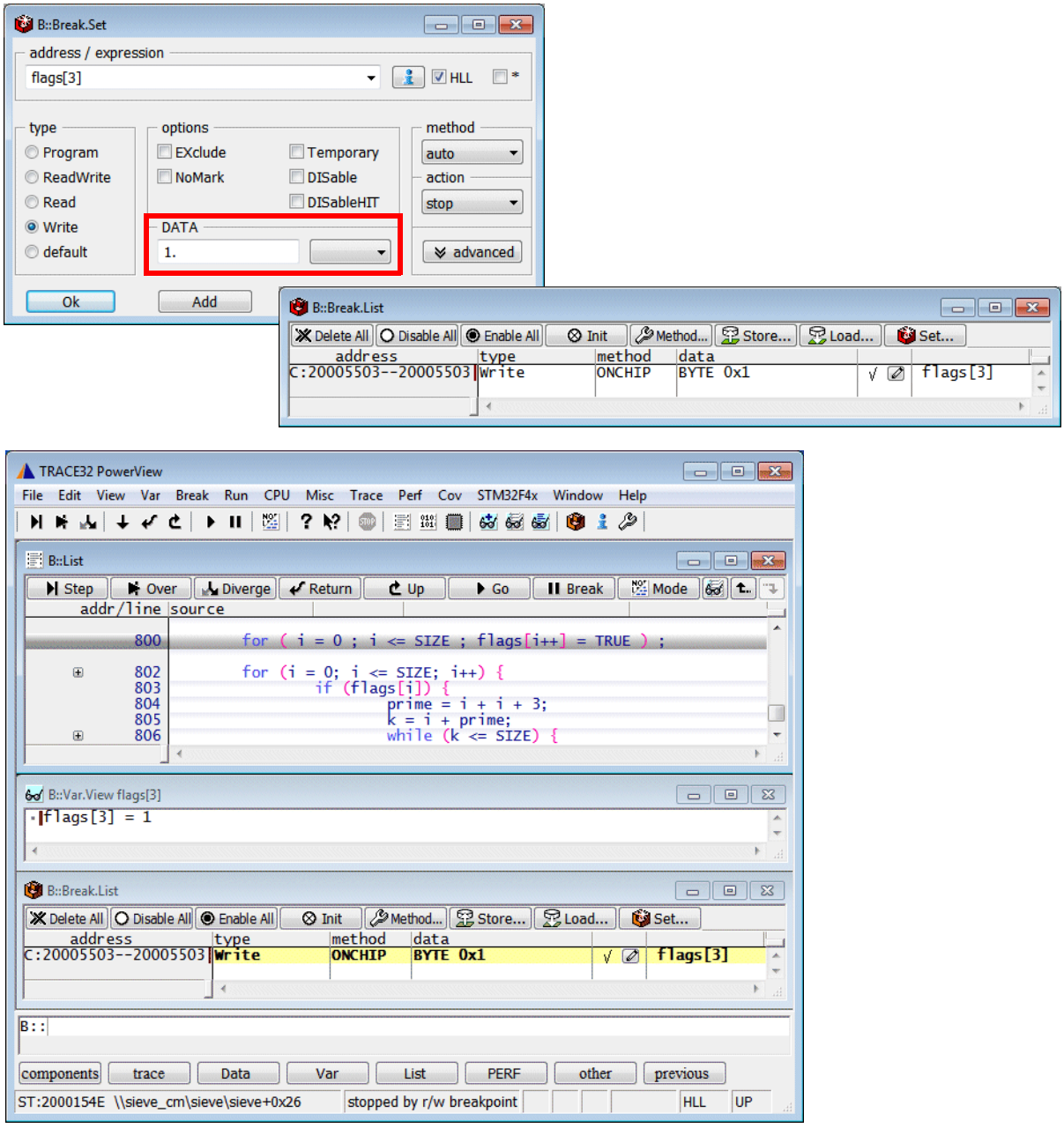
- DATA breakpoints are implemented as intrusive breakpoints if the core does not support Data Value Breakpoints. For details on the intrusive DATA breakpoints refer to the description of the **Break.Set** command.

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

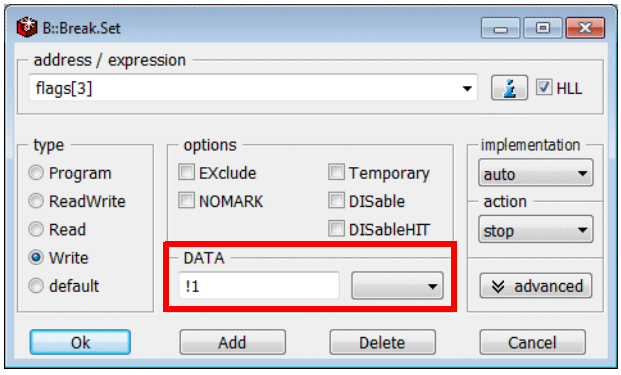
  TRACE32 PowerView allows inverted data values for intrusive DATA breakpoints.

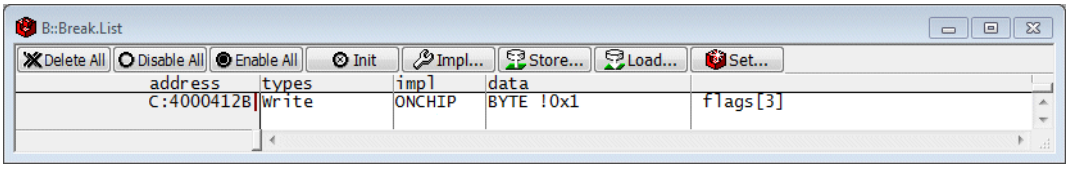
**Example:** Stop the program execution if a 1 is written to flags[3].

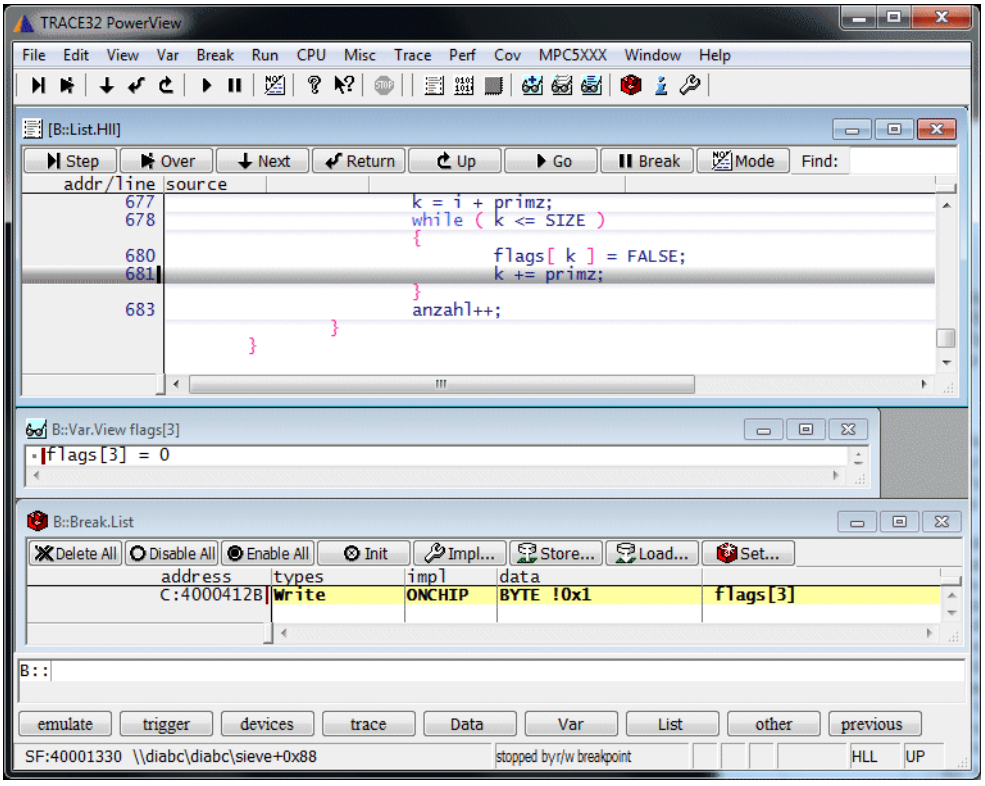



```
Var.Break.Set flags[3] /Write /DATA.auto 1.
```

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



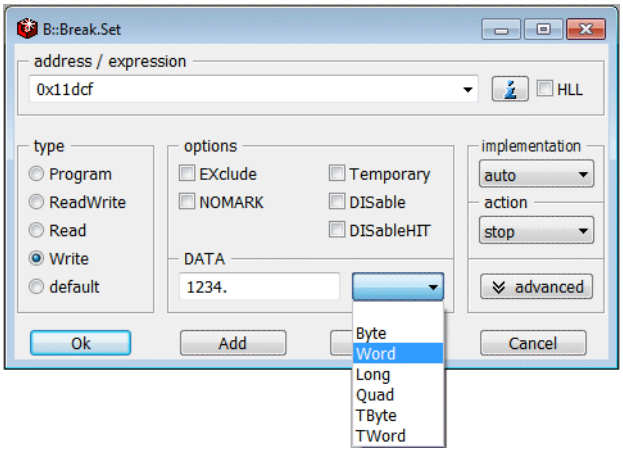




```
Var.Break.Set flags[3] /Write /DATA.auto !1.
```

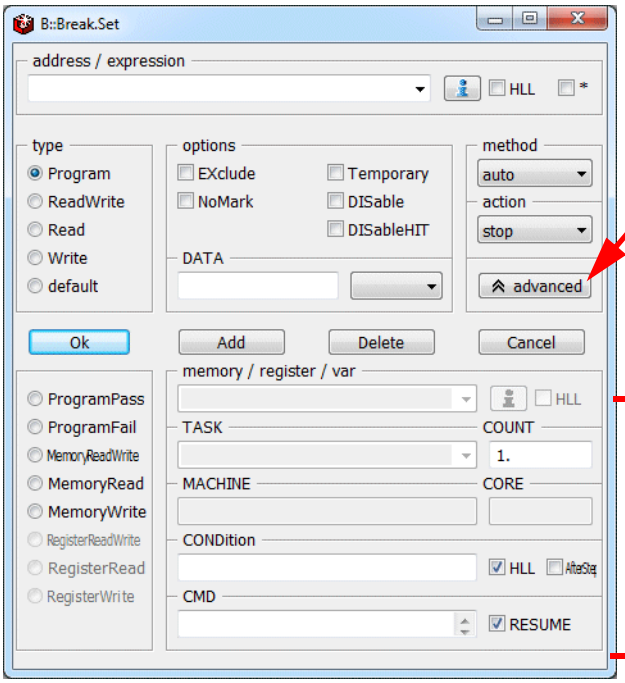
If an HLL expression is used TRACE32 PowerView gets the information if the data is written via a byte, word or long access from the symbol information.

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

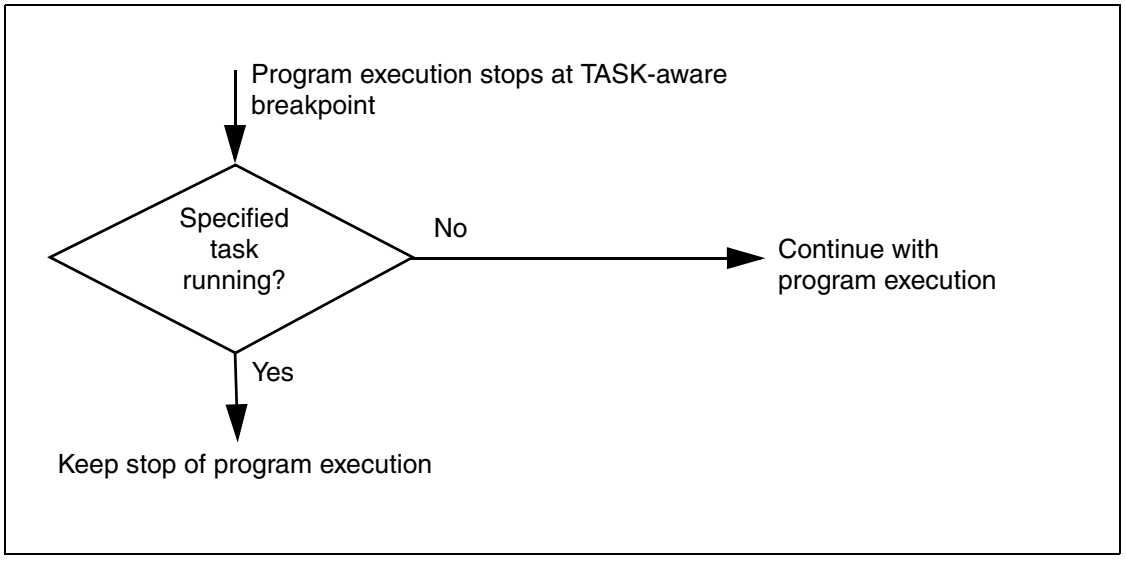
Advanced breakpoint input fields

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.

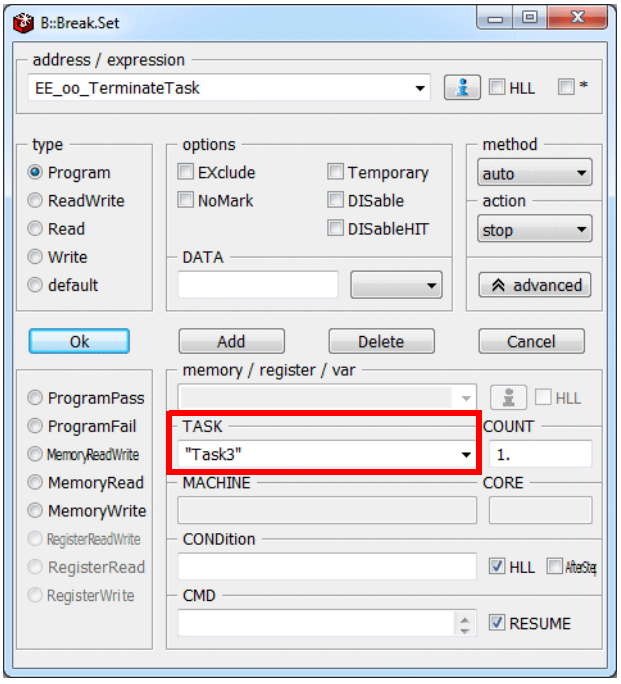
### Intrusive TASK-aware Breakpoint

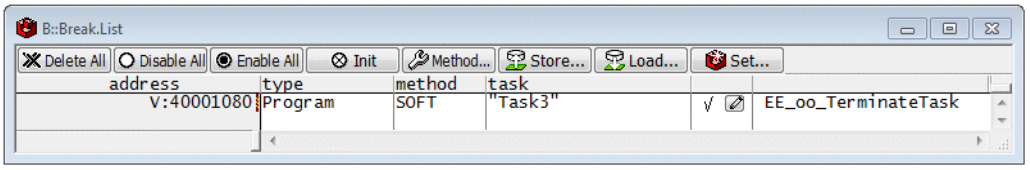
Processing:



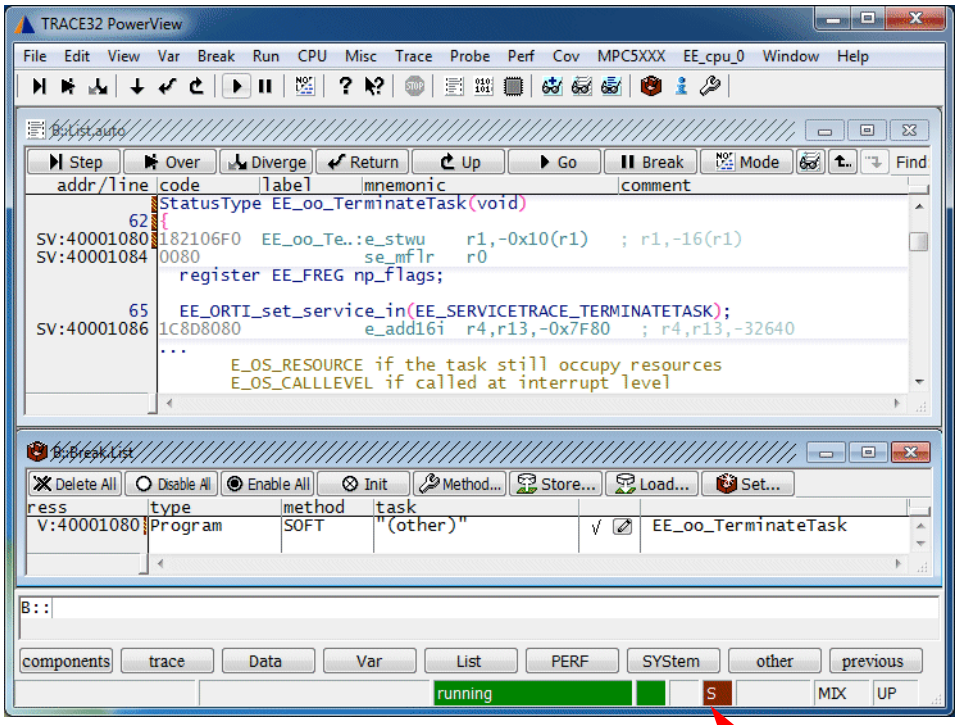

Each stop at the TASK-aware breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

**Example:** Stop the program execution at the entry to the function EE_oo_TerminateTask only if the task/process “Task3” is running.

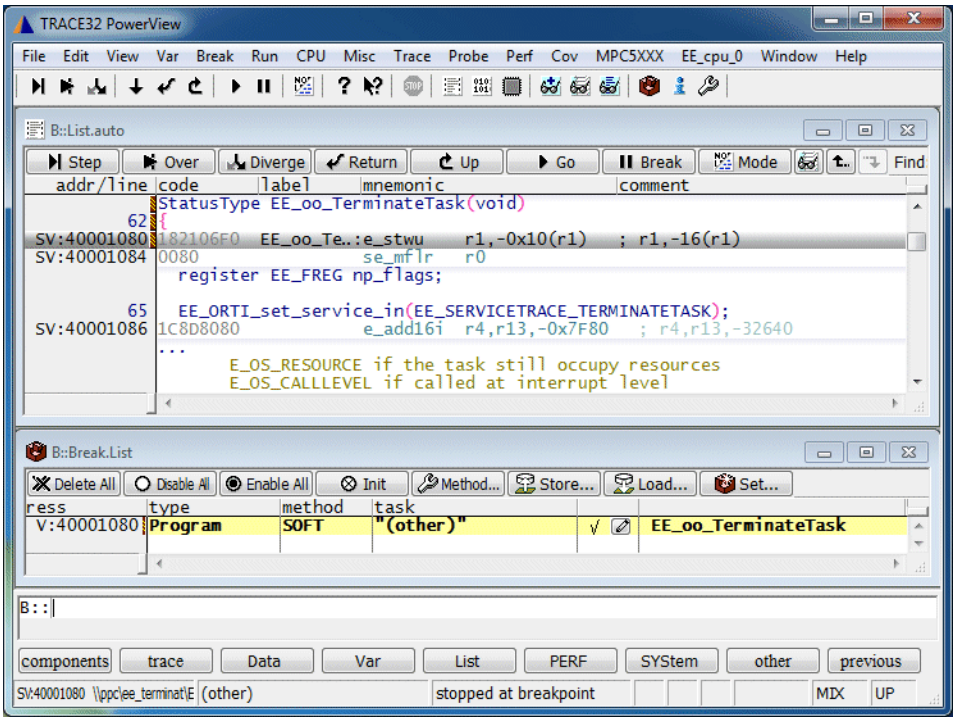


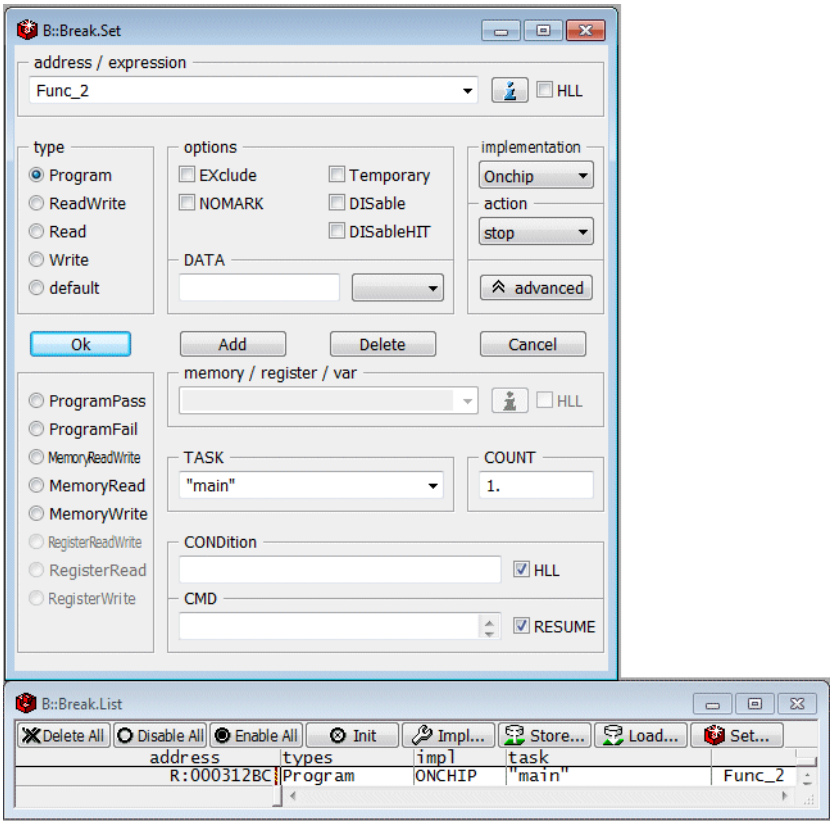

```
Break.Set  EE_oo_TerminateTask /Program /TASK "Task3"
```

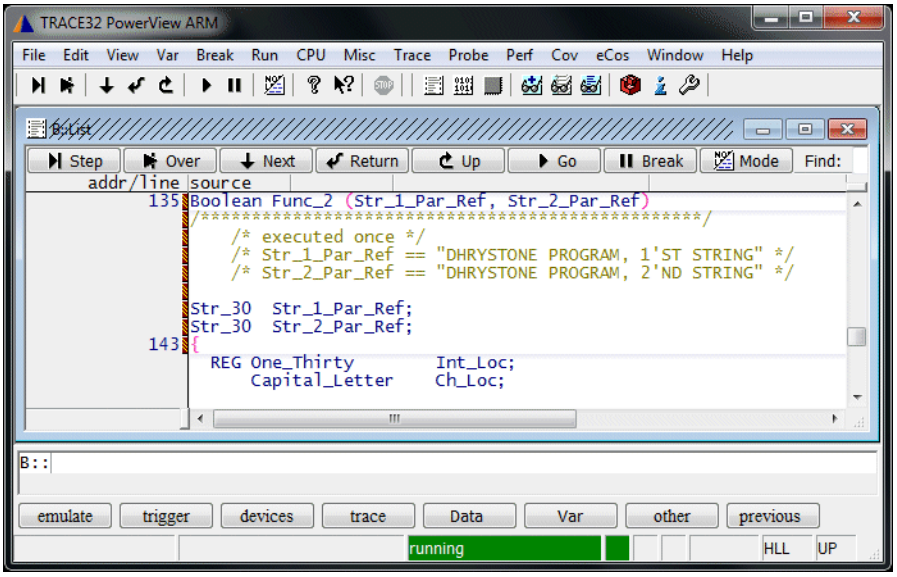



The red S indicates that an intrusive breakpoint is used



Example for ARM9: Stop the program execution at the entry to the function Func_2 only if the taskF “main” is running (Onchip breakpoint).



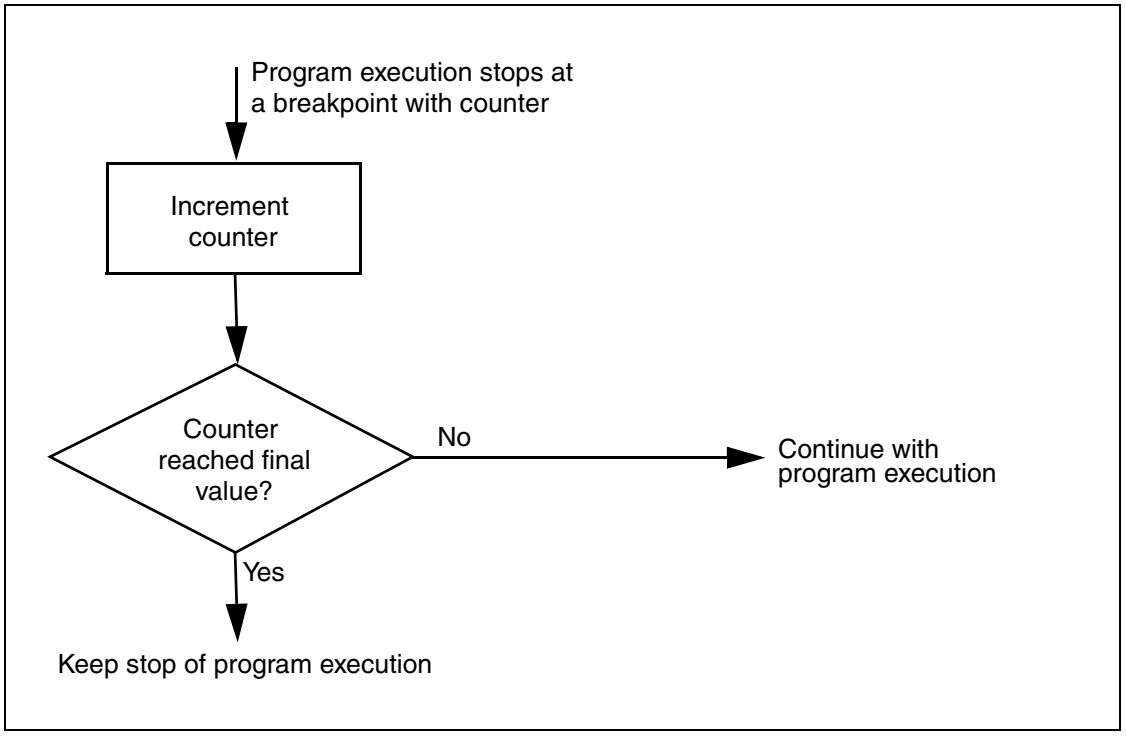



Counters allow to stop the program execution on the ***n th*** hit of a breakpoint.

## Software Counter

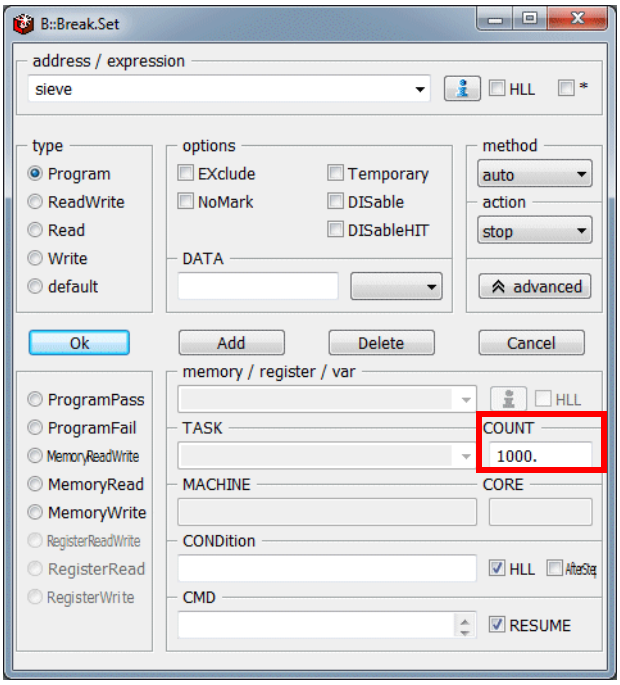
If the on-chip break logic of the core does not provide counters or if a Software breakpoint is used, counters are implemented as software counters.

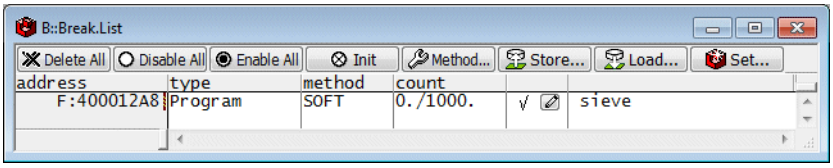
Processing:



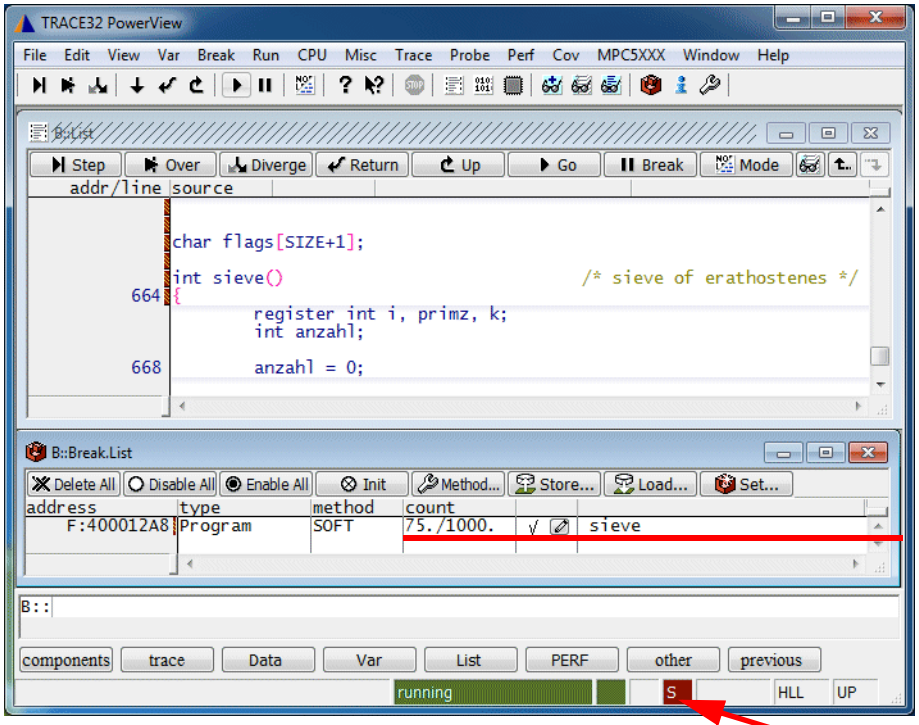

Each stop at a Counter breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

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



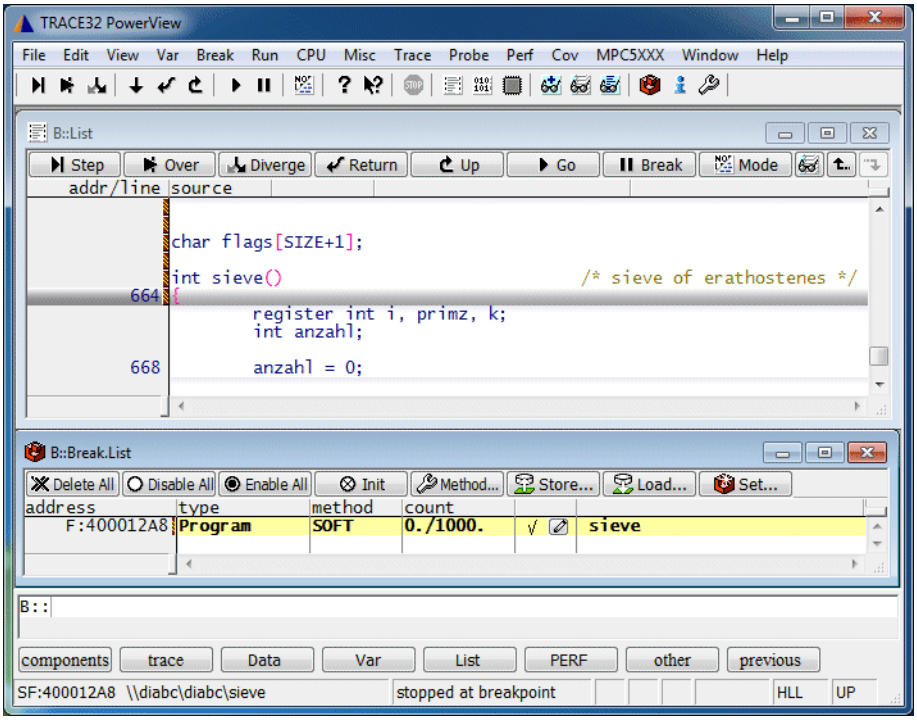




```
Break.Set sieve /COUNT 1000.
```



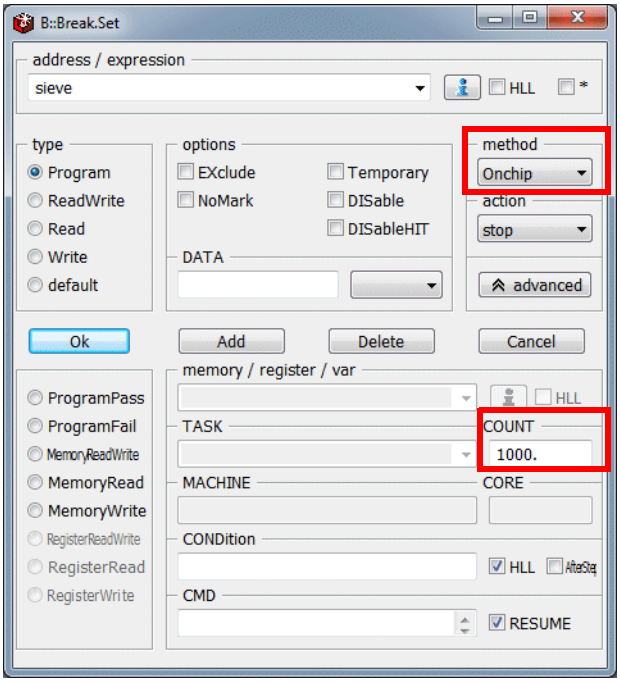

The current counter value is displayed in the **Break.List** window

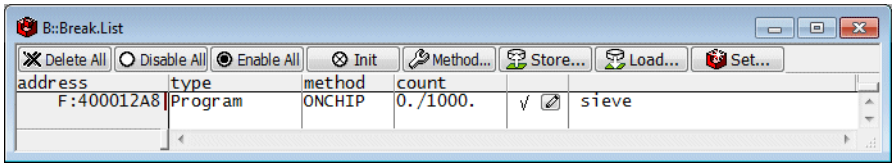
The red S indicates an intrusive breakpoint



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.



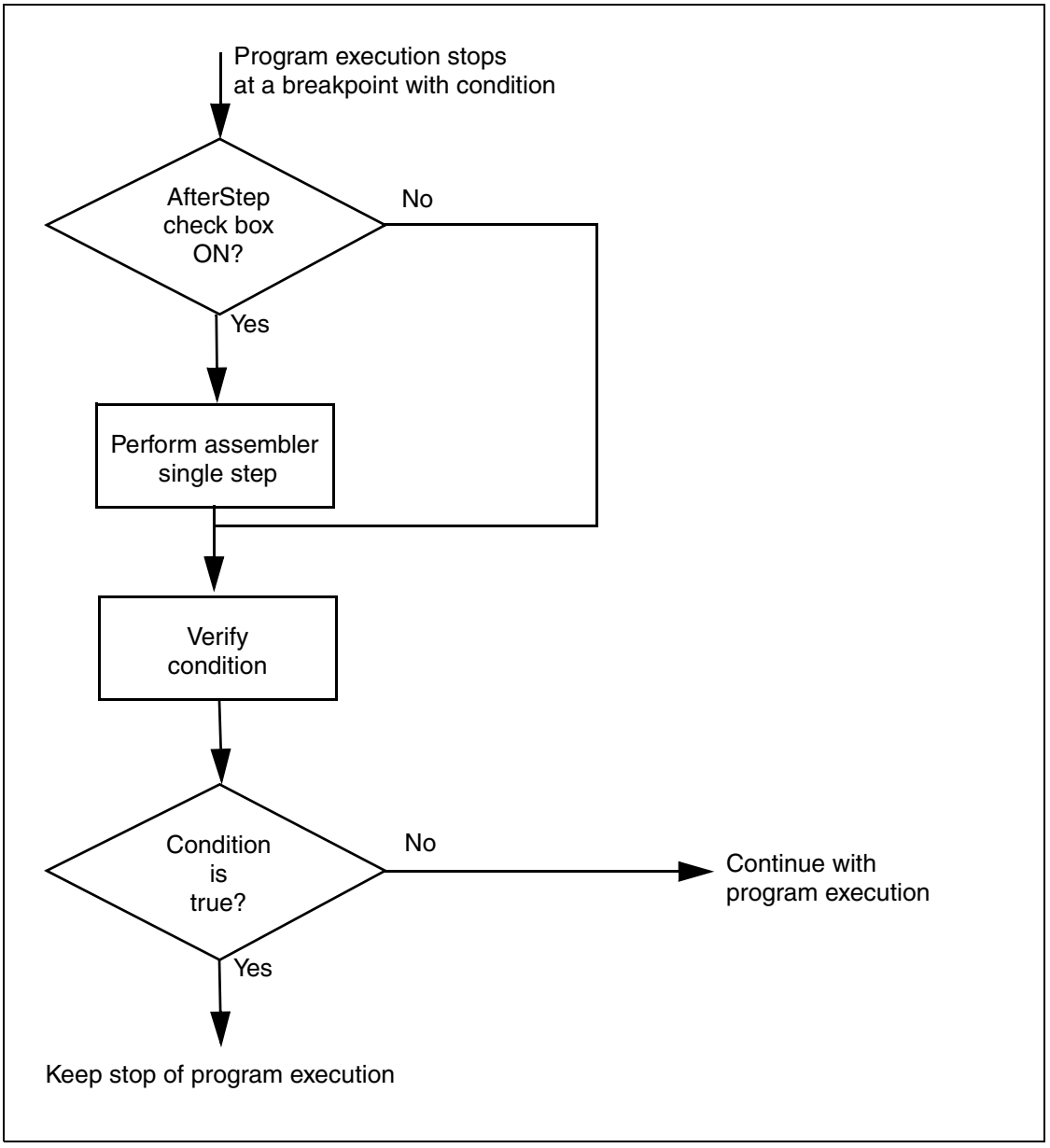

```
Break.Set sieve /COUNT 1000. /Onchip
```

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.

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

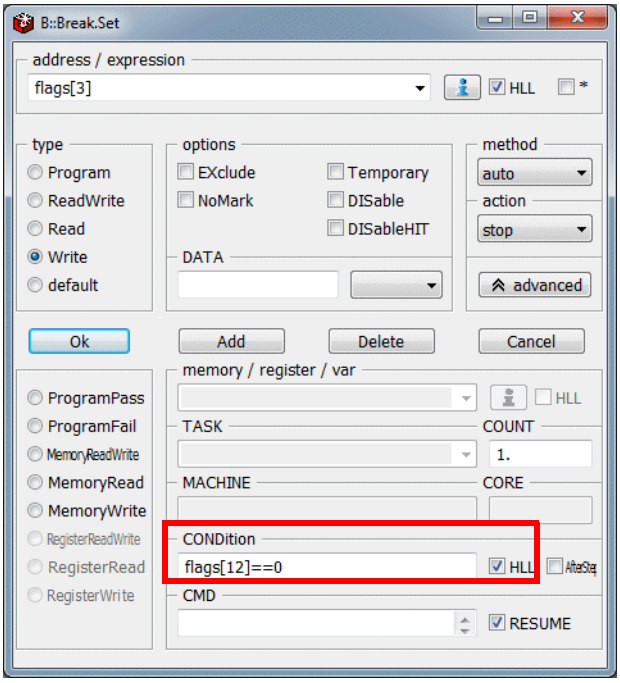
CONDition breakpoints are always intrusive.

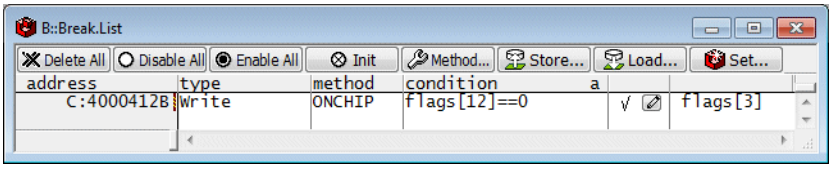
Processing:

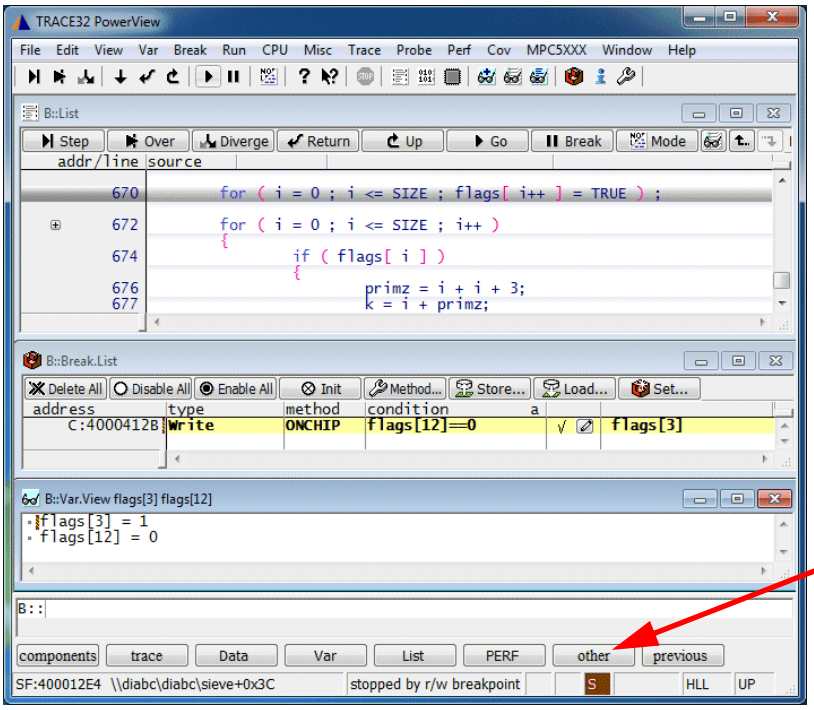



Each stop at a CONDition breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

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





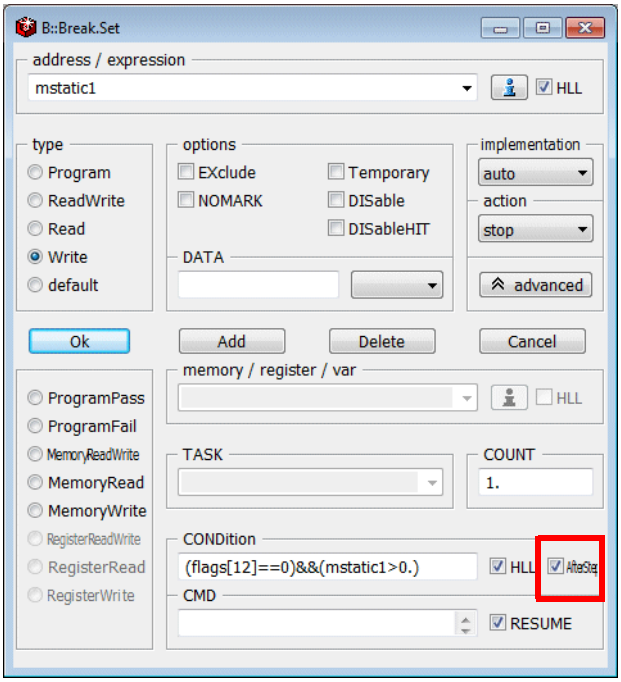

The red S indicates an intrusive breakpoint

```
Var.Break.Set flags[3] /Write /VarCONDition flags[12]==0
```

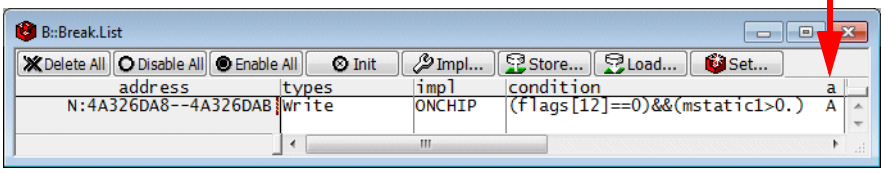
**Example: “Break-before-make” Read/Write breakpoints only**

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.

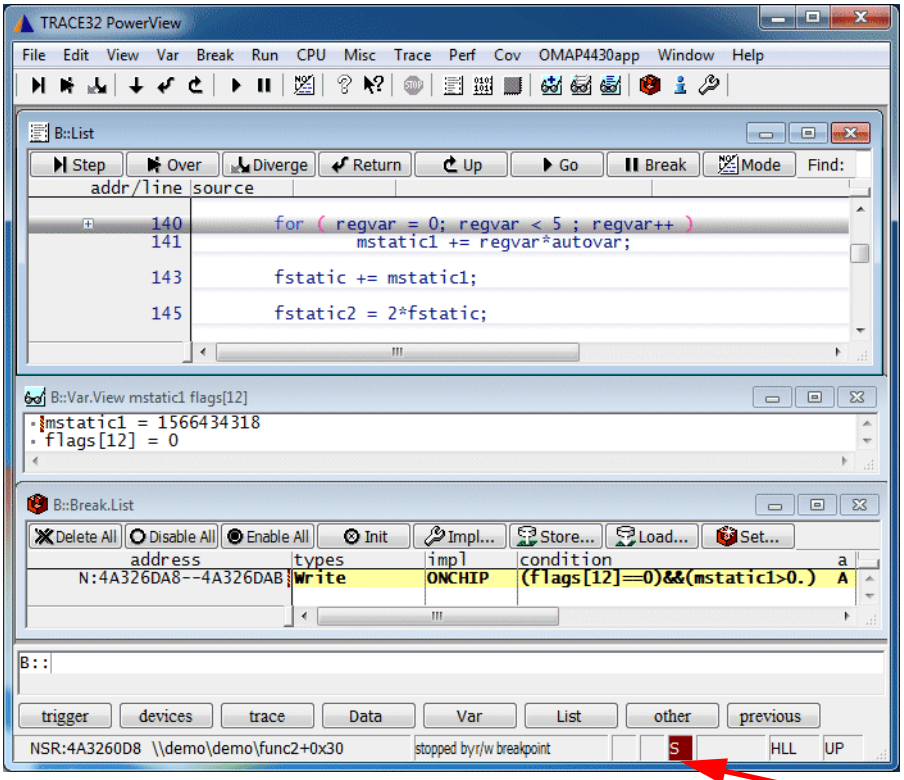
Perform an assembler single step because the processor architecture stops before the write access is performed.




AfterStep checked



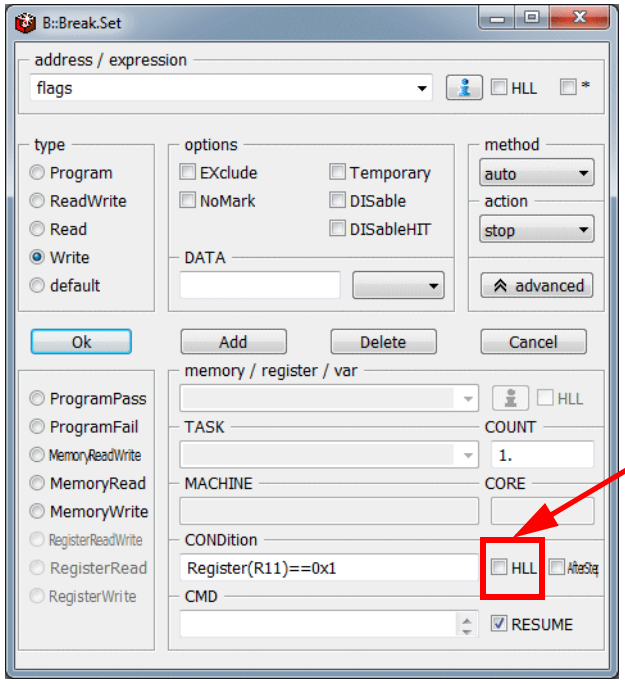

```
Var.Break.Set mstatic1 /Write /VarCONDition (flags[12]==0)&&(mstatic1>0)
/AfterStep
```




The red S indicates an intrusive breakpoint

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

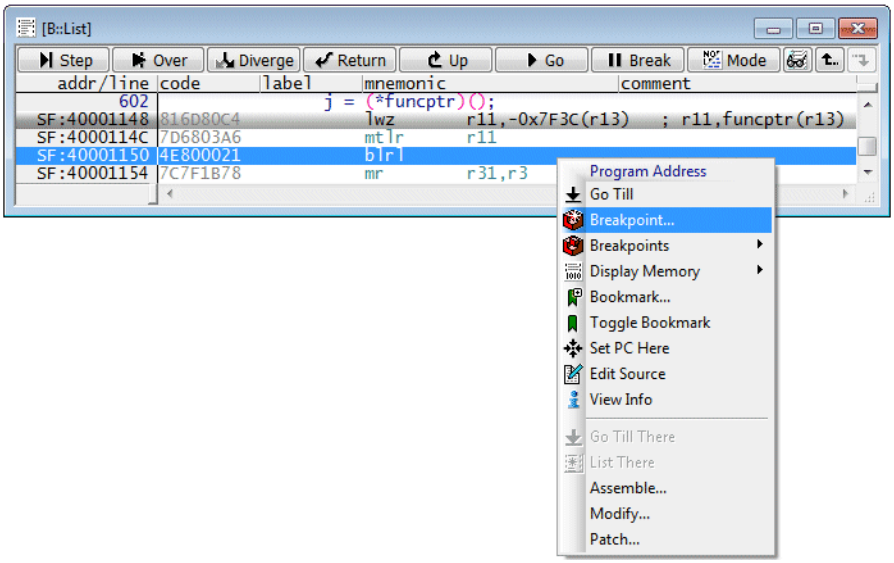
**Examples:** Stop the program executions on a write to the address flags if Register R11 is equal to 1.

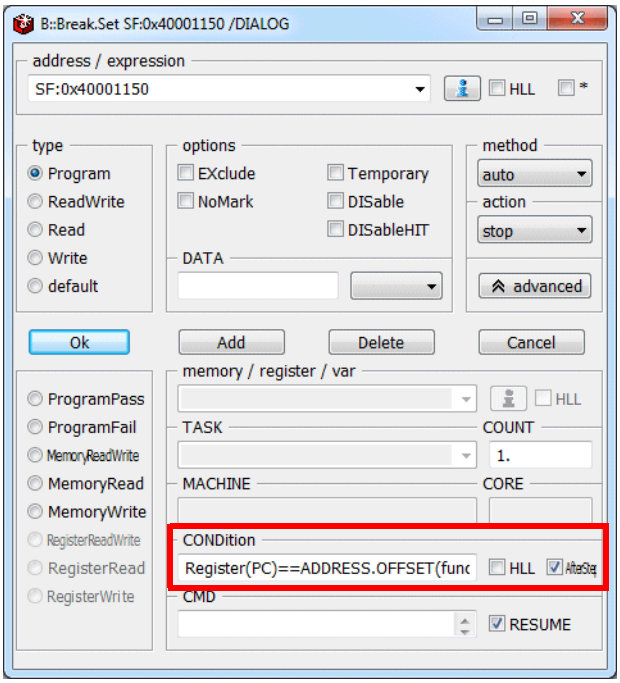


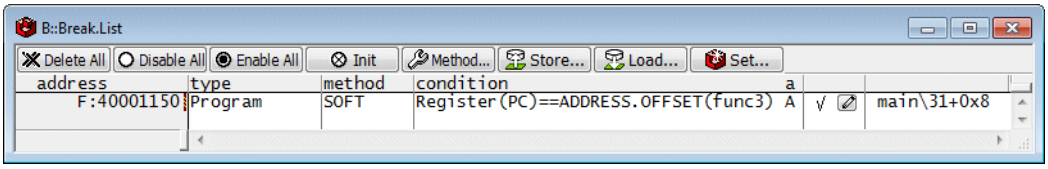

Switch HLL OFF ->
TRACE32 syntax can be used
to specify the condition

```
; 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
```

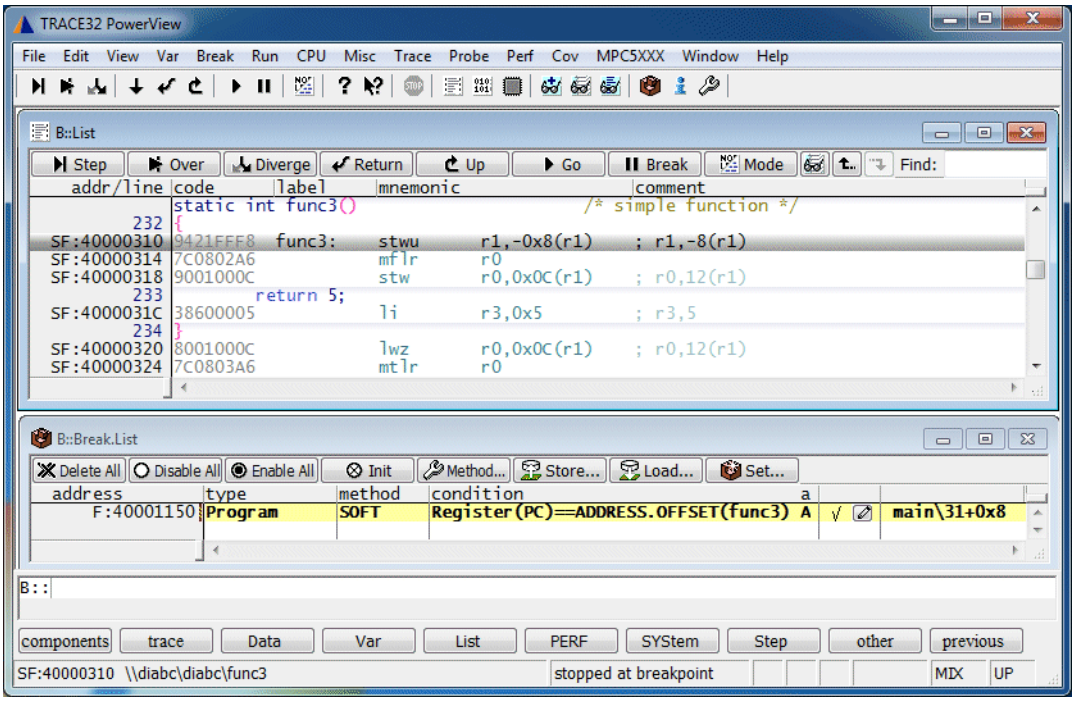
**Example:** Stop the program execution if an register-indirect call calls the function func3.



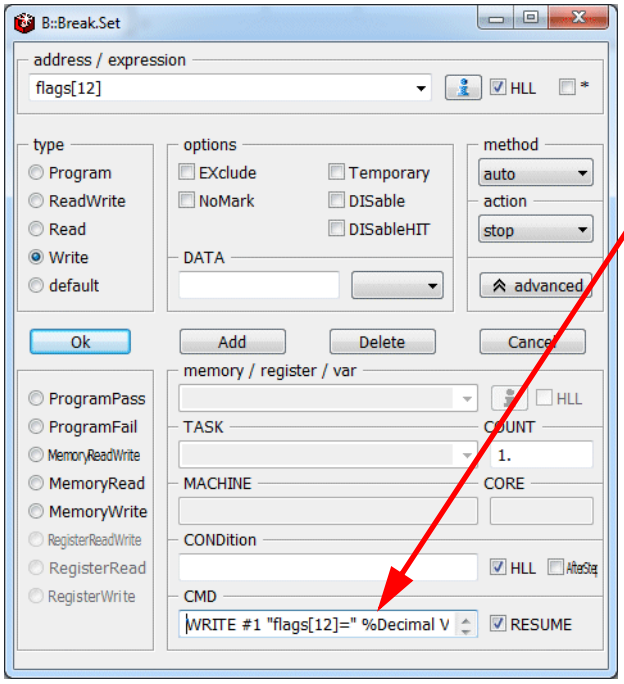

```
Break.Set main\31+0x8 /CONDition Register(PC)==ADDRESS.OFFSET(func3)
/AfterStep
```


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode Find:
addr/line code label mnemonic comment
static int func3() /* simple function */
232 {
SF:40000310 9421FFF8 func3: stwu r1,-0x8(r1) ; r1,-8(r1)
SF:40000314 7C0802A6 mflr r0
SF:40000318 9001000C stw r0,0x0C(r1) ; r0,12(r1)
233 return 5;
SF:4000031C 38600005 li r3,0x5 ; r3,5
234 }
SF:40000320 8001000C lwz r0,0x0C(r1) ; r0,12(r1)
SF:40000324 7C0803A6 mtlr r0
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
address type method condition a
F:40001150 Program SOFT Register(PC)==ADDRESS.OFFSET(func3) A main\31+0x8
B::
components trace Data Var List PERF SYStem Step other previous
SF:40000310 \\diabc\diabc\func3 stopped at breakpoint MIX UP

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

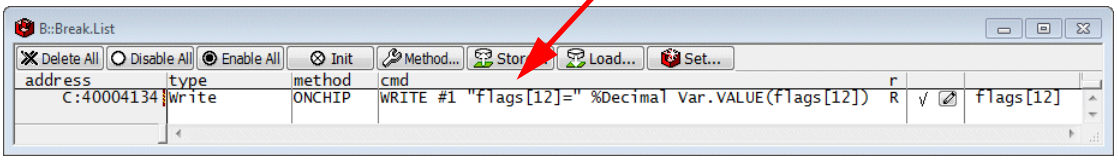
**Example:** Write the contents of flags[12] to a file whenever the write breakpoint at the variable flags[12] is hit.

```
OPEN #1 outflags.txt /Create          ; open the file for writing
```




The specified command(s) is executed whenever the breakpoint is hit. With RESUME ON the program execution will continue after the execution of the command(s) is finished.

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.

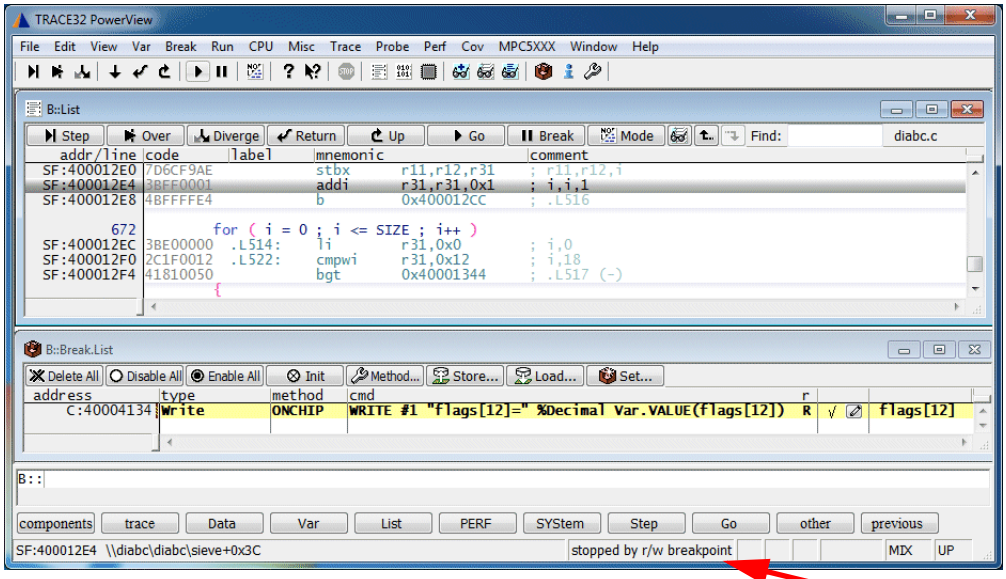



```
Var.Break.Set flags[12] /Write /CMD "WRITE #1 ""flags[12]="" %Decimal
Var.VALUE(flags[12])" /RESUME
```

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

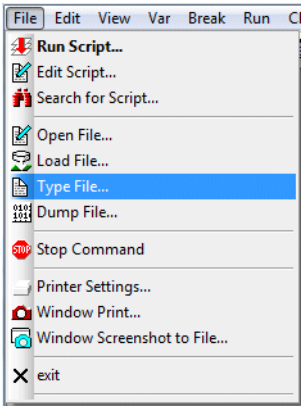
because the program execution is restarted before these commands are completed.

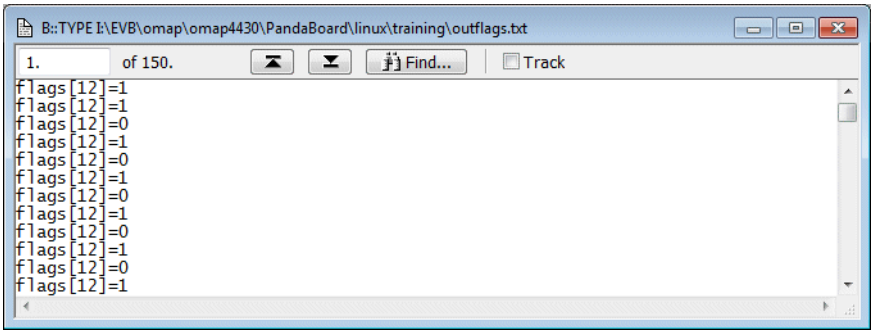



The state of the debugger toggles between **running** and **stopped**

```
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).

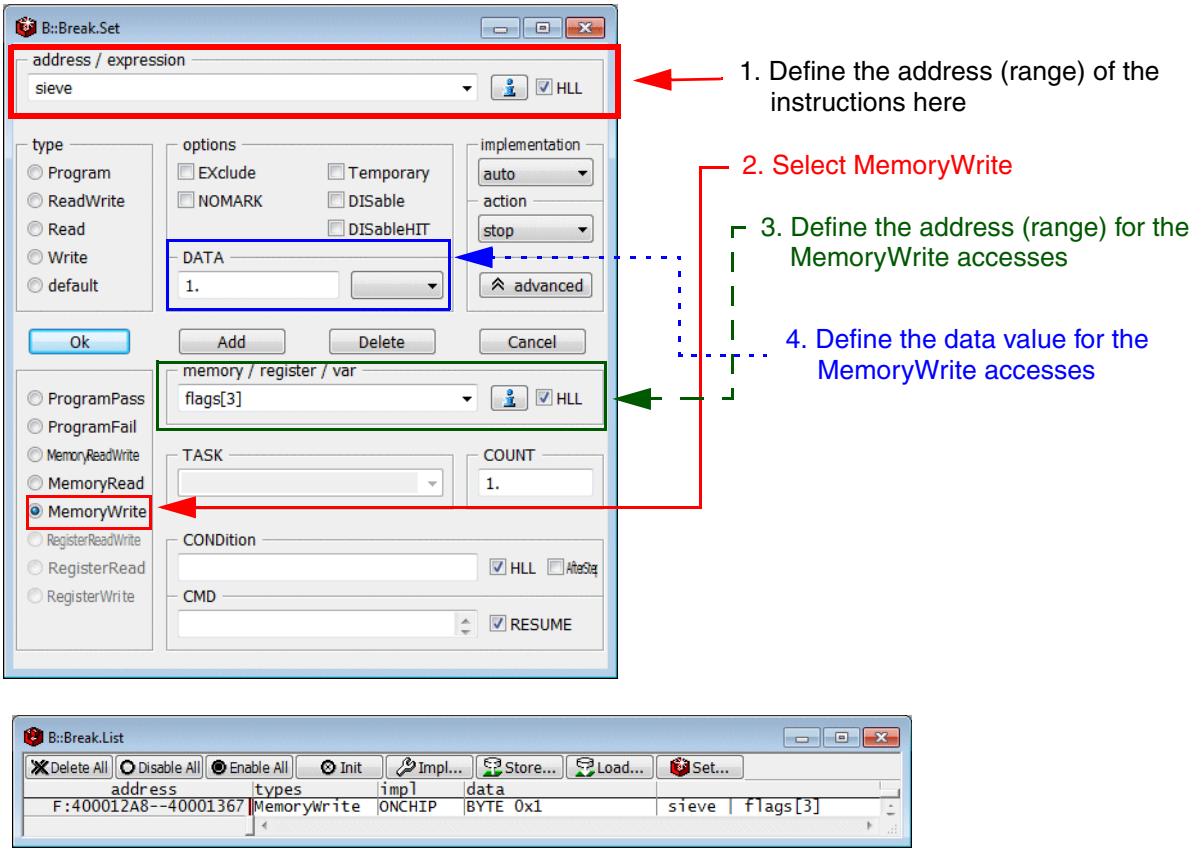
Preconditions

- 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.

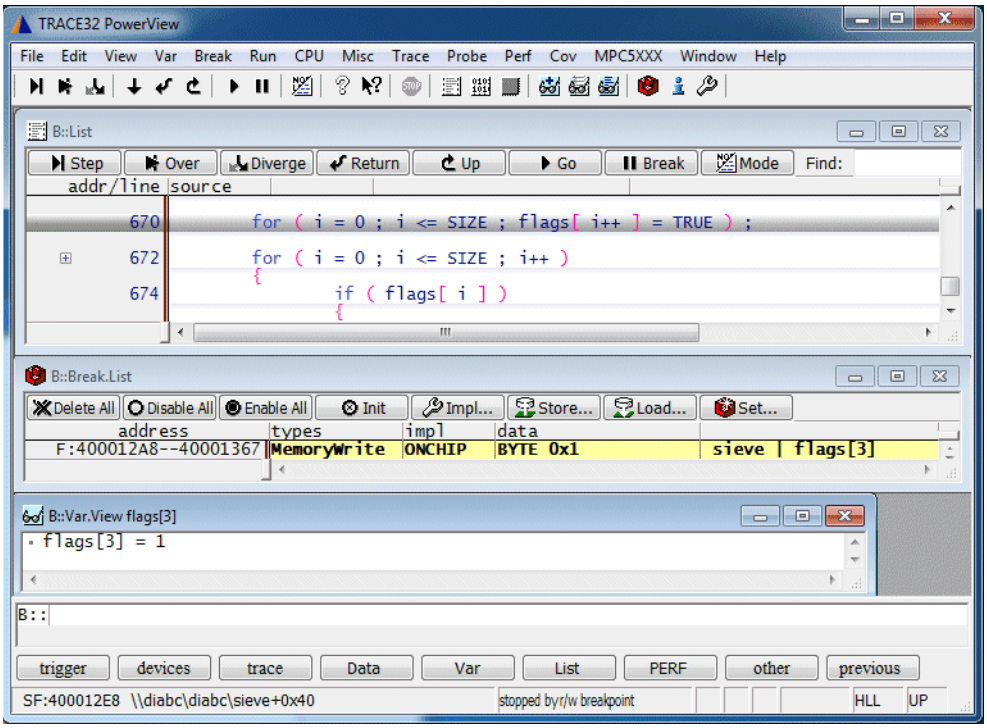
**Example:** Stop the program execution when the function sieve writes a 1 to variable flags[3]. (If your core does not support this feature, the **radio buttons** (MemoryWrite, MemoryRead etc.) are grey.)







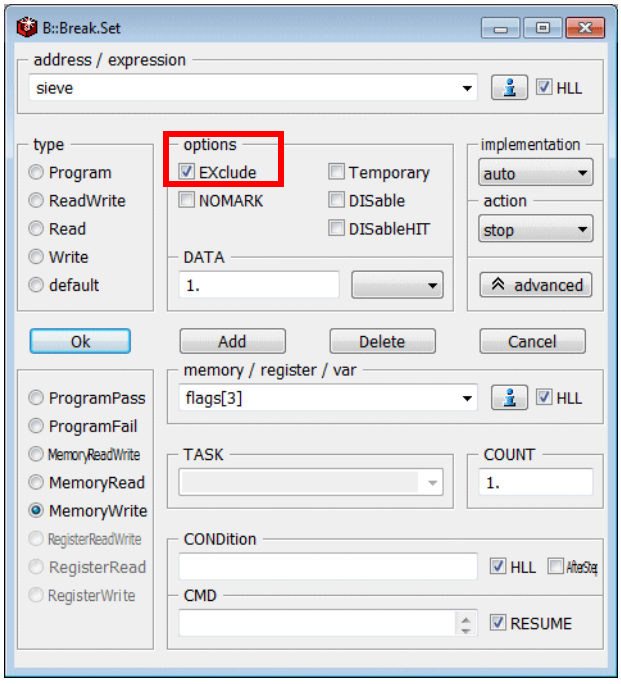
```
Var.Break.Set sieve /VarWrite flags[3] /DATA.auto 1.
```

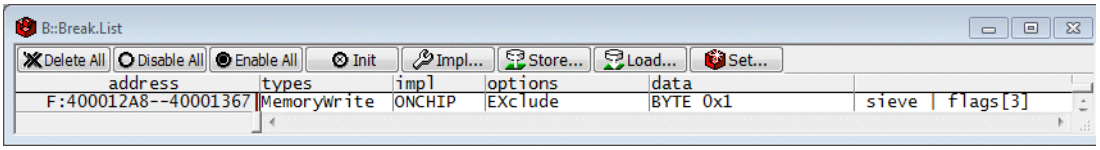



| | |
|---|---|
| **Exclude** | **(Advanced users only, not available on all cores)**<br><br>The breakpoint is inverted.<br><br>• by the inverting logic of the on-chip break logic<br><br>• by setting the specified breakpoint type to the following 2 address ranges<br>`0x0--(start_of_breakpoint_range-1)`<br>`(end_of_breakpoint_range+1)--end_of_memory`<br><br>The EXclude option applies only to Onchip breakpoints.<br><br>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. |

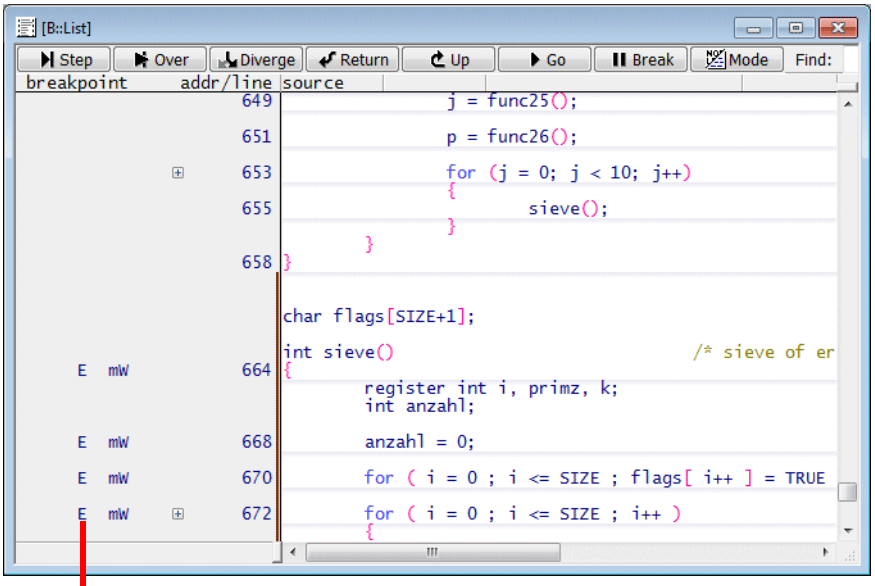
## Example for the Option EXclude

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




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

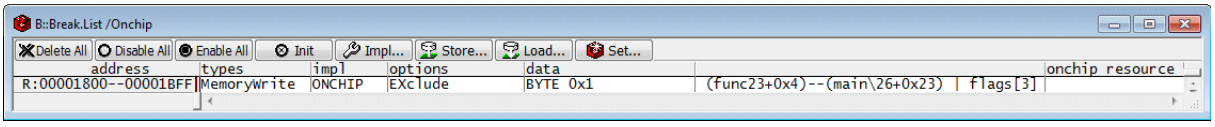



The function sieve is marked with **Exclude memoryWrite** breakpoints

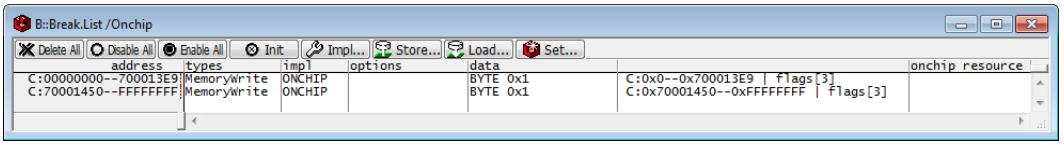
The following command allows to check how the option EXclude is implemented.

```
Break.List /Onchip
```

Inverting logic of on-chip break logic:

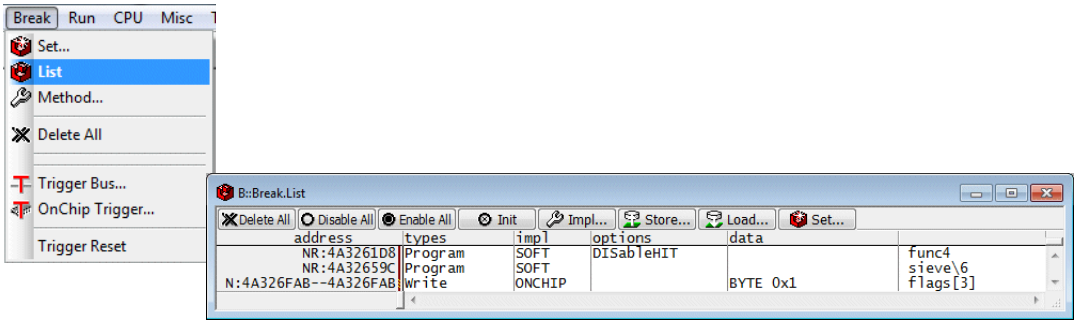



Two address range breakpoints:




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

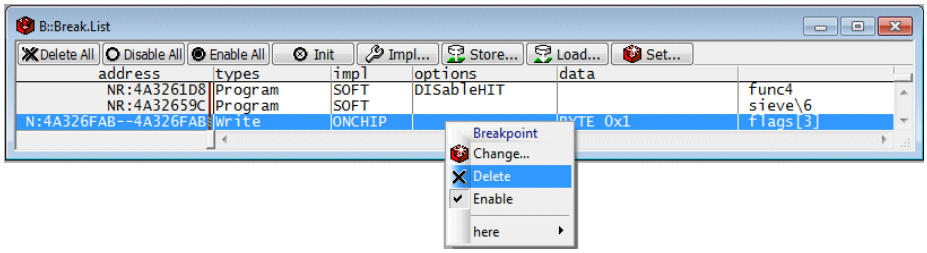
# Display a List of all Set Breakpoints




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

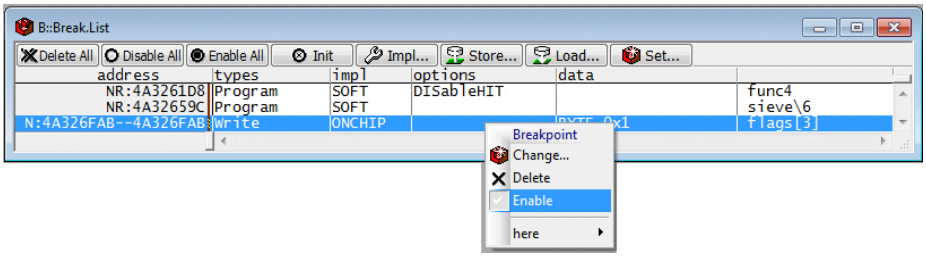
**Break.List** [*/<option>*] List all breakpoints

# Delete Breakpoints



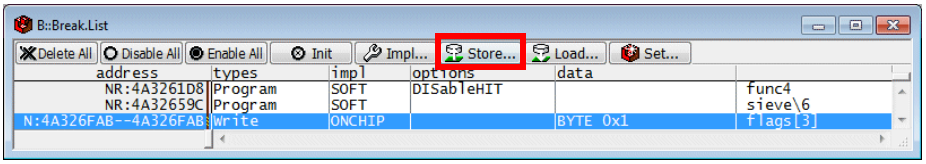

| | |
|---|---|
| **Break.Delete** *<address>\|<address_range>* [*/<type>*] [*/<implem.>*] [*/<option>*] | Delete breakpoint |
| **Var.Break.Delete** *<hll_expression>* [*/<type>*] [*/<implem.>*] [*/<option>*] | Delete HLL breakpoint |

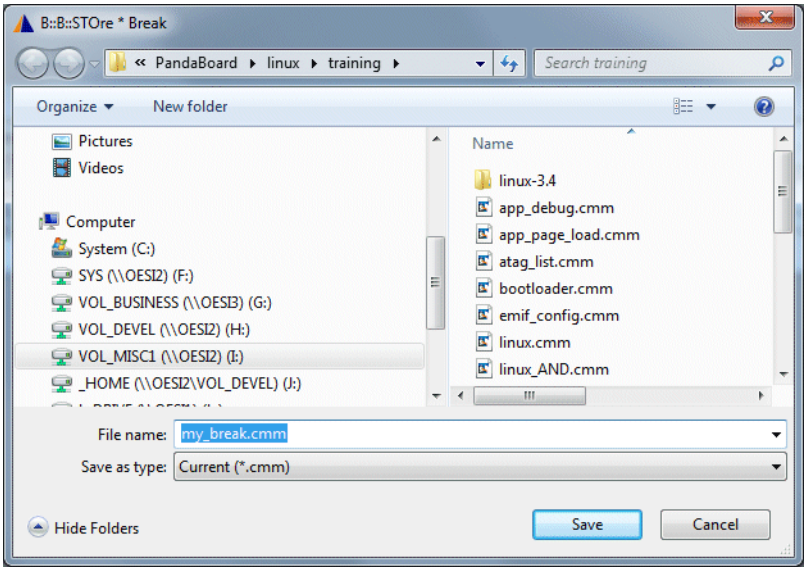
# Enable/Disable Breakpoints




| | |
|---|---|
| **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

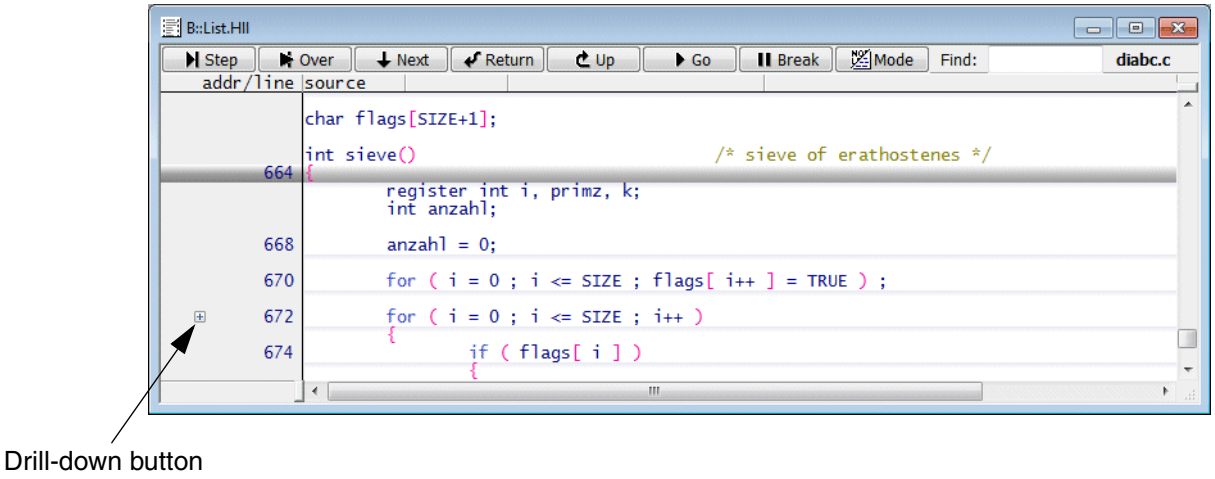
A video tutorial about debugging optimized code can be found here:

**https://www.lauterbach.com/tut_optimized.html**

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.

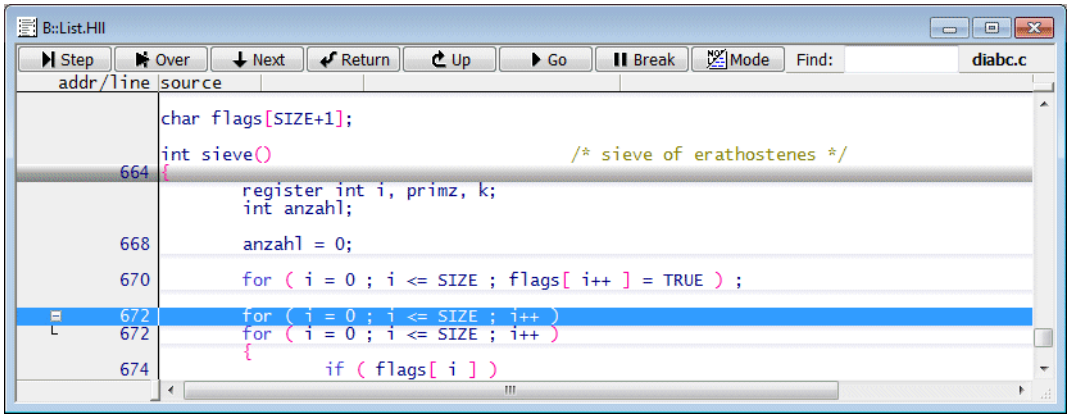


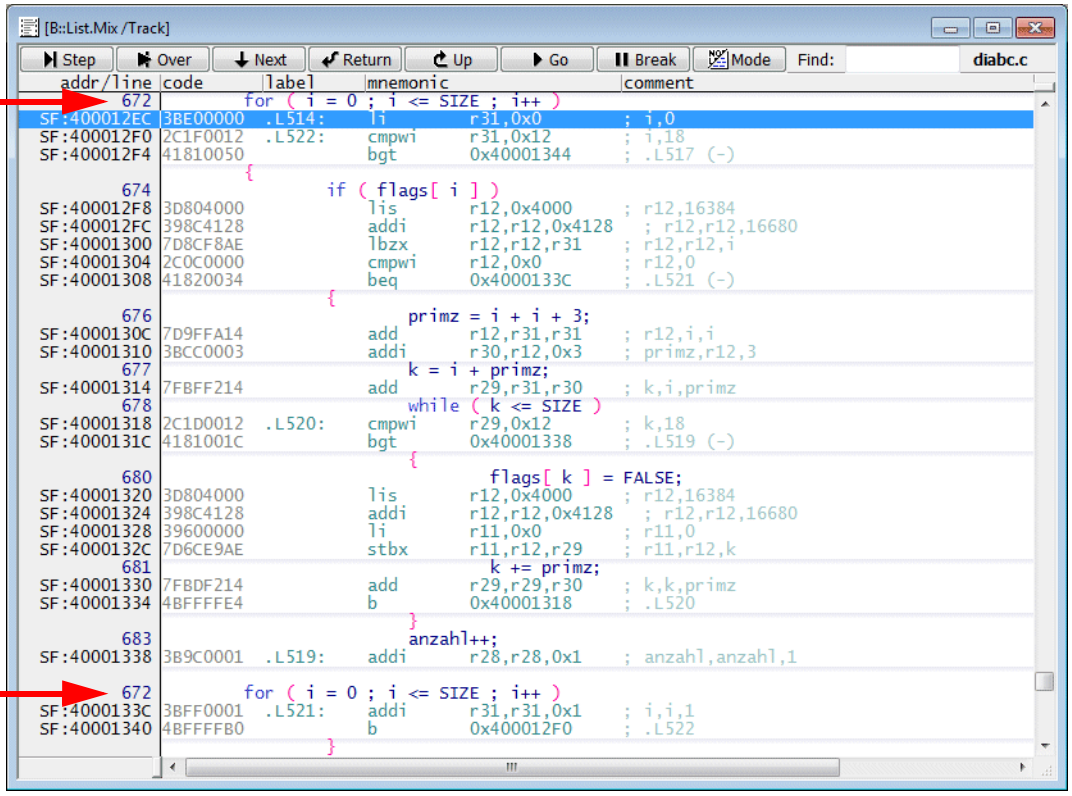

The following background information is fundamental if you want to debug optimized code:

- 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.<br><br>The blue cursor in the MIX mode display follows the cursor movement of the HLL mode display (Track option). |



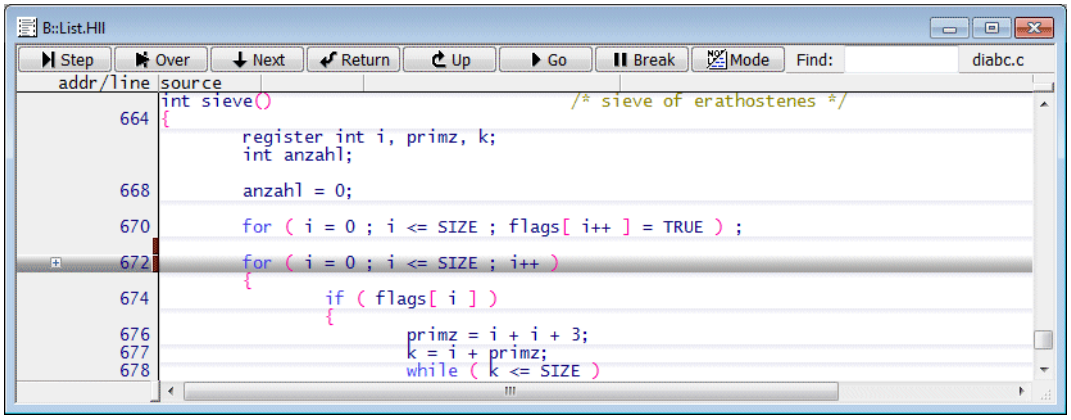



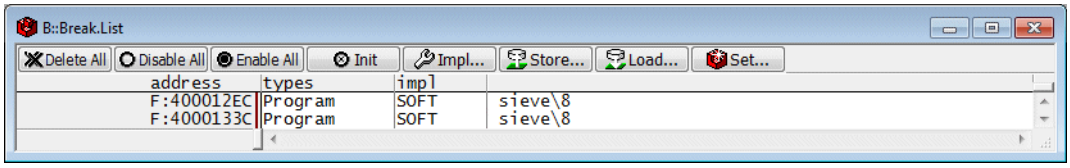
To keep track when debugging optimized code, it is recommended to work with an HLL mode and a MIX mode display of the source listing in parallel.

```
List.Hll

List.Mix
```

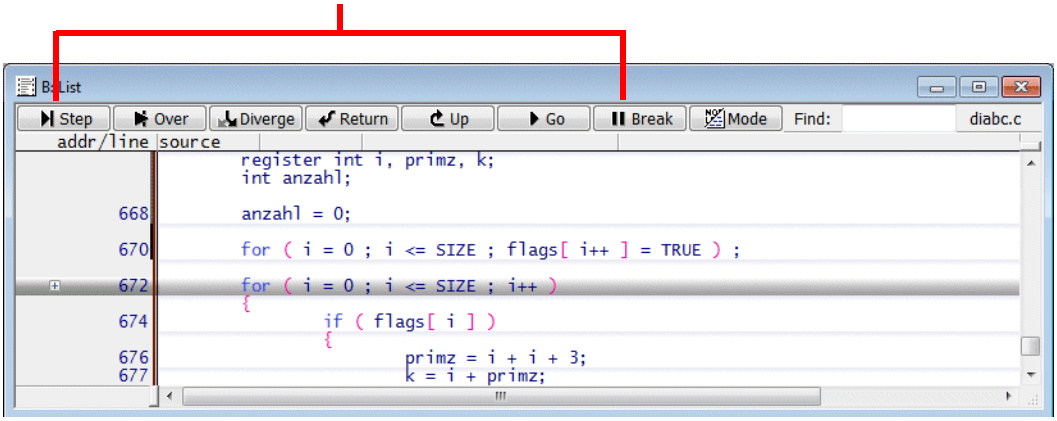
Please be aware of the following:

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.



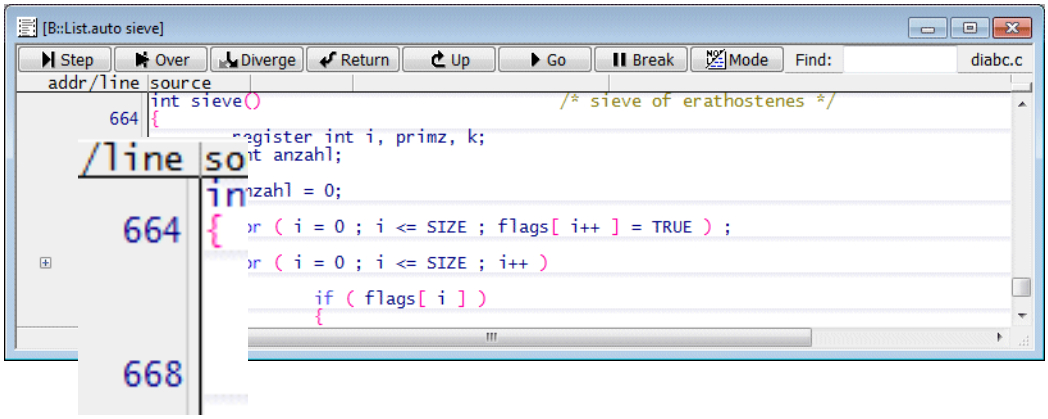
There are local buttons in the **List** window for all basic debug commands




| | |
|---|---|
| **Step** | Single stepping (command: **Step**) |
| **Over** | Step over call (command **Step.Over**). |
| **Diverge** | Exit loops or fast forward to not yet stepped code lines. **Step.Over** is performed repeatedly. |

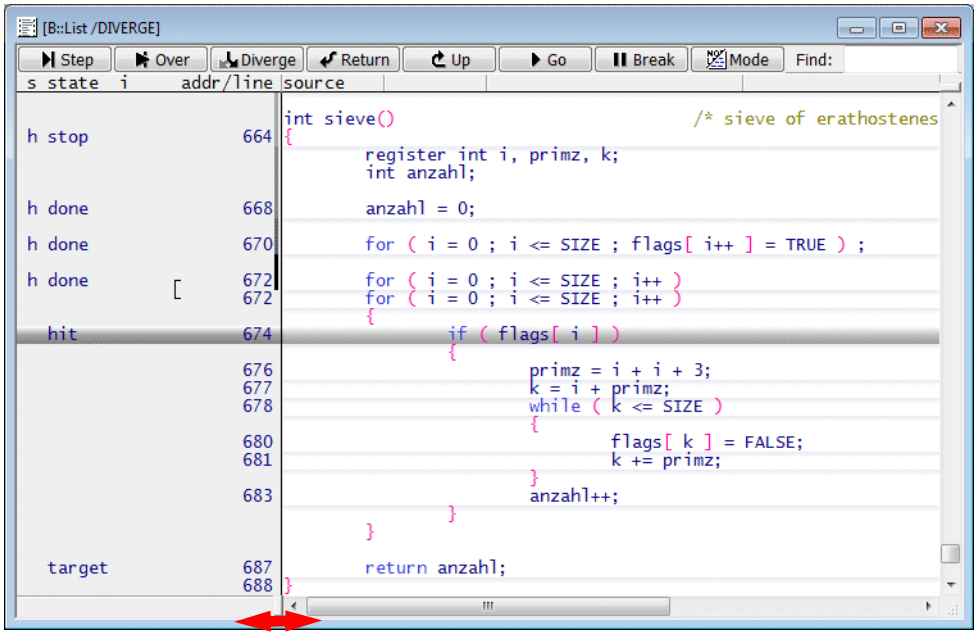
**More details on Step.Diverge**

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.

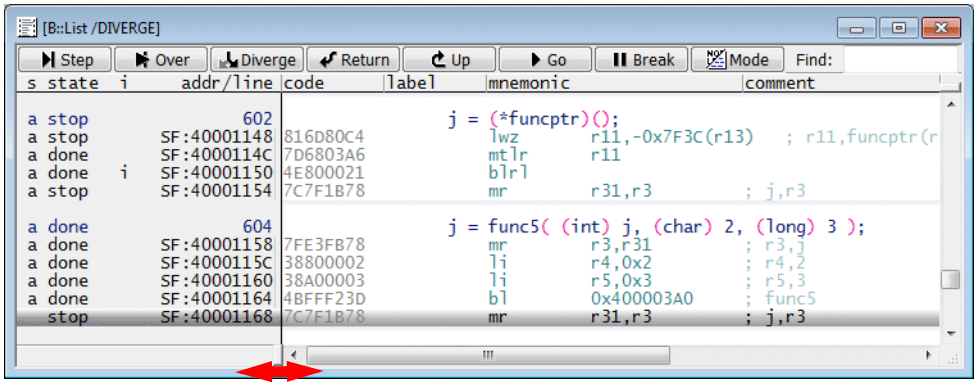



The following command allows you to get more details:

```
List.auto /DIVERGE
```



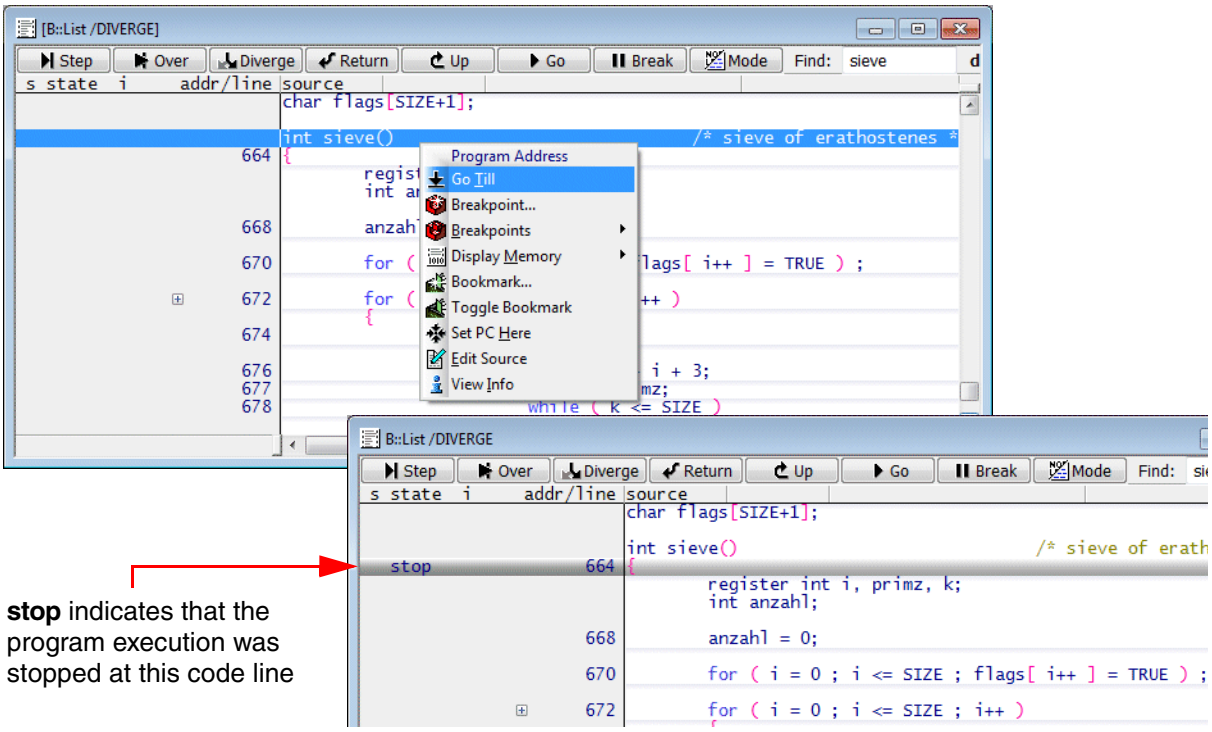

Drag this handle to see the DIVERGE details




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

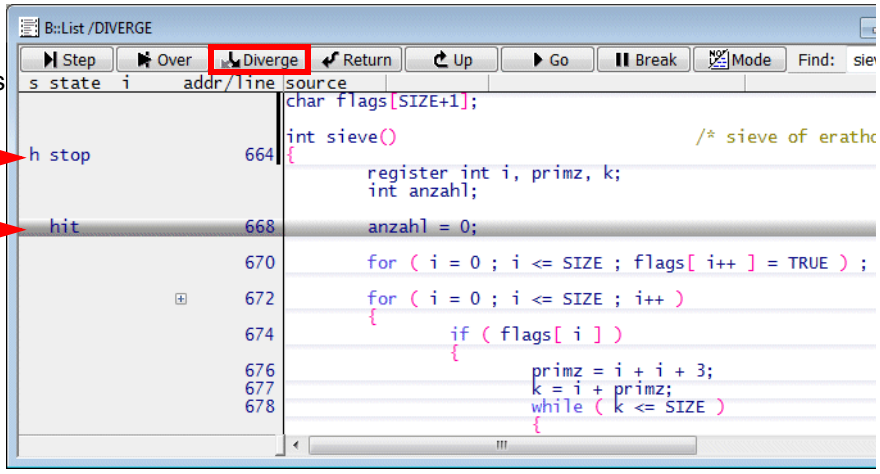
**Example 1:** Diverge through function sieve.

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




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

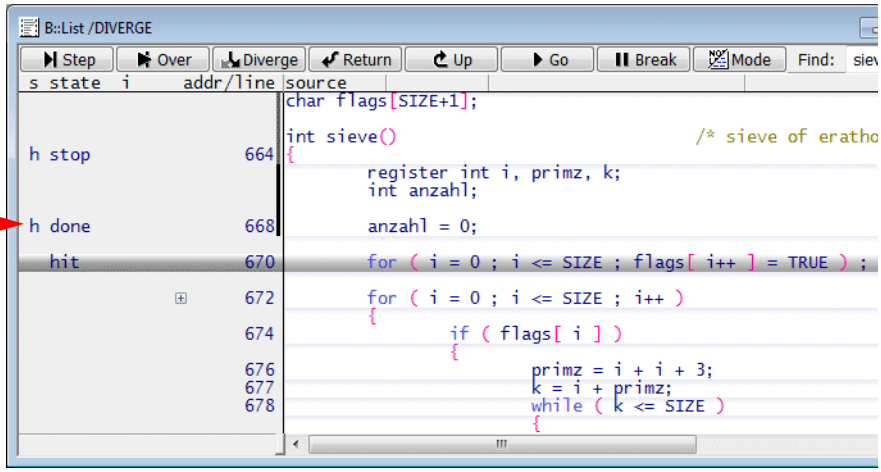
**2. Start a Step.Diverge command.**




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

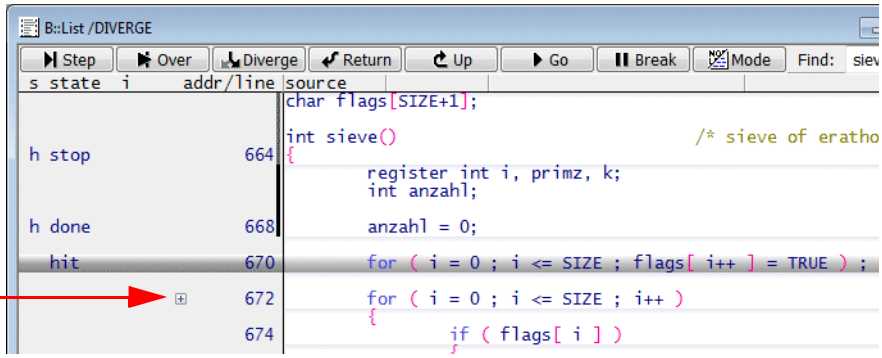
**hit** indicates that this code line was reached by Step command

**3. Continue with Step.Diverge.**

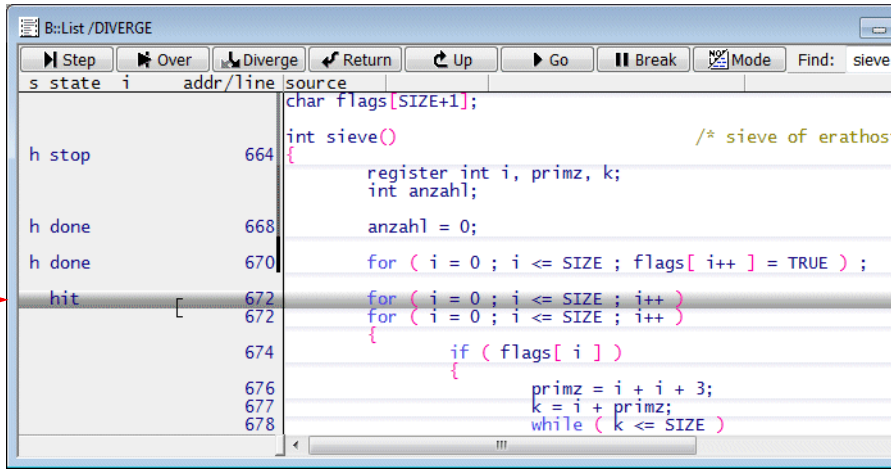



**done** indicates that the code line was reached by a Step command and that a Step command was started from this code line

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

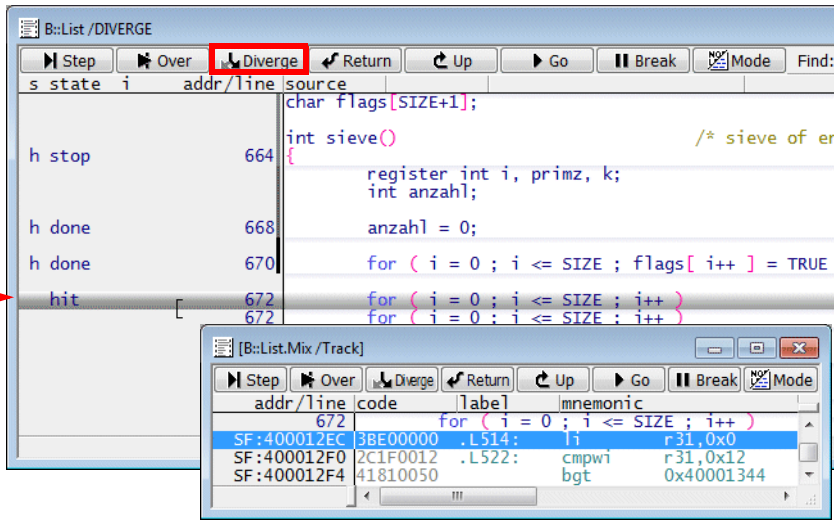



**4. Continue with Step.Diverge.**




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

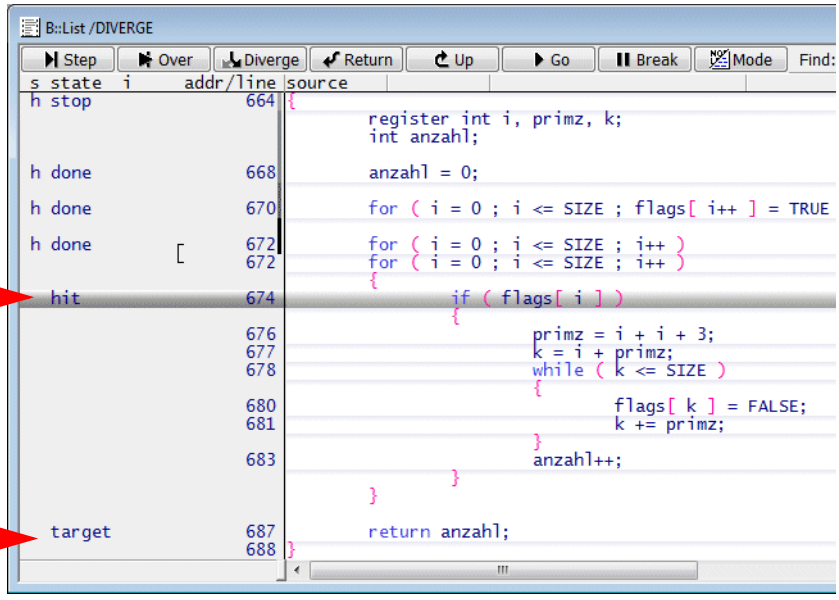
This HLL code line includes a conditional branch




**5. Continue with Step.Diverge.**

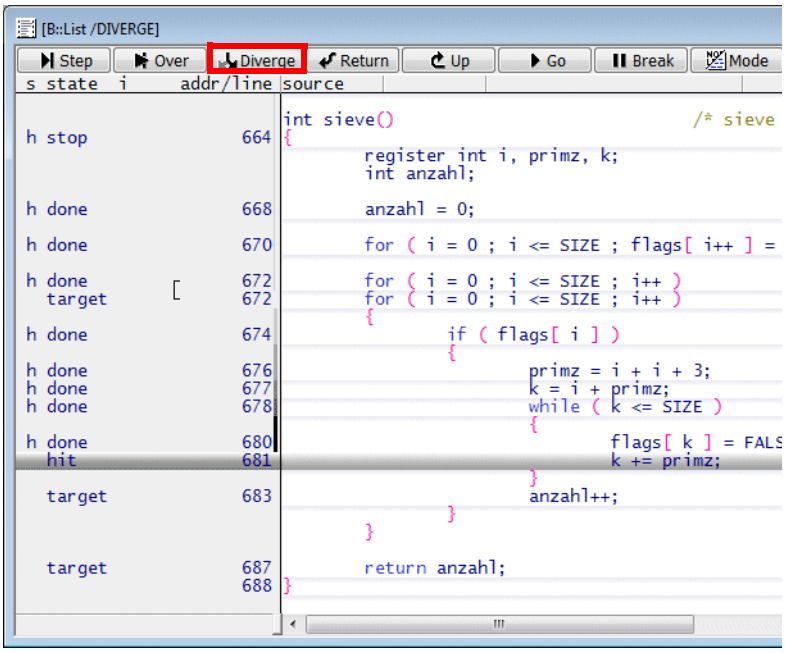
The reached code line is marked as **hit**

The not-reached code line is marked as **target**



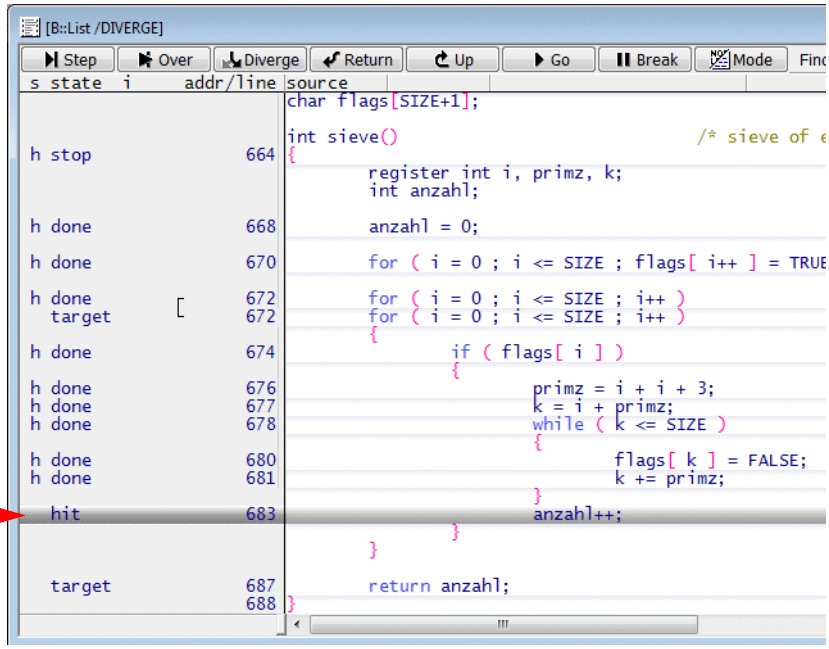
**6. Continue with Step.Diverge (several times).**

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

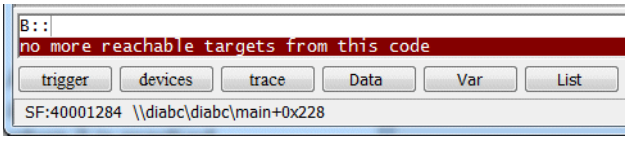



**7. Continue with Step.Diverge.**

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




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

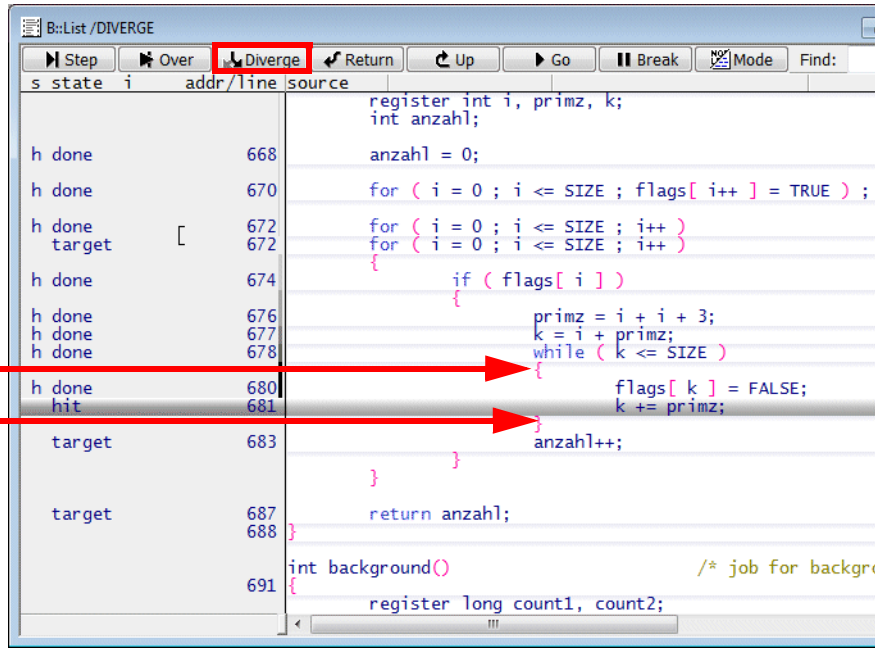


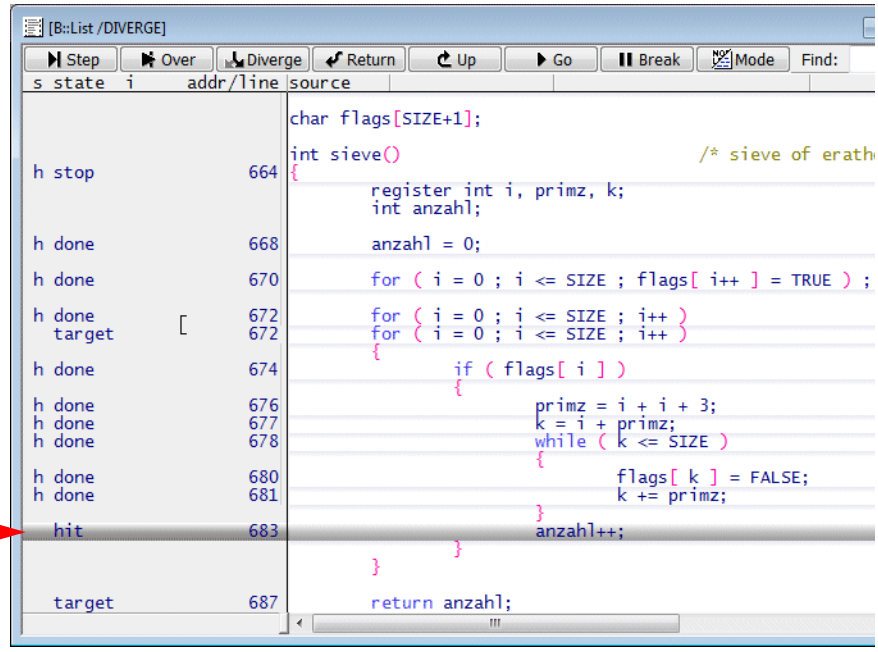
The **DIVERGE marking** is cleared when you use the **Go.direct** command without address or the **Break** command while the program execution is stopped.

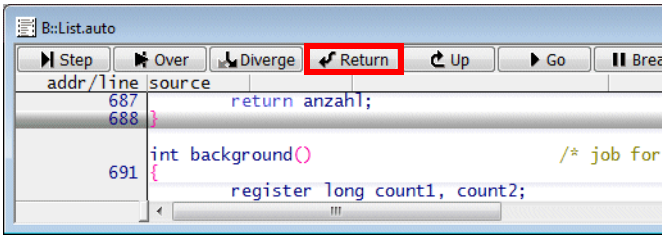
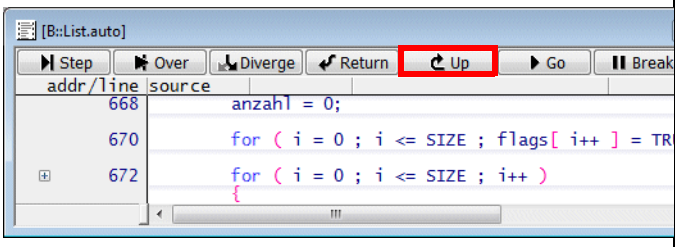
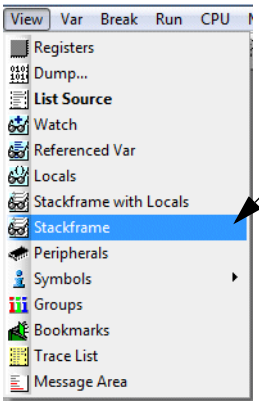
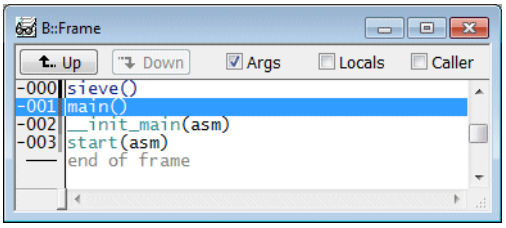
**Example 2:** Exit a loop.

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



| | |
|---|---|
| **Return** | **Return** sets a temporary breakpoint to the last instruction of a function and then starts the program execution.<br>  |
| **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.<br>  <br>  <br>Display the HLL stack to check the function nesting<br>  |

| | |
|---|---|
| **Step** [*<count>*] | Single step |
| **Step.Change** *<expression>* | Step until *<expression>* changes |
| **Step.Till** *<condition>* | Step until *<condition>* becomes true, *<condition>* written in TRACE32 syntax |
| **Var.Step.Change** *<hll_expression>* | Step until *<hll_expression>* changes |
| **Var.Step.Till** *<hll_condition>* | Step until *<hll_condition>* becomes true, *<hll_condition>* as allowed in used programming language |

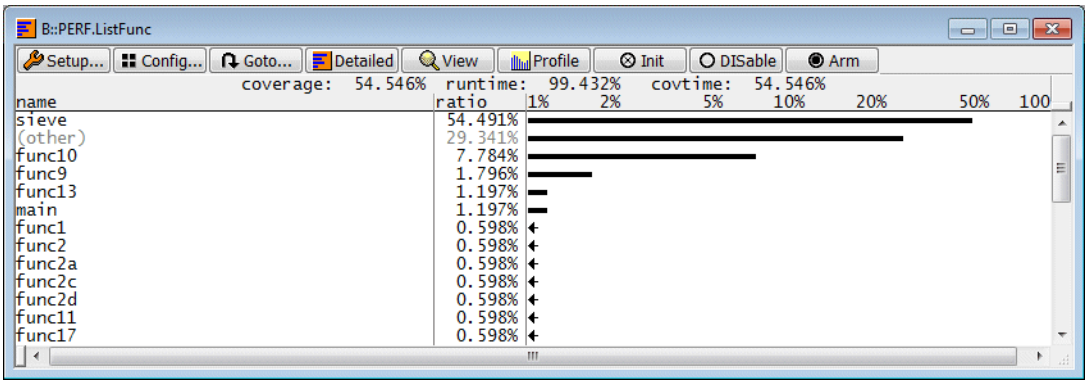
```
Step 10.

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 |

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

## Program Counter Sampling

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

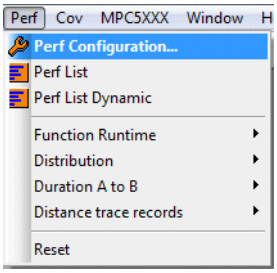


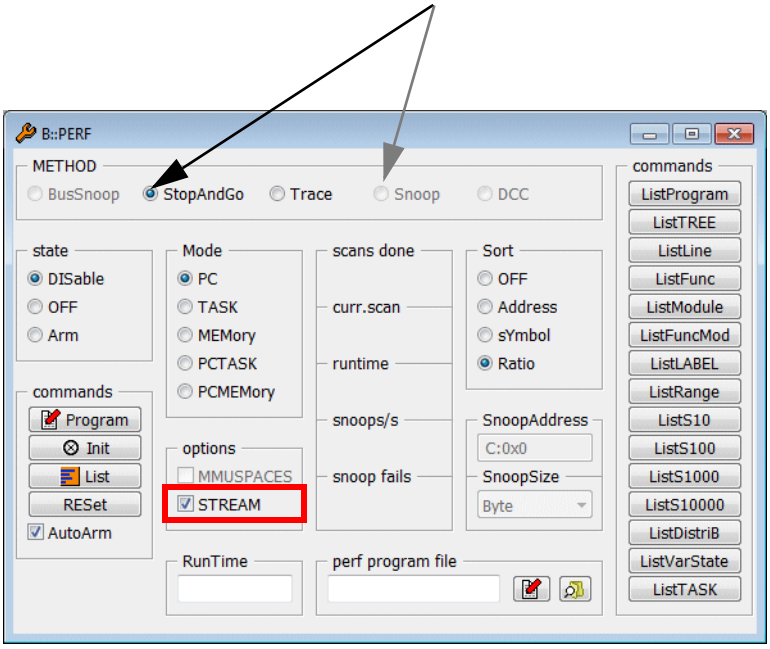

**Measurement procedure:** The Program Counter is sampled periodically. This is implemented in two ways.

- **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.

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.

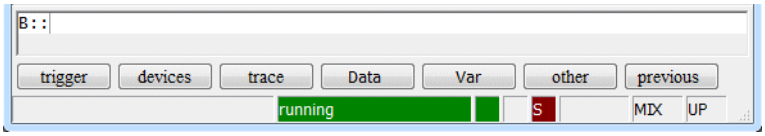
The default METHOD for all other processor architectures is **StopAndGo**.

**Remarks on the StopAndGo method**

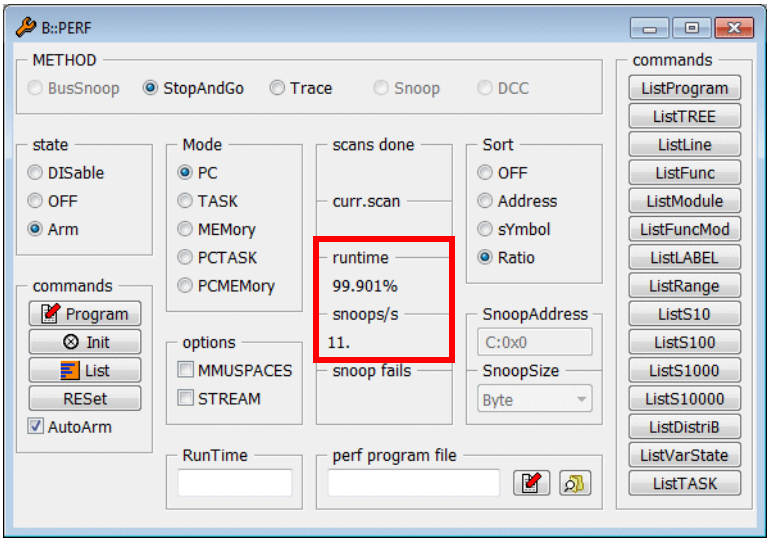
StopAnd Go means that the core is stopped periodically in order to get the actual Program Counter.

| | |
|---|---|
| **STREAM ON** | The software running on the TRACE32 debug hardware initiates the periodic stops. This has the following advantages:<br>• Low intrusive (approx. 50. to 100.us)<br>• More samples per second are possible |
| **STREAM OFF** | The software running on the host initiates the periodic stops.<br>• More intrusive (1 ms in a worst case scenario)<br>• Less samples per second are possible |

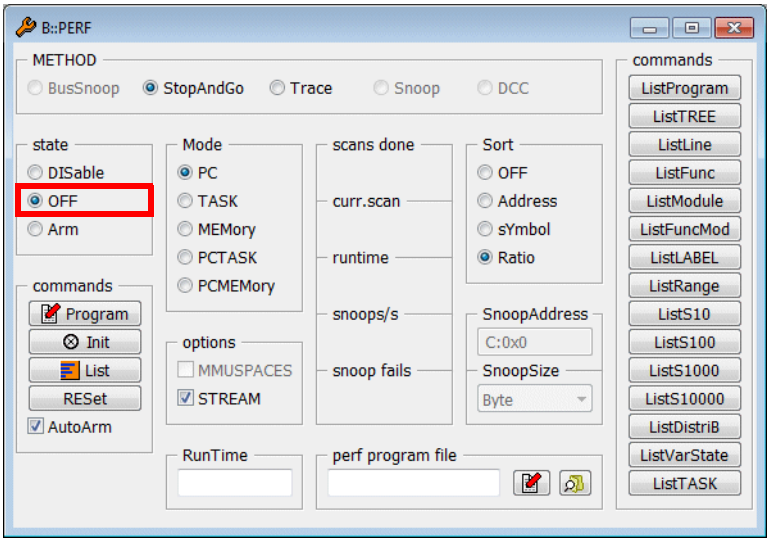
The display of a red **S** in the TRACE32 state line indicates that the program execution is periodically interrupted by the sample-based profiling.




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

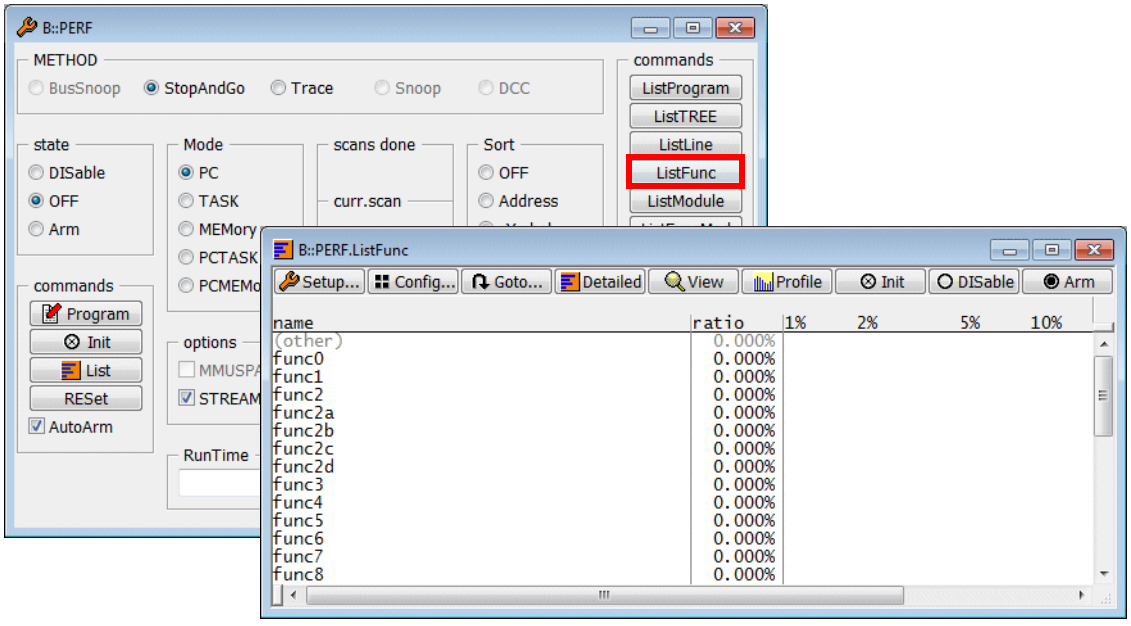


2. **Enable the sample-based profiling by selecting the OFF state.**



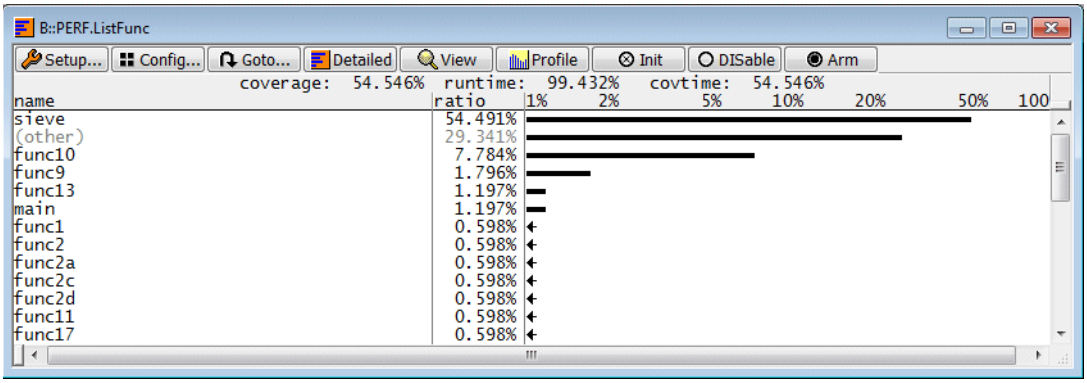

**PERF.OFF** Enable the sample-based profiling

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



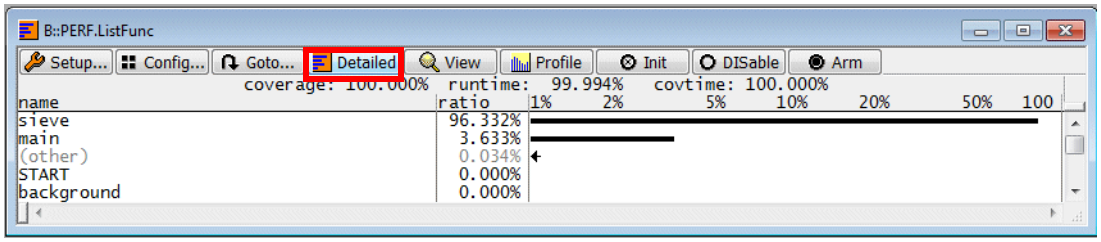

**PERF.ListFunc** Open an HLL function profiling window

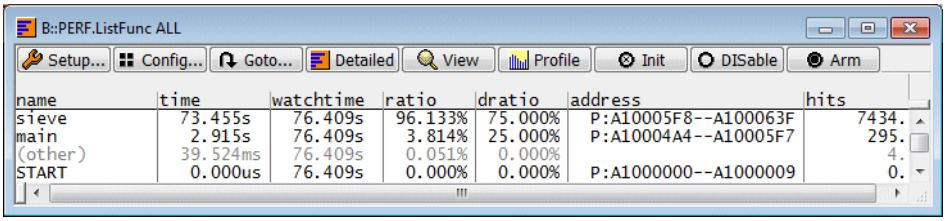
**4. Start the program execution and the sampling.**



## In-depth Result

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

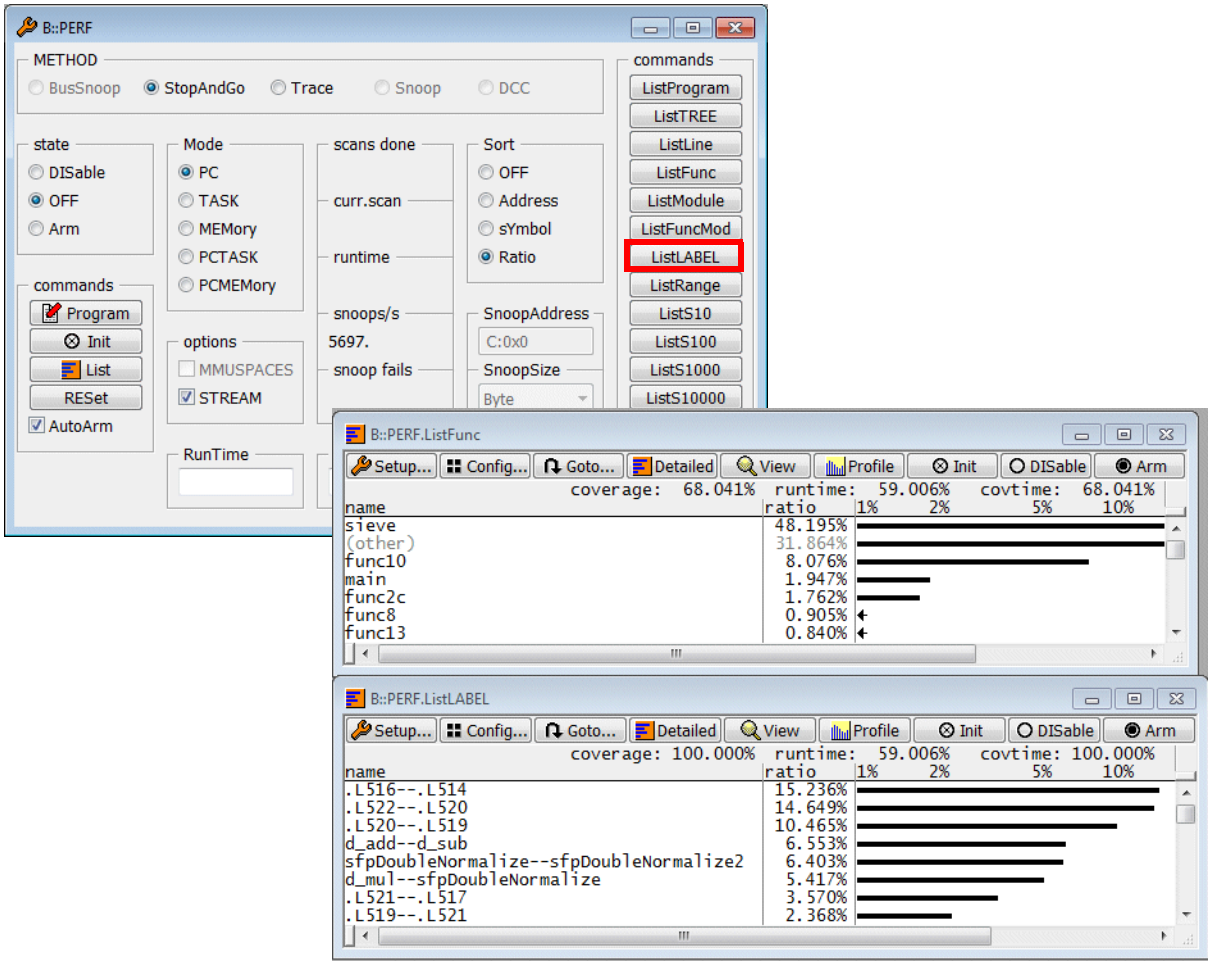



**PERF.ListFunc ALL** Open a detailed HLL function profiling window

| | |
|---|---|
| **name** | Function name |
| **time** | Time in function |
| **watchtime** | Time the function is observed |
| **ratio** | Ratio of time spent by the function in percent |
| **dratio** | Similar to **Ratio**, but only for the last second |
| **address** | Function´s address range |
| **hits** | Number of samples taken for the function |

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.

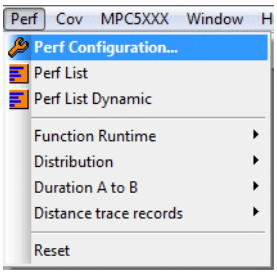


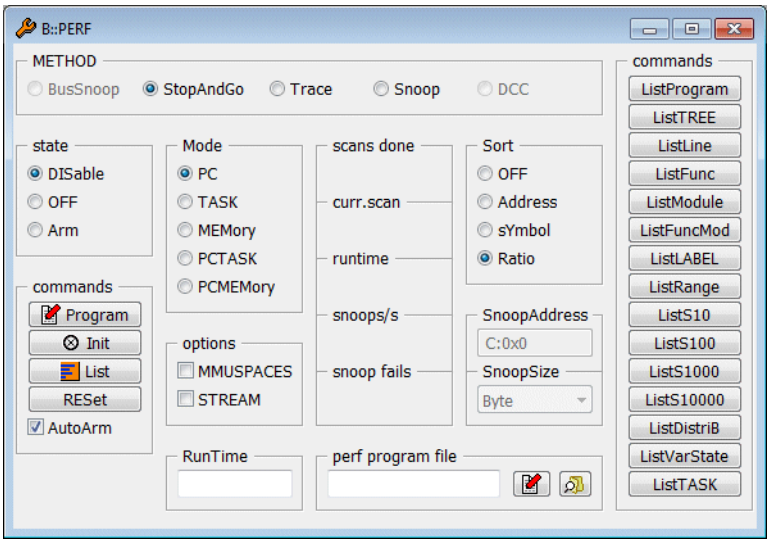

| | |
|---|---|
| **PERF.ListLABEL** | Open a window for label-based profiling |

If OS-aware debugging is configured (refer to **"OS-aware Debugging"** in TRACE32 Glossary, page 31 (glossary.pdf)), TASK information can be sampled.

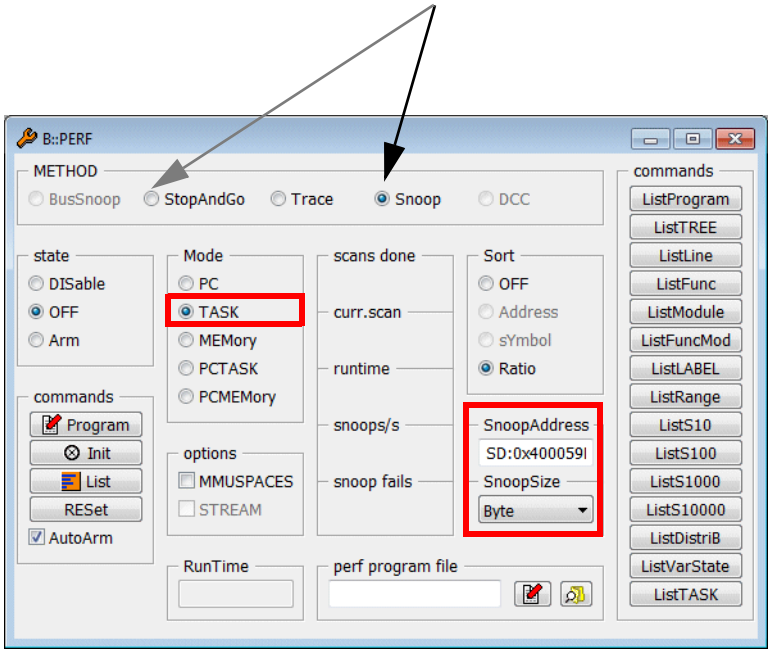
Steps to be taken:

**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.

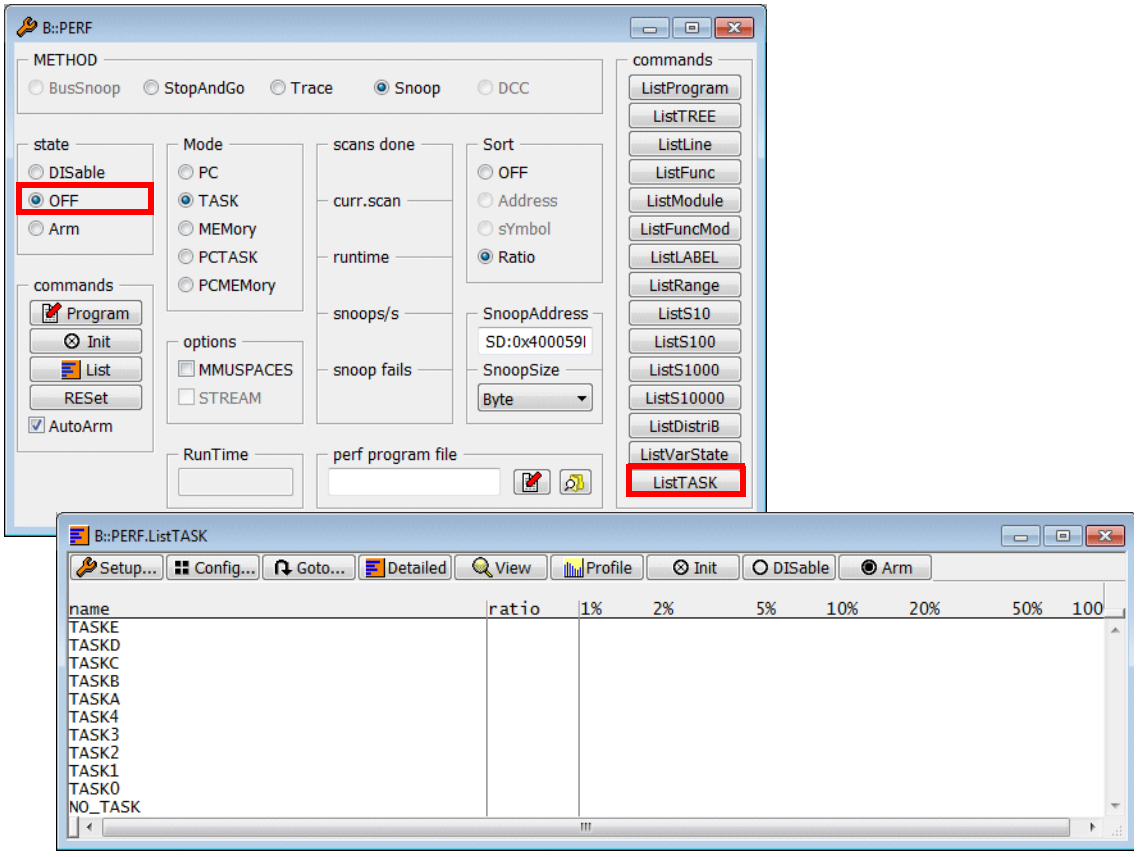
- the **SnoopSize** field with the size of the variable.

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.**