

Training Basic SMP Debugging

Release 09.2023

MANUAL

Training Basic SMP Debugging

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Training	
Debugger Training	
Training Basic SMP Debugging	1
System Concept	6
On-chip Debug Interface	7
Debug Features	7
TRACE32 Tools	8
On-chip Debug Interface plus On-chip Trace Buffer	11
On-chip Debug Interface plus Trace Port	13
NEXUS Interface	14
Starting a TRACE32 PowerView Instance	15
Basic TRACE32 PowerView Parameters	15
Configuration File	15
Standard Parameters	16
Examples for Configuration Files	17
Additional Parameters	21
Application Properties (Windows only)	22
Configuration via T32Start (Windows only)	23
About TRACE32	24
Version Information (SMP)	24
Prepare Full Information for a Support Email	25
Establish your Debug Session	26
TRACE32 PowerView	27
SMP Concept	27
TRACE32 PowerView Components	30
Main Menu Bar and Accelerators	31
Main Tool Bar	33
Window Area	35
Command Line	38
Message Line	41
Softkeys	42
State Line	43
Basic Debugging (SMP)	45

Go/Break	45
Single Stepping on Assembler Level	47
Single Stepping on High-Level Language Level	49
Registers	51
Core Registers	51
Display the Core Registers	51
Colored Display of Changed Registers	52
Modify the Contents of a Core Register	53
Special Function Register	54
Display the Special Function Registers	54
Details about a Single Special Function Register	59
Modify a Special Function Register	60
The PER Definition File	61
Memory Display and Modification	62
The Data.dump Window	64
Display the Memory Contents	64
Modify the Memory Contents	69
Run-time Memory Access	70
Colored Display of Changed Memory Contents	80
The List Window	81
Displays the Source Listing Around the PC	81
Displays the Source Listing of a Selected Function	82
Breakpoints	84
Breakpoint Implementations	84
Software Breakpoints in RAM	84
Onchip Breakpoints in NOR Flash	86
Onchip Breakpoints on Read/Write Accesses	90
Onchip Breakpoints by Processor Architecture	91
ETM Breakpoints for ARM or Cortex-A/-R	92
Breakpoint Types	95
Program Breakpoints	96
Read/Write Breakpoints	98
Breakpoint Handling	100
Breakpoint Setting at Run-time	100
Real-time Breakpoints vs. Intrusive Breakpoints	101
Break.Set Dialog Box	105
The HLL Check Box - Function Name	106
The HLL Check Box - Program Line Number	108
The HLL Check Box - Variable	109
The HLL Check Box - HLL Expression	111
Implementations	114
Actions	115

Options	119
DATA Breakpoints	123
Advanced Breakpoints	126
TASK-aware Breakpoints	127
Intrusive TASK-aware Breakpoint	127
Real-time TASK-aware Breakpoint	129
COUNTER	132
Software Counter	132
CONDition	135
CMD	142
Display a List of all Set Breakpoints	145
Delete Breakpoints	146
Enable/Disable Breakpoints	146
Store Breakpoint Settings	147
Debugging	148
Debugging of Optimized Code	148
Basic Debug Control	151

System Concept

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

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

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

On-chip Debug Interface

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

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

Debug Features

Depending on the processor architecture different debug features are available.

Debug features provided by all multi-core chips:

- Read/write access to the registers of all cores
- Read/write access to memories
- Start/stop of the program execution
- Start/stop synchronization for all cores

Debug features specific for each multi-core chip:

- 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:

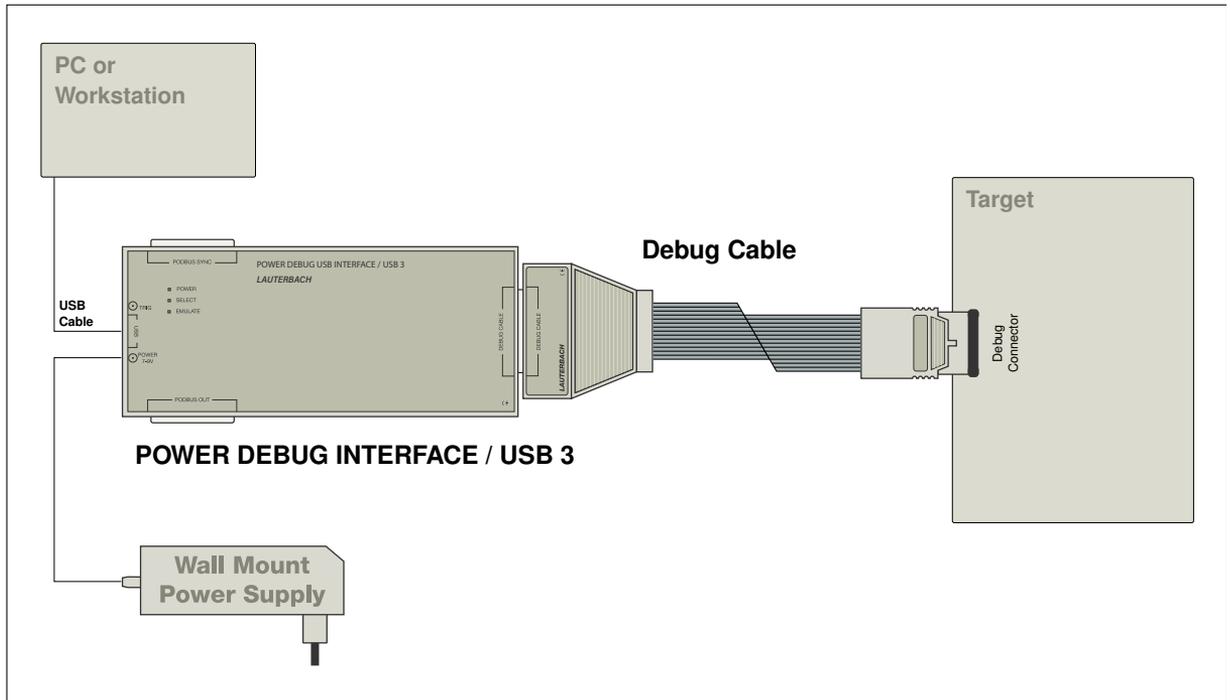
- Universal debugger hardware
- Debug cable specific to the processor architecture

For SMP debugging the debug cable needs to provide a **License for Multicore Debugging**.

This is not required for the following debug cables:

- ARMv8-A
- Intel Atom/x86
- Hexagon
- PowerArchitecture QorIQ 32- and 64-bit

because all these cores are always implemented in a multi-core chip.

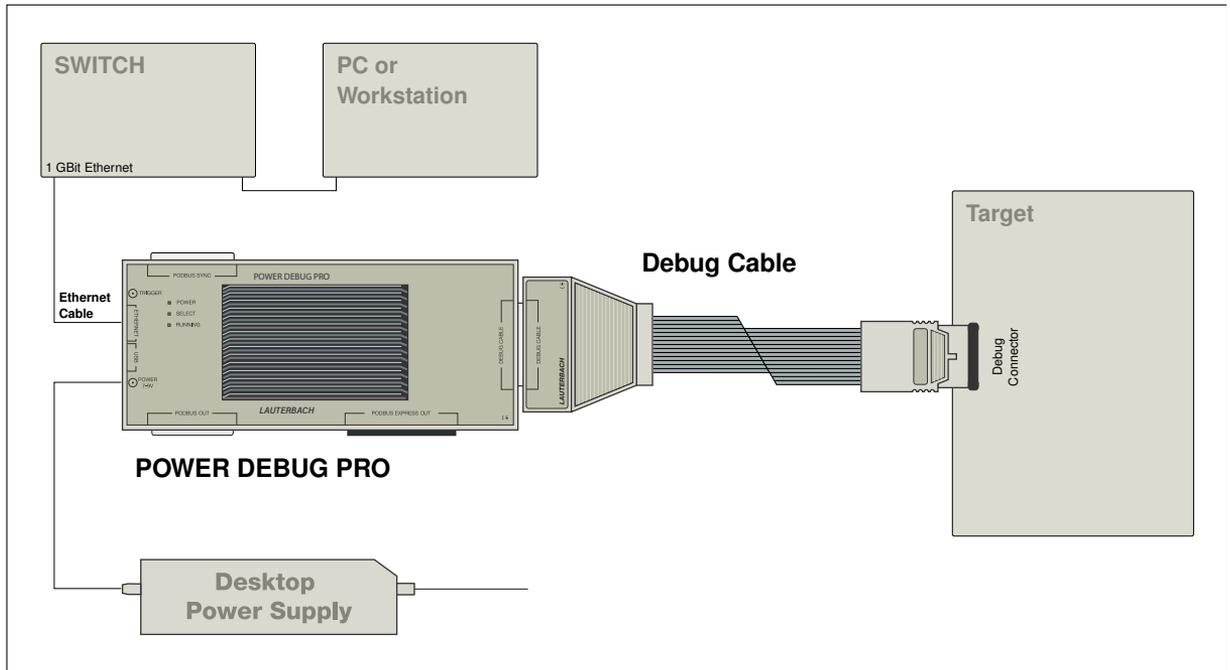


Current module:

- 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

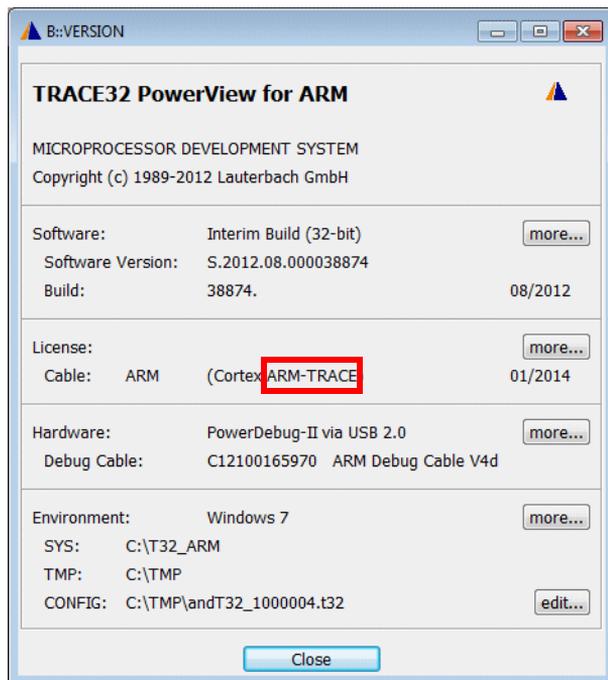
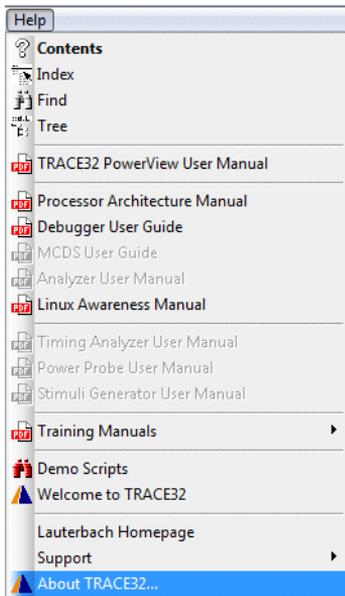
The on-chip trace buffer can store core-trace information:

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

System trace information and bus trace information is also possible.

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 CoreSight ETM Tracing”** (training_arm_etm.pdf).
- **“Training Cortex-M Tracing”** (training_cortexm_etm.pdf).
- **“Training AURIX Tracing”** (training_aurix_trace.pdf).
- **“Training Hexagon ETM Tracing”** (training_hexagon_etm.pdf).
- **“Training Nexus Tracing”** (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 core-trace information:

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

System trace information and bus trace information is also possible.

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

- [“Training Arm CoreSight ETM Tracing”](#) (training_arm_etm.pdf)
- [“Training Cortex-M Tracing”](#) (training_cortexm_etm.pdf)
- [“Training AURIX Tracing”](#) (training_aurix_trace.pdf)
- [“Training Hexagon ETM Tracing”](#) (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 of all cores
- Read/write access to all memories
- Start/stop synchronization for all cores
- 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 Tracing”](#) (training_nexus.pdf).

Starting a TRACE32 PowerView Instance

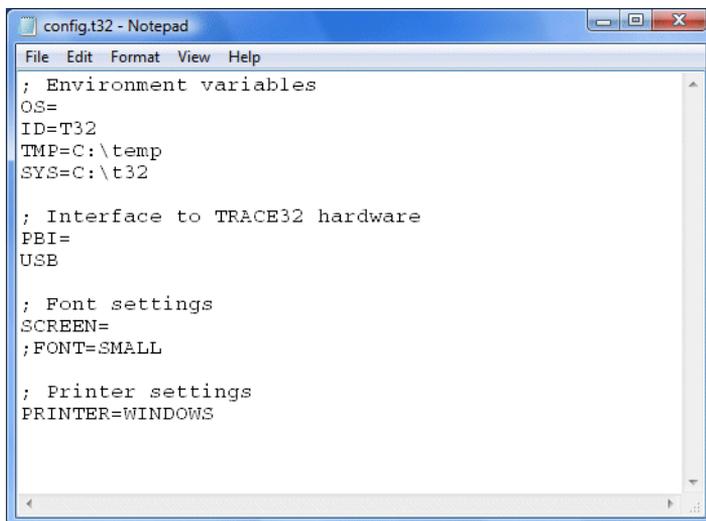
Basic TRACE32 PowerView Parameters

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

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

Configuration File

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



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

; Interface to TRACE32 hardware
PBI=
USB

; Font settings
SCREEN=
;FONT=SMALL

; Printer settings
PRINTER=WINDOWS
```

The following rules apply to the configuration file:

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

Standard Parameters

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

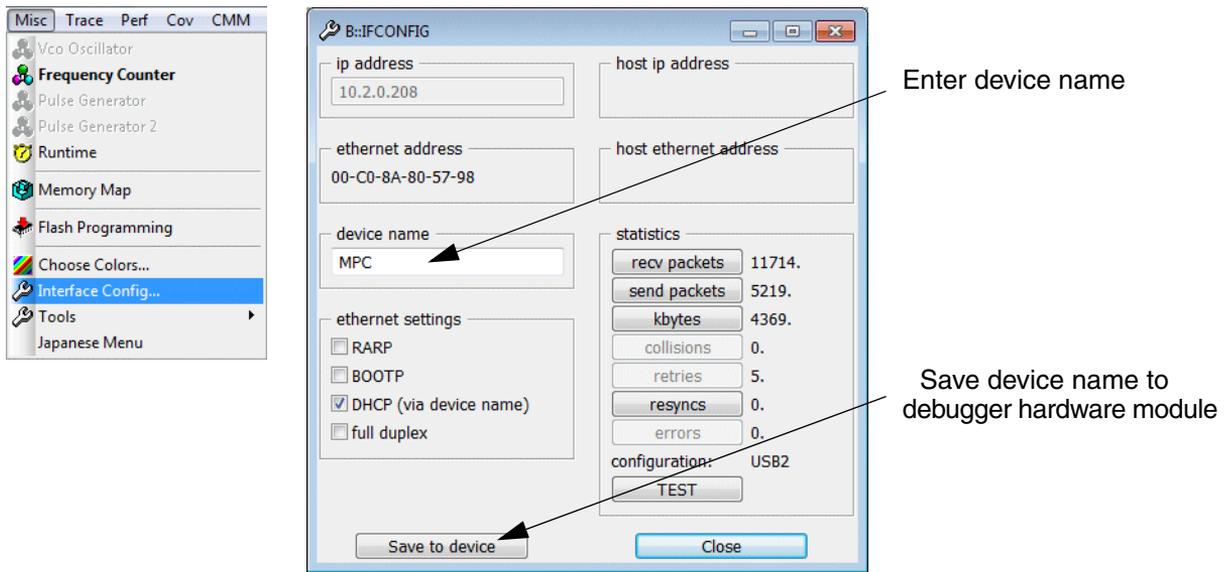
```
; Host interface
PBI=
USB
NODE=training1       ; NODE name of TRACE32

; 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

Please be aware that USB device names are case-sensitive

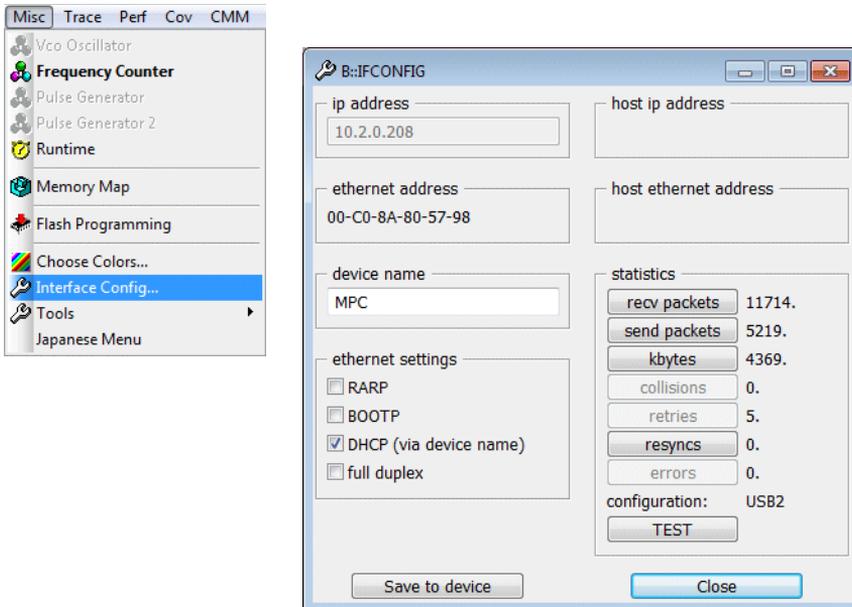
TRACE32 allows to communicate with a POWER DEBUG INTERFACE USB from a remote PC. For an example, see [“Example: Remote Control for POWER DEBUG INTERFACE / USB”](#) in TRACE32 Installation Guide, page 56 (installation.pdf).

```
; Host interface
PBI=
NET
NODE=training1

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

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

Ethernet Configuration and Operation Profile



IFCONFIG

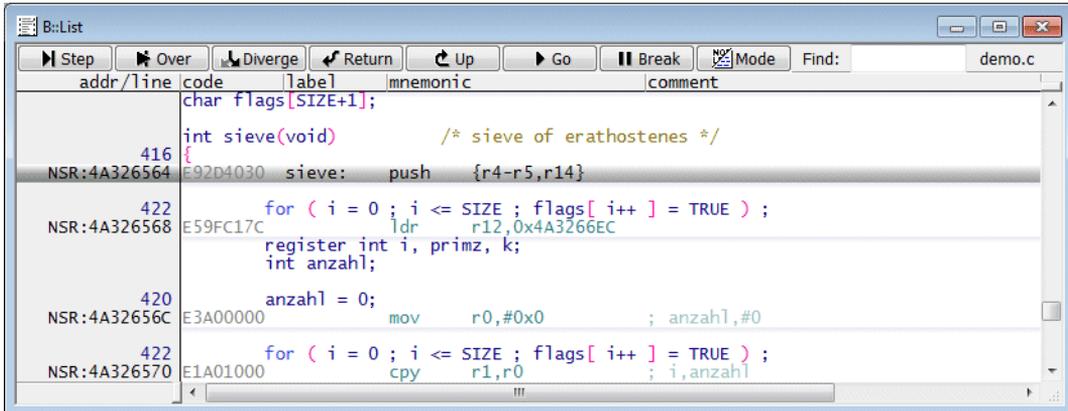
Dialog to display and change information for the Ethernet interface

Additional Parameters

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

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

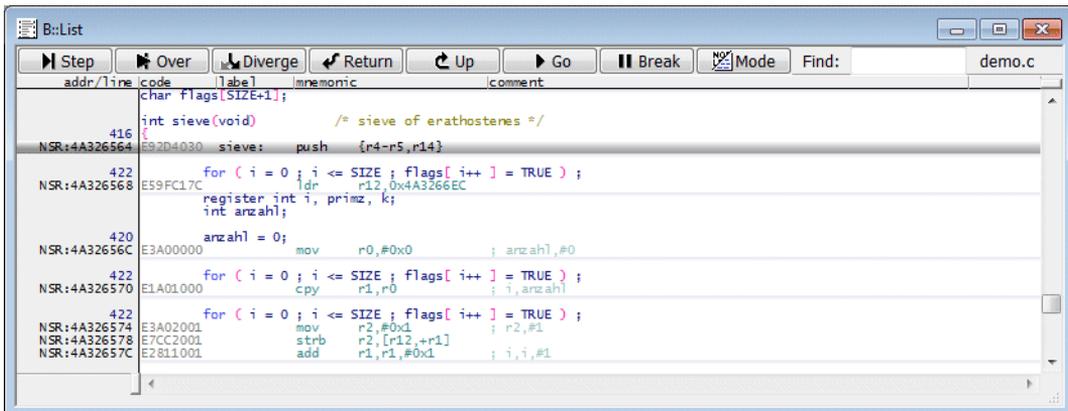
Display with normal font:



The screenshot shows the TRACE32 interface with the assembly code for 'demo.c'. The font size is normal, making the code easy to read. The code includes a function 'sieve' with a loop and several instructions.

```
char flags[SIZE+1];
int sieve(void) /* sieve of erathostenes */
{
NSR:4A326564 E92D4030 sieve: push {r4-r5,r14}
422 NSR:4A326568 E59FC17C for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
register int i, primz, k;
int anzahl;
420 NSR:4A32656C E3A00000 anzahl = 0;
mov r0,#0x0 ; anzahl,#0
422 NSR:4A326570 E1A01000 for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
cpy r1,r0 ; i,anzahl
```

Display with small font:



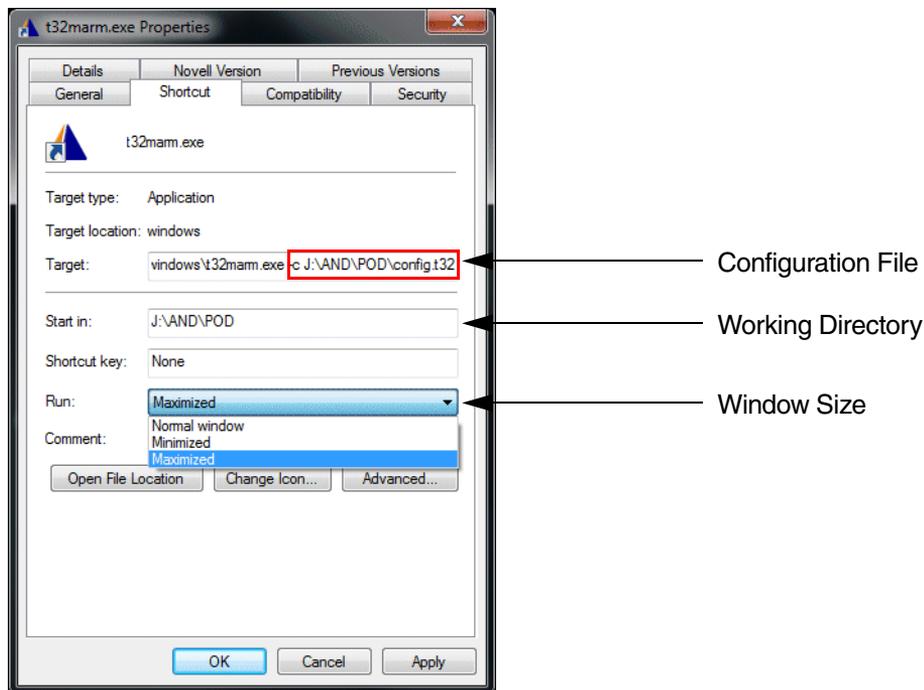
The screenshot shows the TRACE32 interface with the assembly code for 'demo.c'. The font size is small, making the code difficult to read. The code includes a function 'sieve' with a loop and several instructions.

```
char flags[SIZE+1];
int sieve(void) /* sieve of erathostenes */
{
NSR:4A326564 E92D4030 sieve: push {r4-r5,r14}
422 NSR:4A326568 E59FC17C for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
register int i, primz, k;
int anzahl;
420 NSR:4A32656C E3A00000 anzahl = 0;
mov r0,#0x0 ; anzahl,#0
422 NSR:4A326570 E1A01000 for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
cpy r1,r0 ; i,anzahl
422 NSR:4A326574 E3A02001 for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
mov r2,#0x1 ; r2,#1
NSR:4A326578 E7CC2001 strb r2,[r12,+r1]
NSR:4A32657C E2811001 add r1,r1,#0x1 ; i,i,#1
```

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

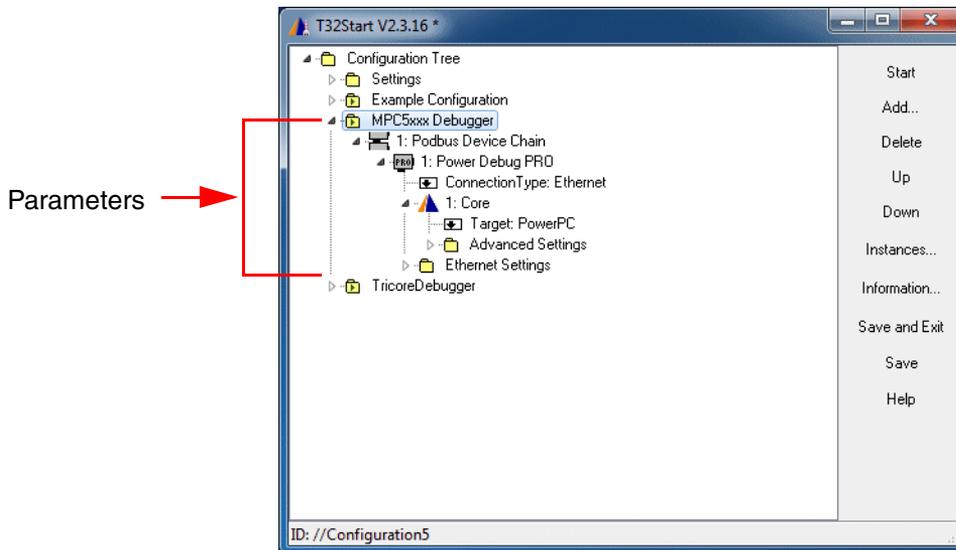
Definition of the Window Size for TRACE32 PowerView

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**.

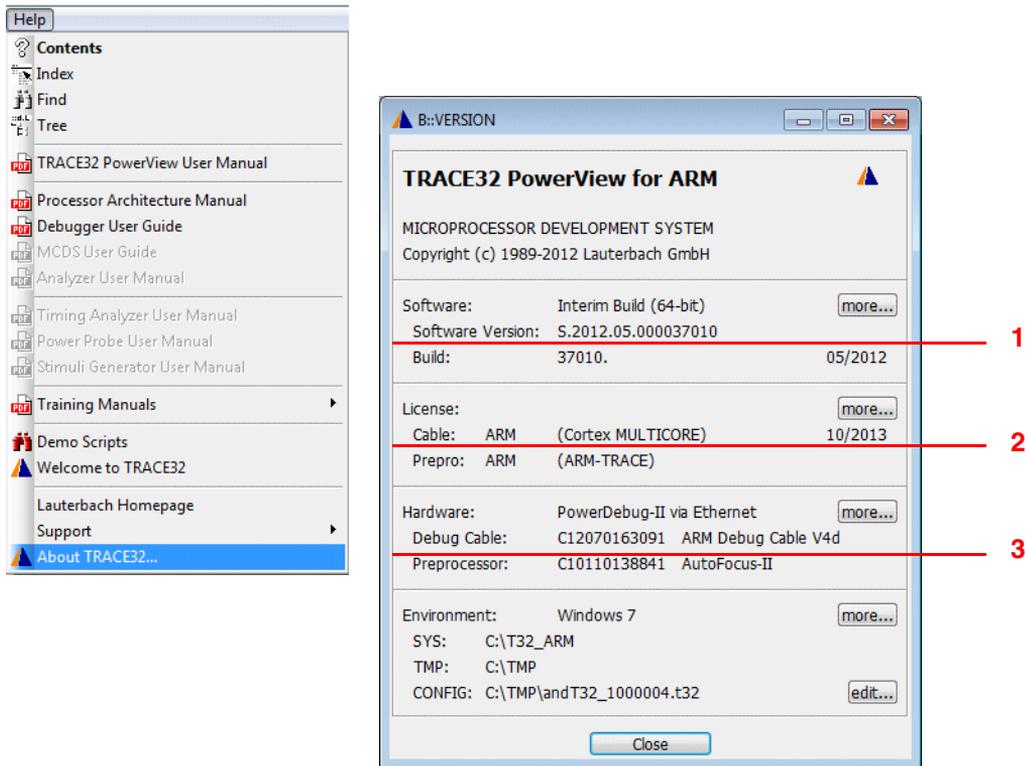
A detailed online help for **t32start.exe** is available via the **Help** button or in **“T32Start”** (app_t32start.pdf).



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 (SMP)



The VERSION window informs you about:

1. the version of the TRACE32 software
2. the debug licenses programmed into the debug cable, the multicore license, the expiration date of your software guarantee 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.

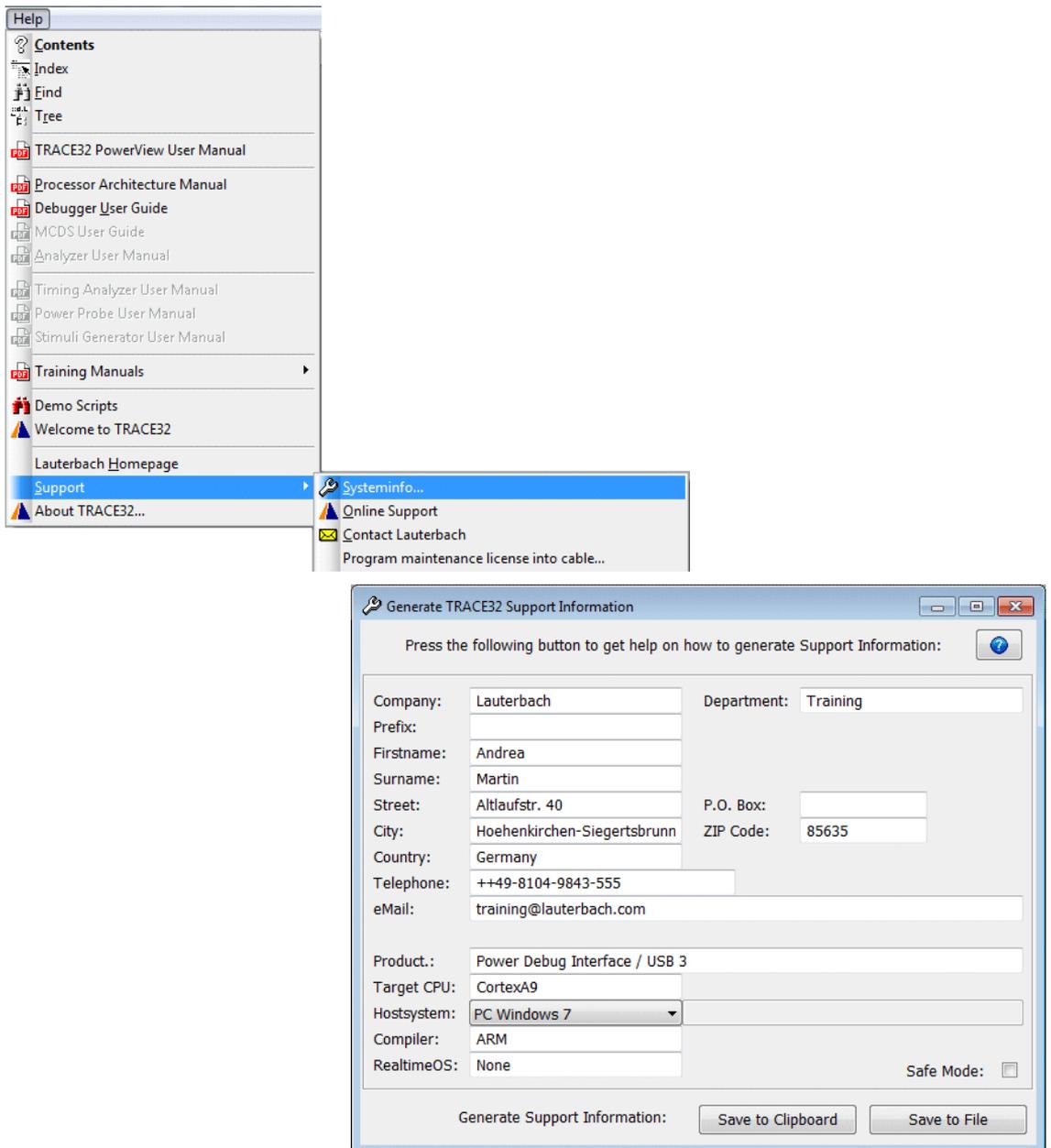
VERSION.SOFTWARE

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**.



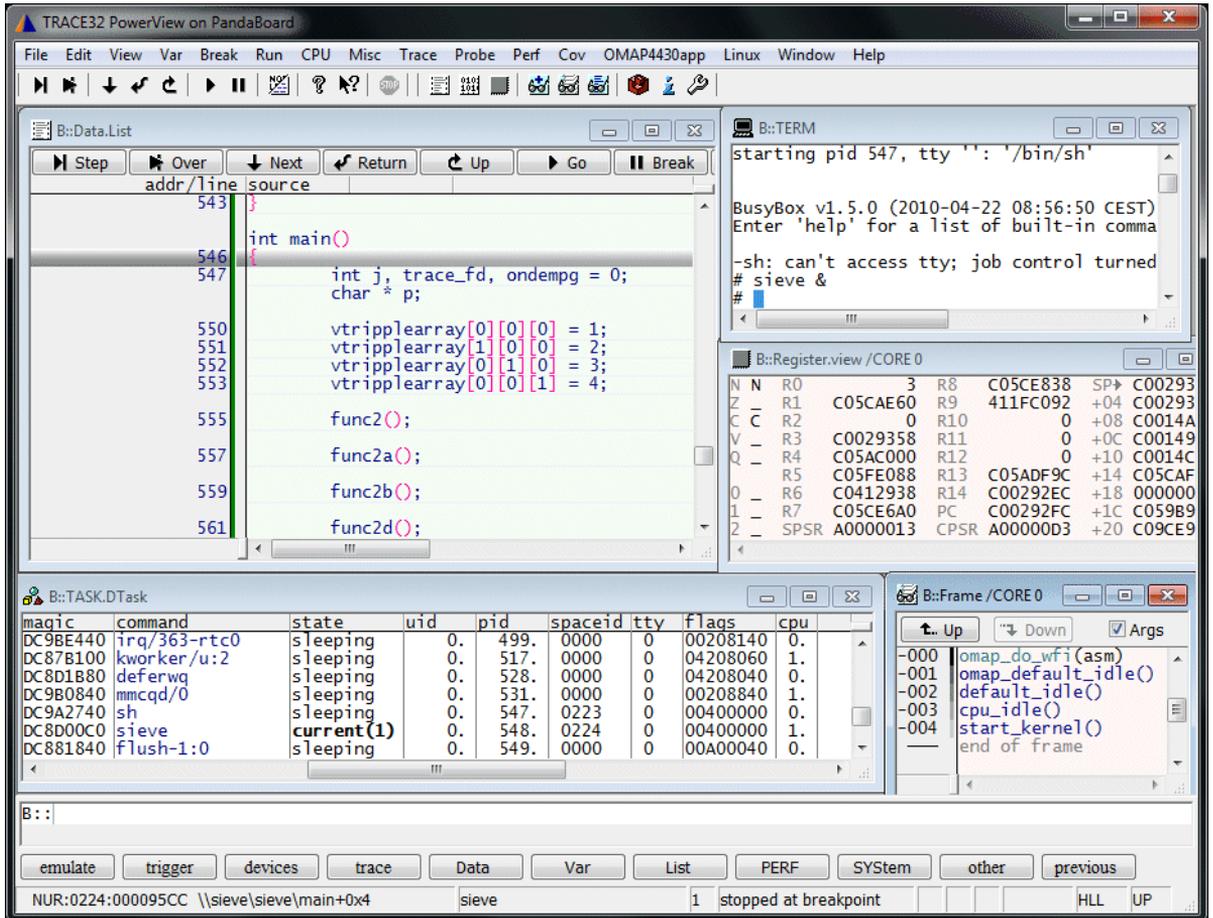
2. Preferred: click **Save to File**, and send the system information as an attachment to your e-mail.
3. Click **Save to Clipboard**, and then paste the system information into your e-mail.

Establish your Debug Session

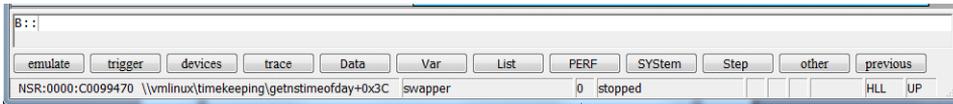
Before you can start debugging, the debug environment has to be set up. An overview on the most common setups is given in [“Establish Your Debug Session”](#) (tutor_setup.pdf).

SMP Concept

One TRACE32 PowerView GUI is opened to control all cores and to visualize all system information.



In the TRACE32 PowerView GUI one core is the selected one.



The **Cores** field in the state line displays the number of the currently selected core

The fact that one core is the selected one has the following consequences:

- By default system information is visualized from the perspective of the selected core.

```
; core 0 is the selected core

List                               ; display a source listing around
                                   ; the program counter of core 0

Register.view                       ; display the register contents of
                                   ; core 0
```

- System information from the perspective of another core can be visualized by using the option **CORE <number>**.

```
List /CORE 1                       ; display a source listing around
                                   ; the program counter of core 1

Register.view /CORE 1              ; display the register contents of
                                   ; the core 1
```

The selected core can be change by selecting another core via the **Cores** pull-down menu or via the **CORE.select** command:



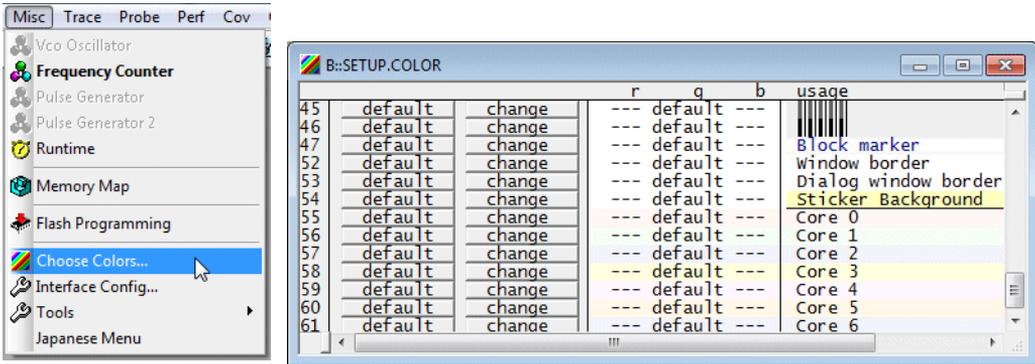
CORE.select <number> Select a different core

TRACE32 PowerView distinguishes two types of information:

- **Core-specific information** which is displayed on a colored background.

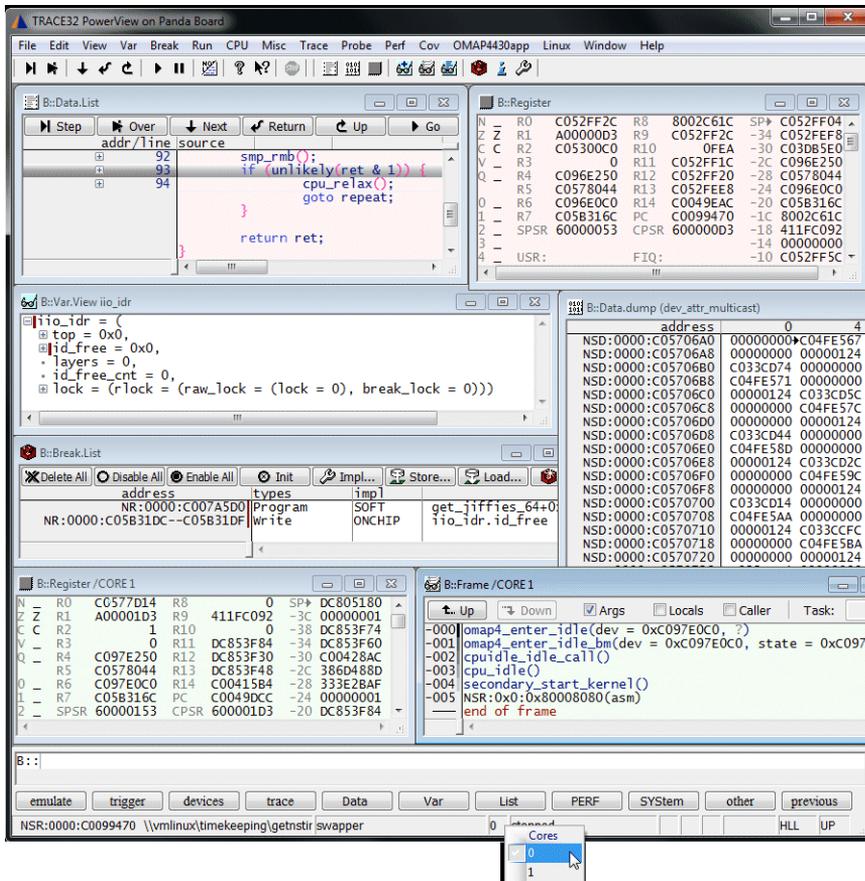
Typical core-specific information are: register contents, source listing of the code currently executed by the core, the stack frame.

TRACE32 PowerView uses predefined color settings for the cores.



- **Information common for all core** which is displayed on a white background.

Typical common information are: memory contents, values of variables, breakpoint setting.

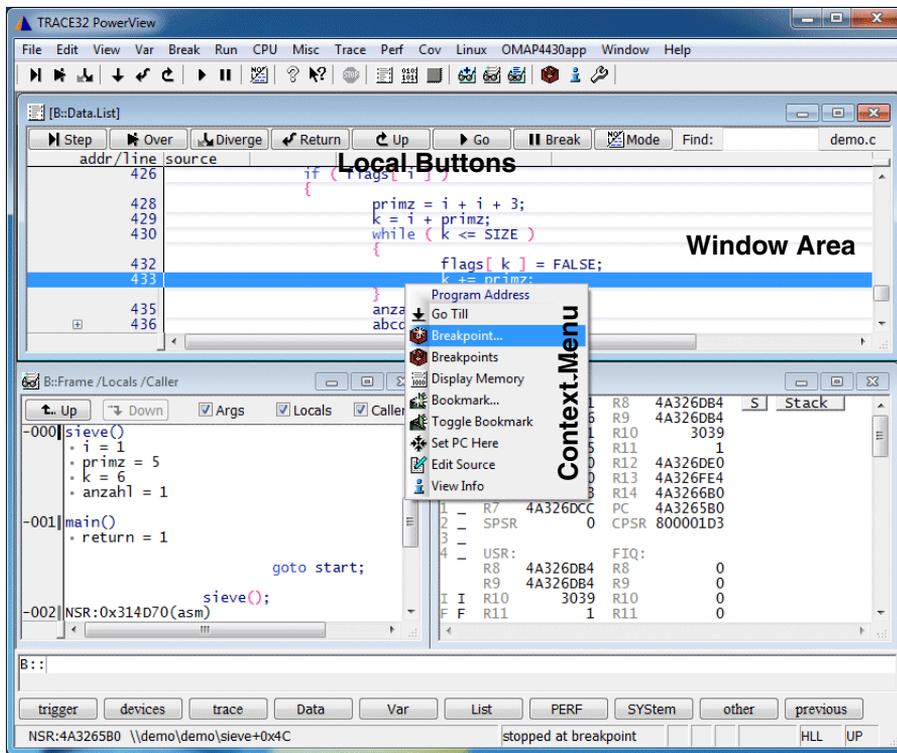


Core-specific information (here for the currently selected one)

Information common for all cores

Core-specific information (here for the core 1)

TRACE32 PowerView Components



← Main Menu Bar
← Main Tool Bar

← Command Line
← Message Line
← SoftkeyLine
← State Line



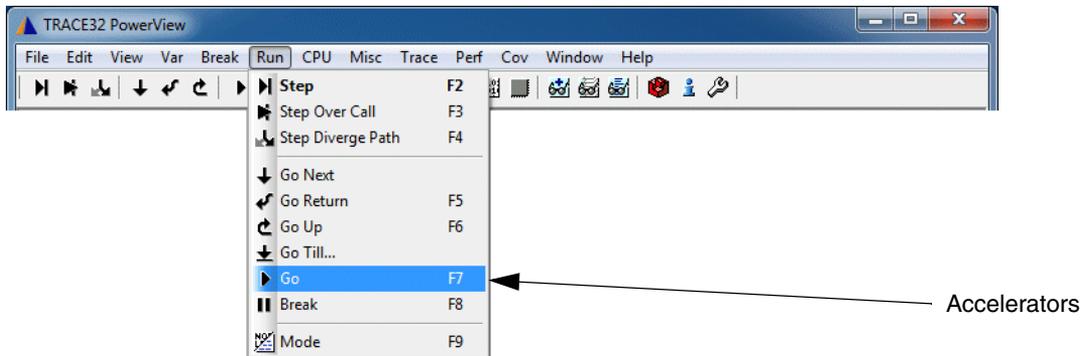
The structure of the menu bar and the tool bar are defined by the file `t32.men` which is located in the TRACE32 system directory.

TRACE32 allows you to modify the menu bar and the tool bar so they will better fit your requirements. Refer to [“Training Menu Programming”](#) (training_menu.pdf) for details.

Main Menu Bar and Accelerators

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

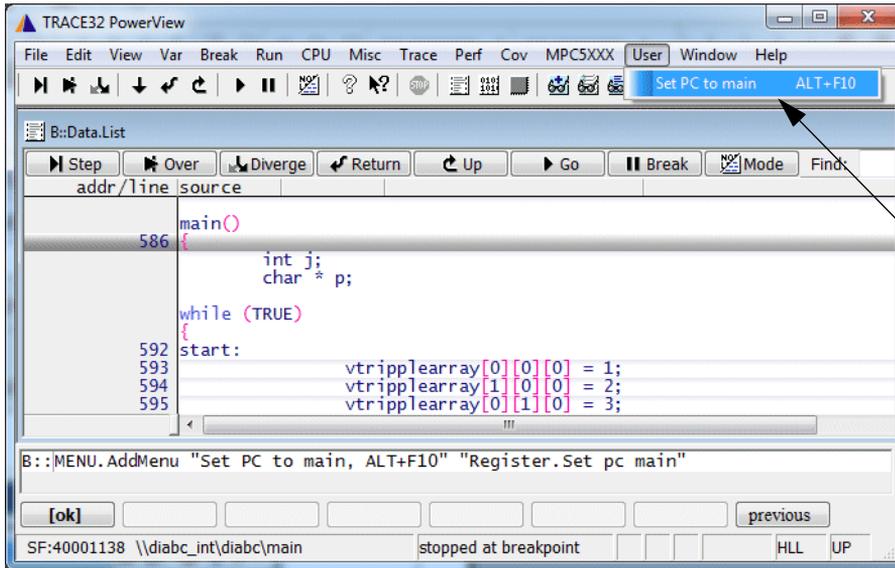
For often used commands accelerators are defined.



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"
```



User Menu



For more complex changes to the main menu bar refer to [“Training Menu Programming”](#) (training_menu.pdf).

Videos about the menu programming can be found here:
support.lauterbach.com/kb/articles/trace32-user-interface-customization

Main Tool Bar

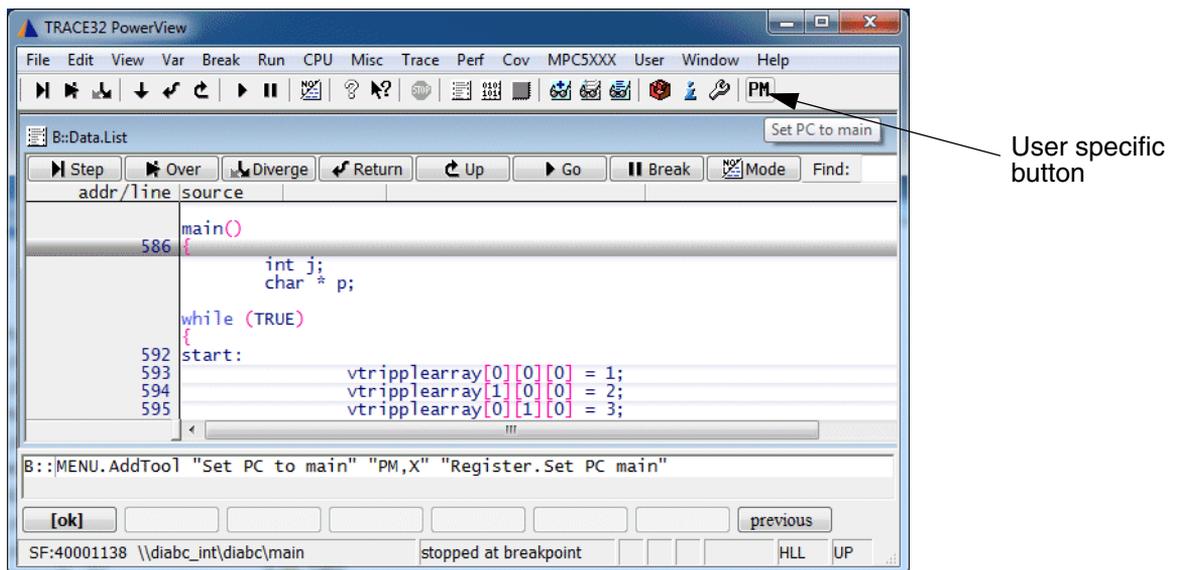
The main tool bar provides fast access to often used commands.

The user can add his own buttons very easily:

MENU.AddTool <tooltip_text> <tool_image> <command>	Add a button to the toolbar
MENU.RESet	Reset menu to default

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

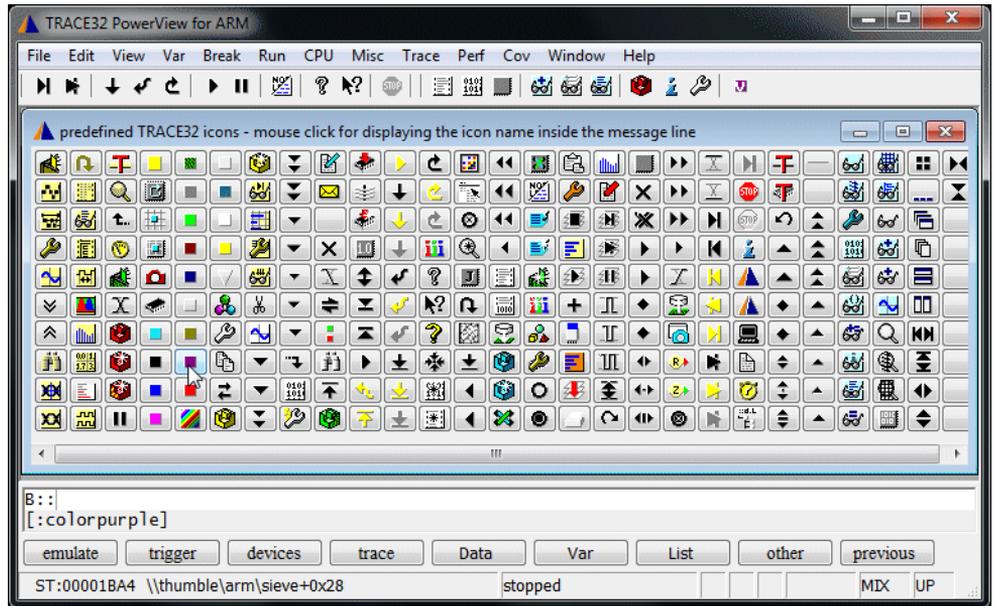
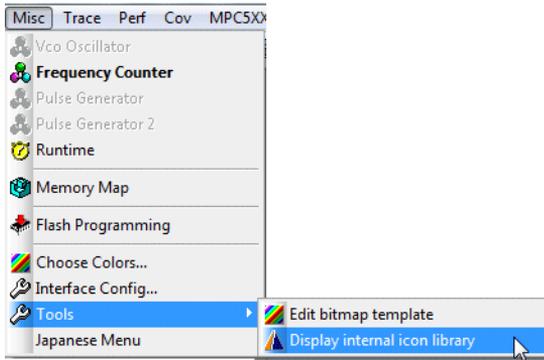
```
MENU.AddTool "Set PC to main" "PM,X" "Register.Set PC main"
```



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

TRACE32 Documents -> IDE User Interface -> PowerView Command Reference -> MENU -> Programming Commands -> [TOOLITEM](#).

All predefined TRACE32 icons can be inspected as follows:



Or by following TRACE32 command:

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

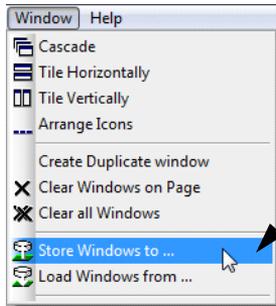
The predefined icons can easily be used to create new icons.

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

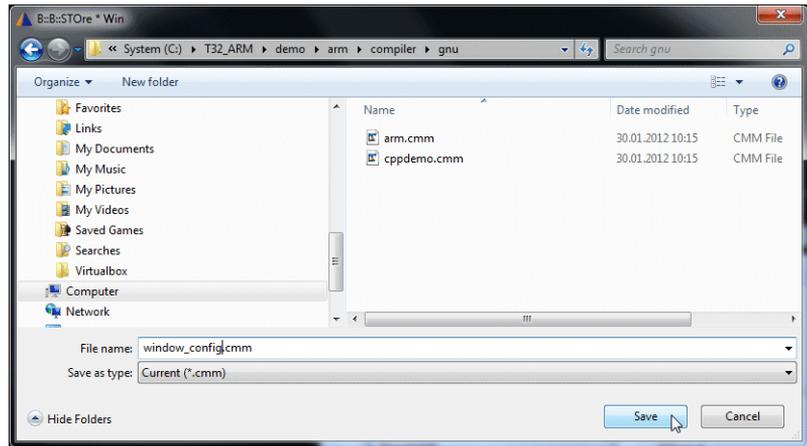
	<p>For more complex changes to the main tool bar refer to “Training Menu Programming” (training_menu.pdf).</p> <p>Videos about the menu programming can be found here: support.lauterbach.com/kb/articles/trace32-user-interface-customization</p>
---	---

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::

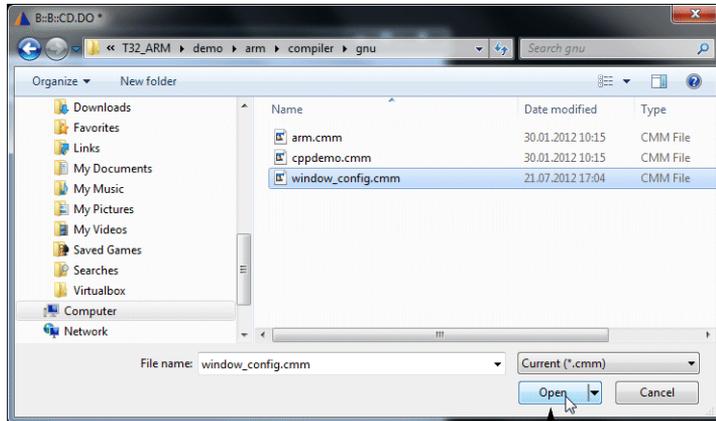
TOOLBAR ON
STATUSBAR ON
FramePOS 68.0 5.2857 107. 45.
WinPAGE.RESet

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

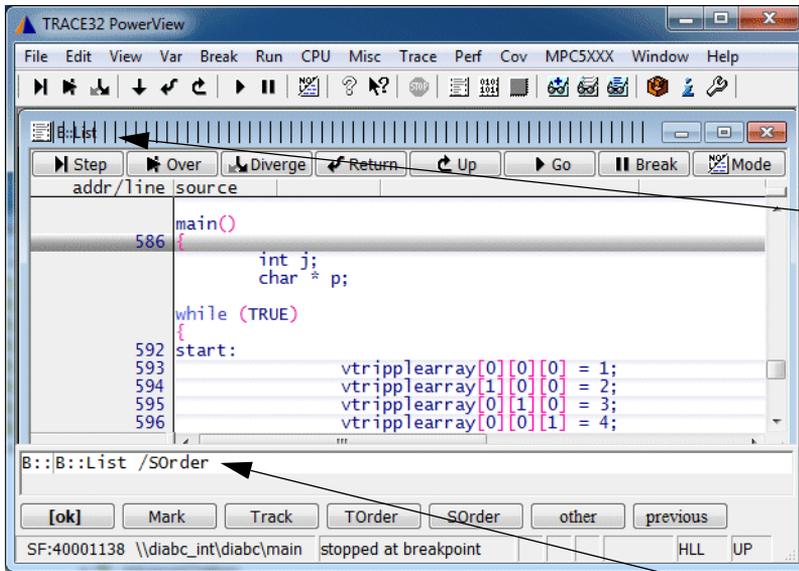
WinPOS 0.0 21.643 80. 5. 25. 1. W001
WinTABS 13. 0. 0. 0. 0. 0. 0.
Break.List

WinPAGE.select P000

ENDDO
```



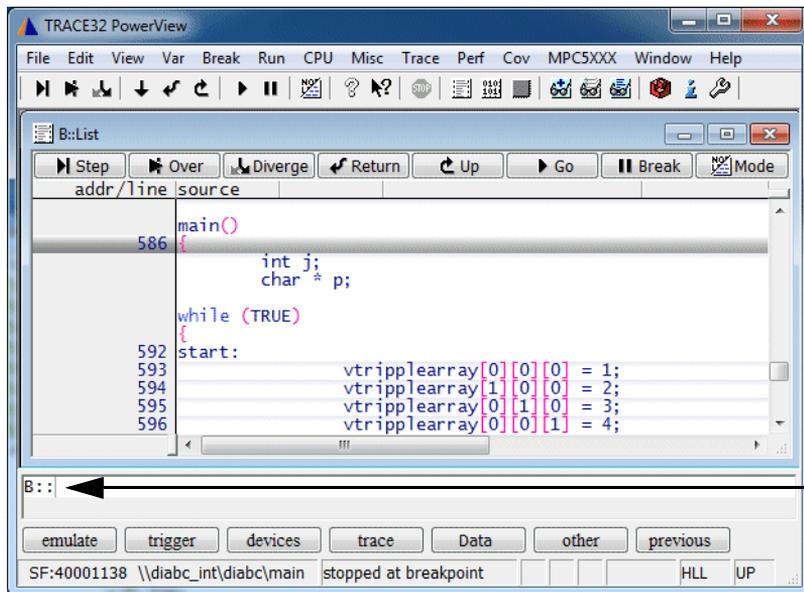
Run the script to reactivate the stored window-configuration



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 Line



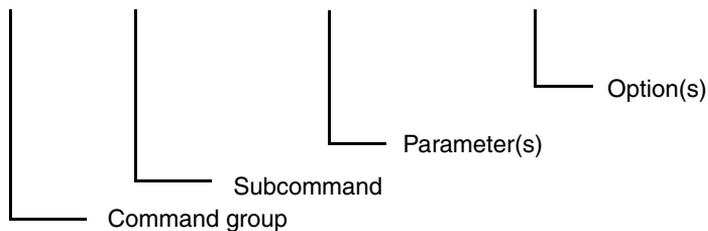
Command line

Command Structure

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

A TRACE32 command has the following structure:

```
Data.dump 0x1000--0x1fff /Byte
```



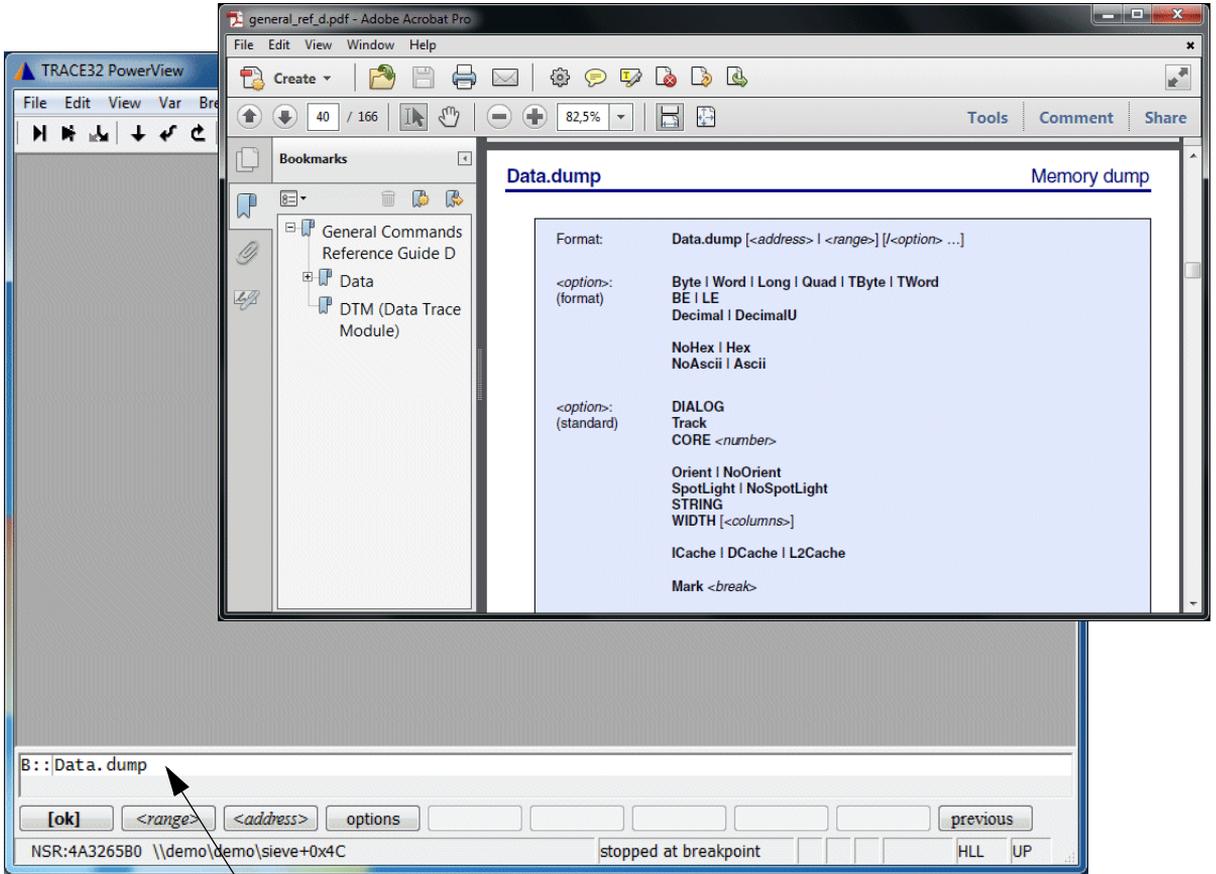
Command Examples

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

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

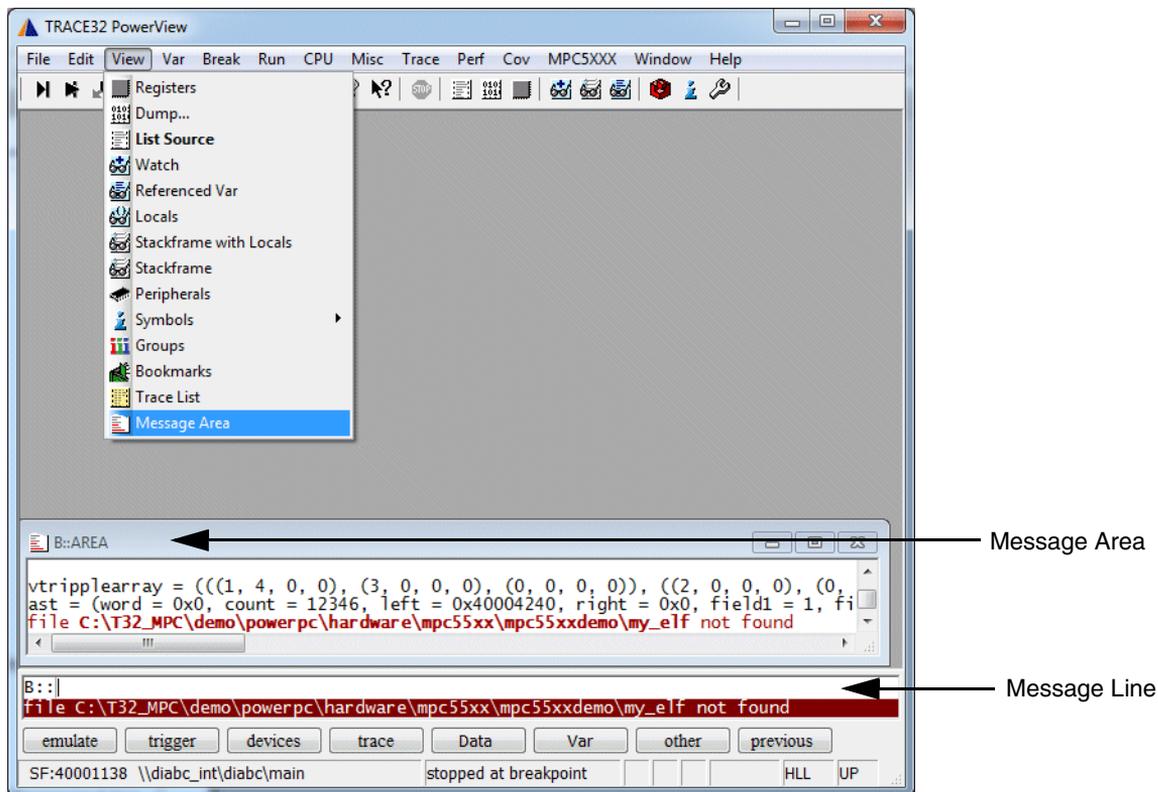
Each command can be abbreviated. The significant letters are always written in upper case letters.

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.

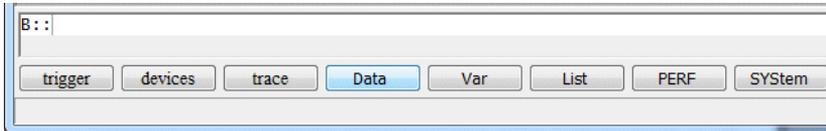
Message Line



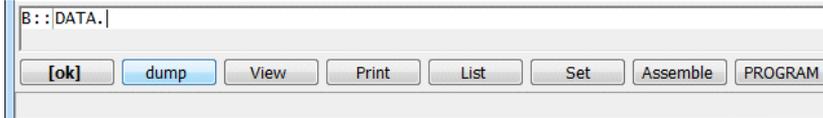
- **Message line** for system and error messages
- **Message Area window** for the display of the last system and error messages

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

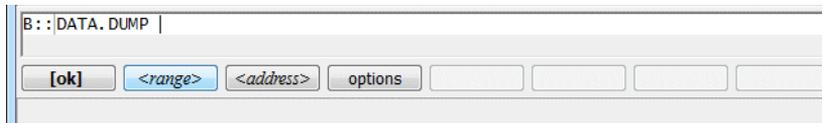
Select the command group, here **Data**.



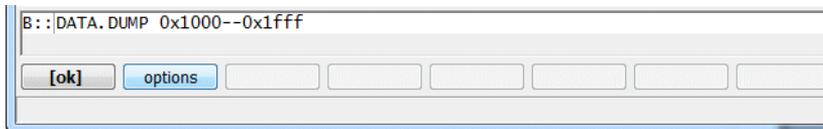
Select the subcommand, here **dump**.



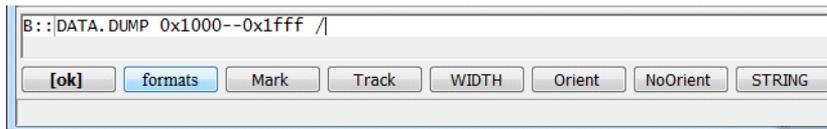
Angle brackets request an entry from the user, here e.g. the entry of a <range> or an <address>.



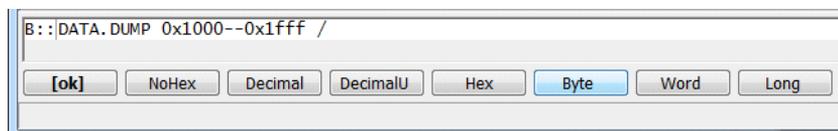
The display of the hex. dump can be adjusted to your needs by an option.



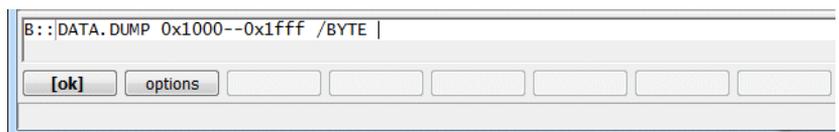
Select the option **formats** to get a list of all format options.

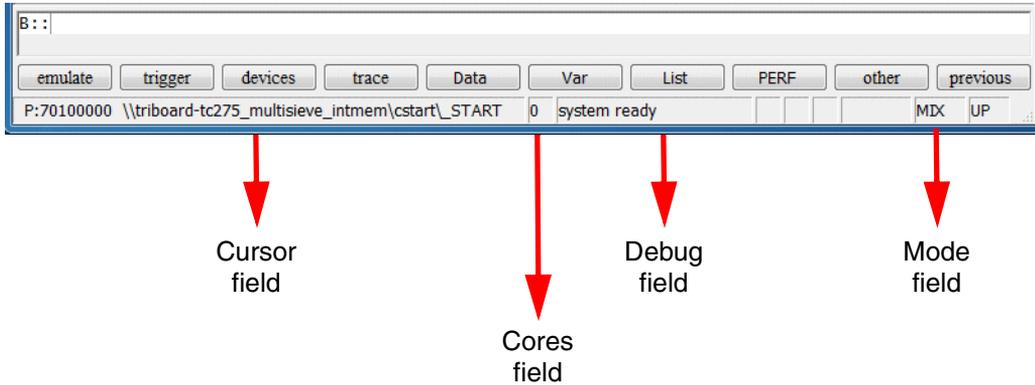


Select a format option, here **Byte**.



The command is complete now.





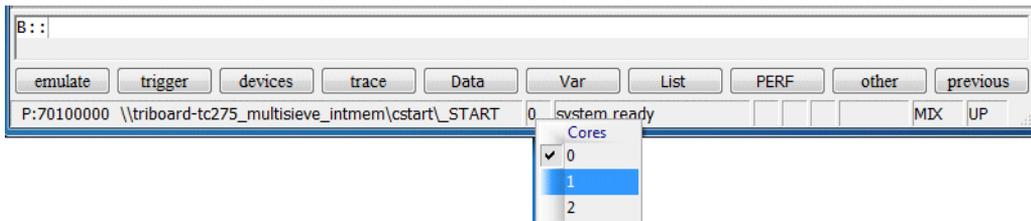
The **Cursor** field of the state line provides:

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

The **Cores** field shows the currently select core.

- TRACE32 PowerView visualizes all system information from the perspective of the selected core if not specified otherwise.

The **Cores** pull-down allows to change the selected core.



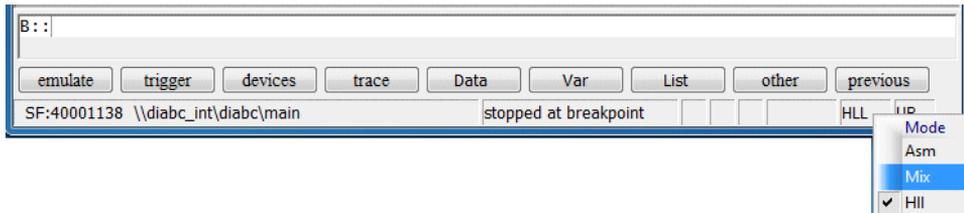
The **Debug** field of the state line provides:

- 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).

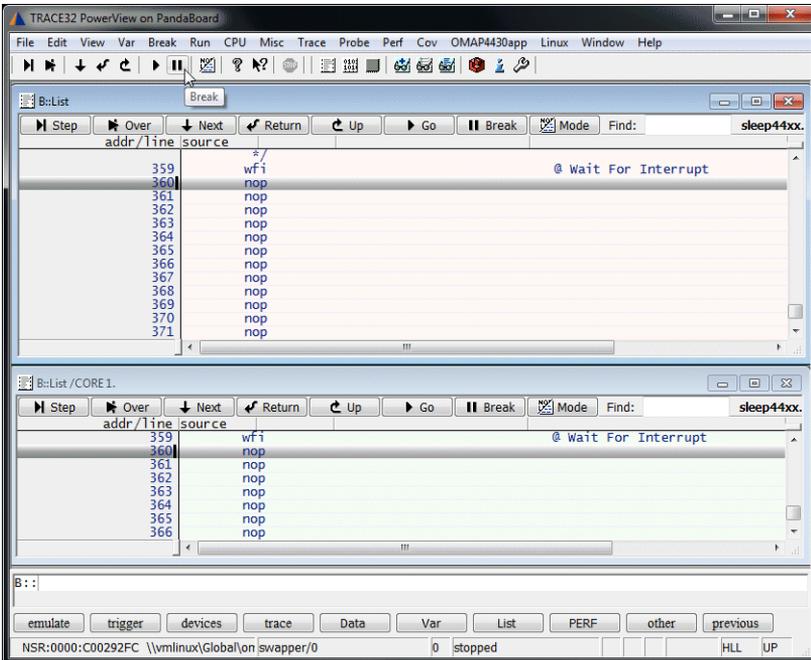
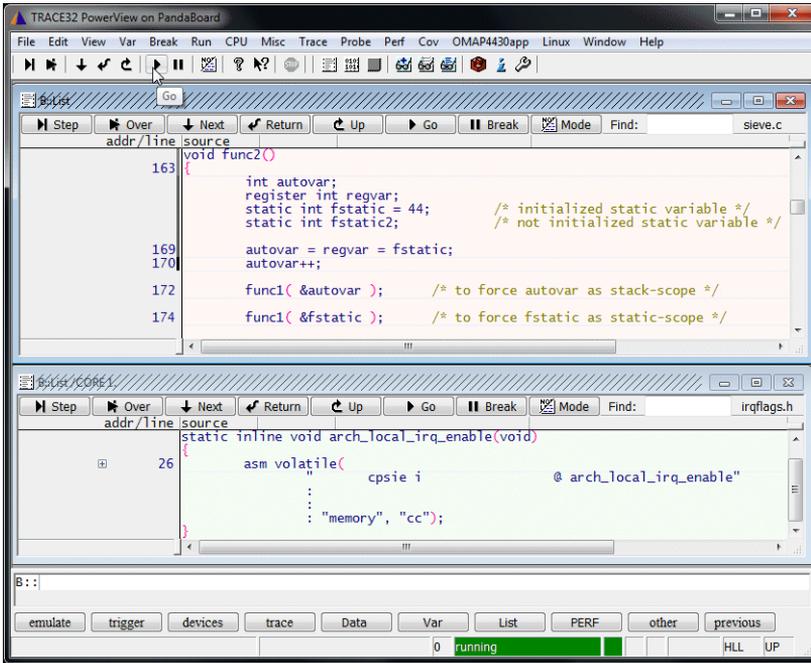


The debug mode can be changed by using the **Mode** pull-down.

Basic Debugging (SMP)

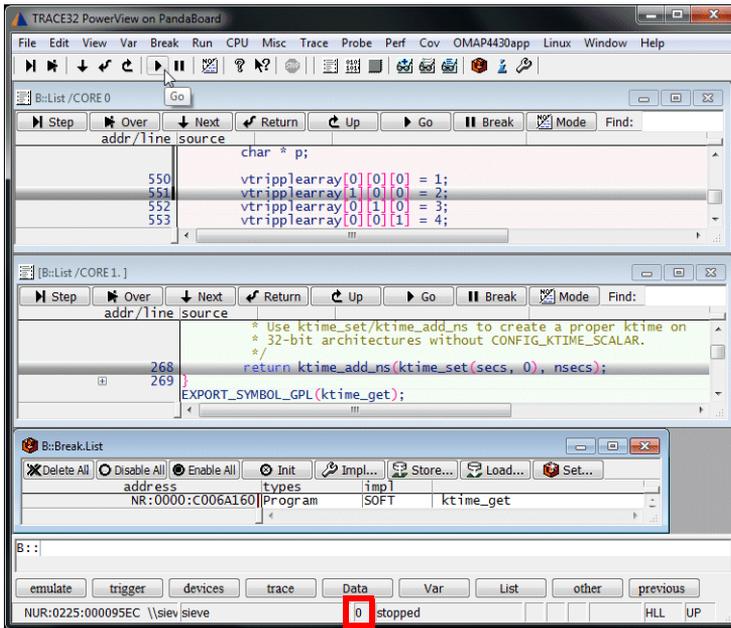
Go/Break

On an SMP systems the program execution on all cores is started with **Go** and stopped with **Break**.

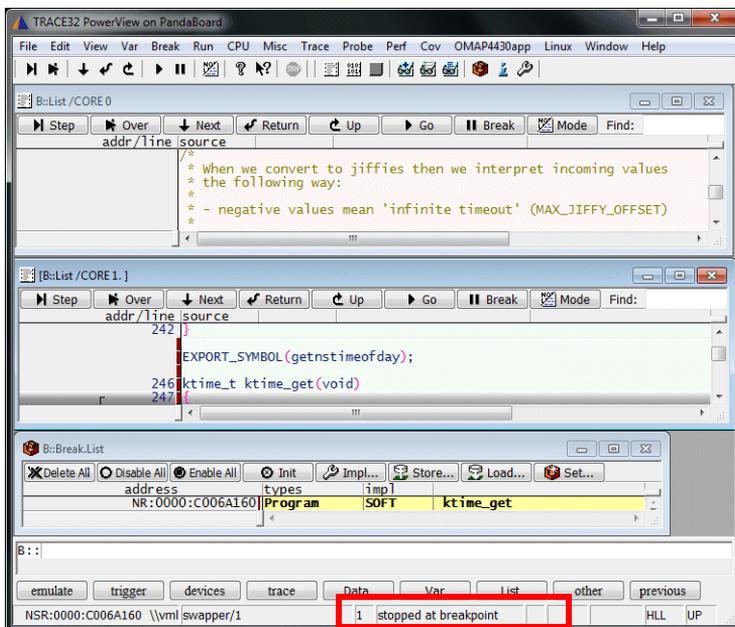


If a breakpoint is hit, TRACE32 makes the core the selected one on which the breakpoint occurred.

Not possible for all processor architectures (e.g. not possible for Aurix chips).



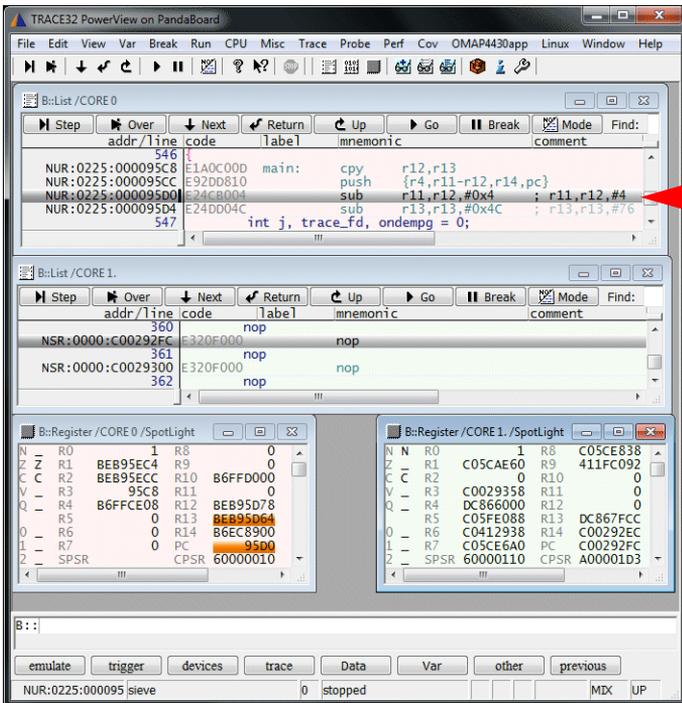
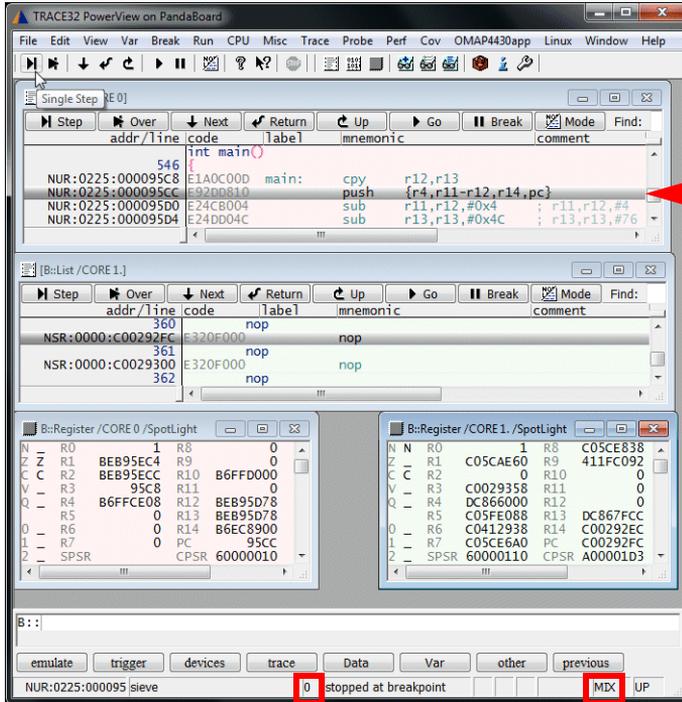
Core 0 was the selected one when the program execution started.



The breakpoint occurred on core 1. So **core 1** is the selected one after the program execution stopped.

Single Stepping on Assembler Level

Assembler single steps are only performed on the selected core.



Only the program counter of core 0 has changed

**Mode.Mix
Step**

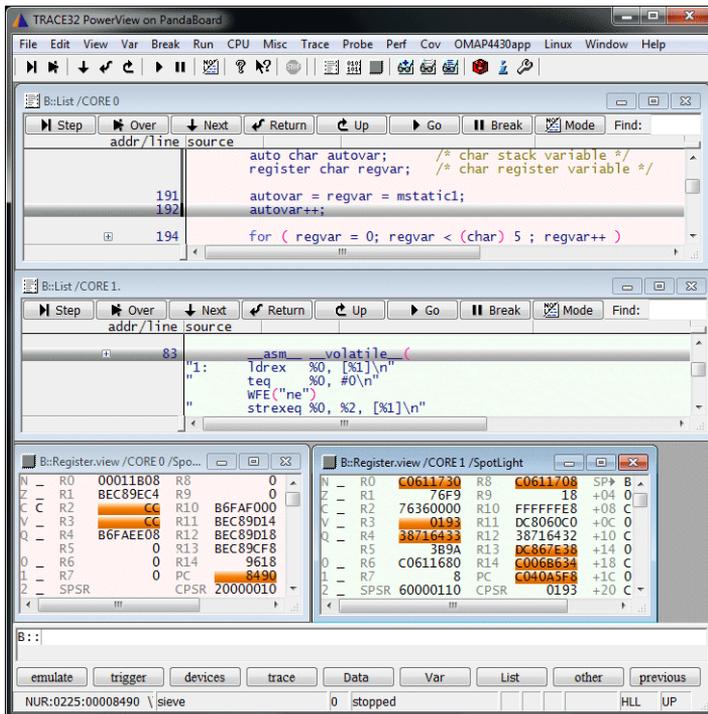
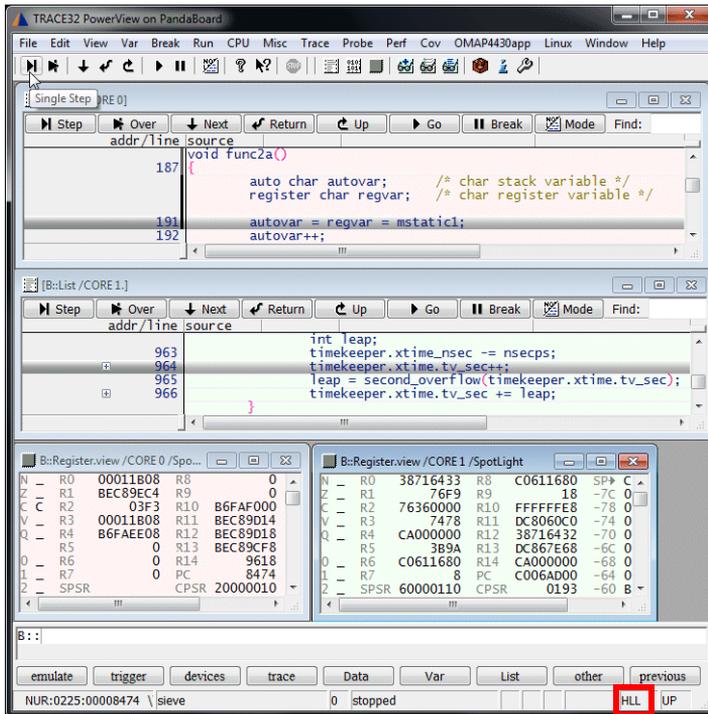
Select Mix mode for debugging and perform a single step on the selected core.

Step.Asm

Perform an assembler single step on the selected core.

Single Stepping on High-Level Language Level

An HLL single step is performed on the selected core. All other cores are started and will stop, when this HLL single step is done.



**Mode.Hll
Step**

Select High-level language mode for debugging and perform a single step.

Step.Hll

Perform an HLL single step.

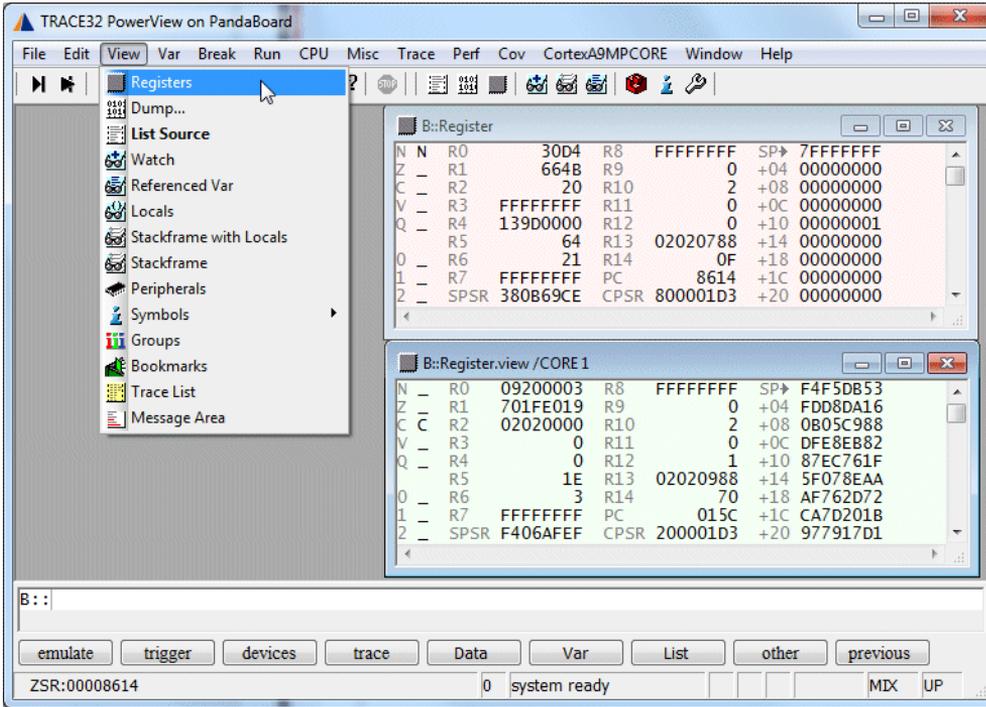
SETUP.Step.WithinTASK ON

When ON all HLL stepping is performed only in the currently active task.

Registers

Core Registers

Display the Core Registers



The core register contents is core-specific information. It is printed on a colored background.

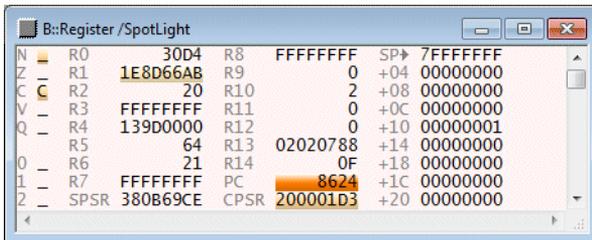
Please be aware that all menus and buttons apply to the currently selected core.

```
Register.view ; display core register contents of  
              ; currently selected core  
              ; (here core 0)  
  
Register.view /CORE 1. ; display core register contents of  
                       ; core 1
```

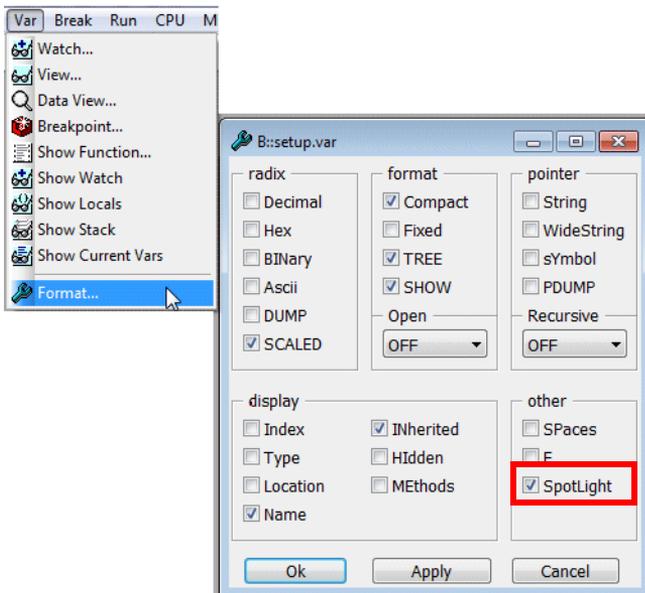
Colored Display of Changed Registers

The option /SpotLight advises TRACE32 PowerView to mark changes.

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



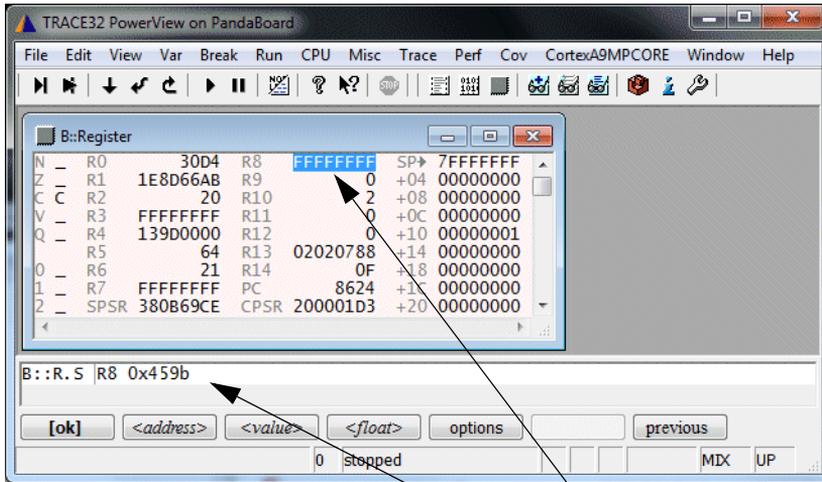
Establish /SpotLight as default setting



SETUP.Var %SpotLight

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

Modify the Contents of a Core Register



By double clicking to the register contents a **Register.Set** command is automatically displayed in the command line. Enter the new value and press Return to modify the register contents.

Register.Set <register> <value>

Modify core register of selected core

Register.Set <register> <value> /CORE <n>

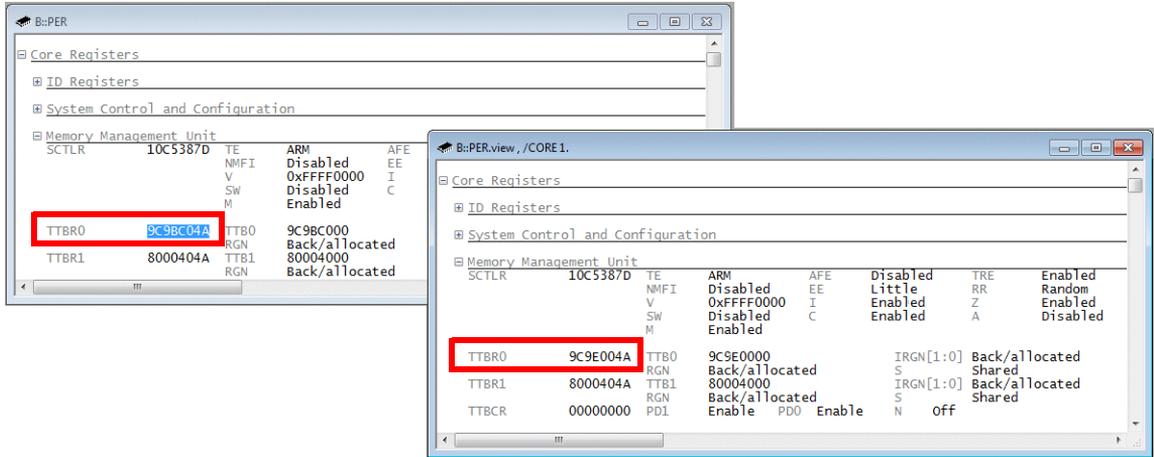
Modify core register of specified core

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.

In an SMP system all cores have equal rights to use configuration registers, external interfaces and external devices. So TRACE32 PowerView regards all these registers as common resources and thus displayed them on a white background.

But not all configuration registers are common resources: Exceptions are the core-related registers e.g. CPUIDs, MMU translation tables ...



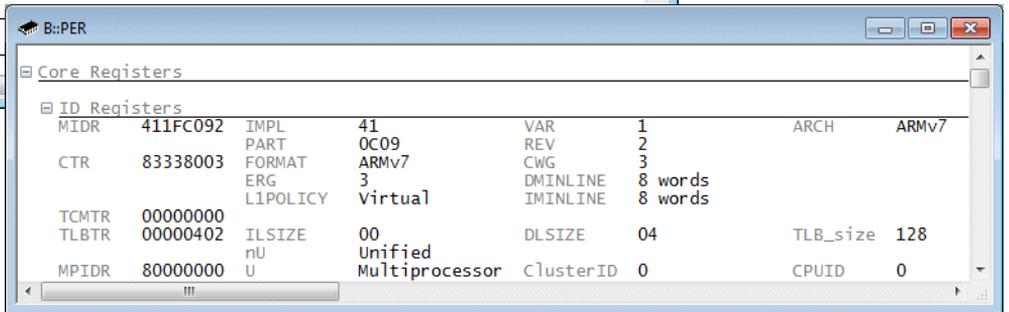
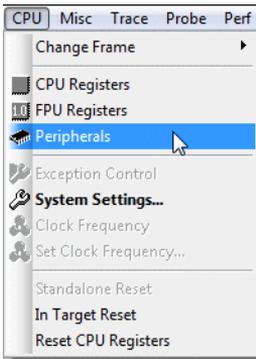
Translation Table Base Register 0 contains a different contents on each core

TRACE32 PowerView provides the **/CORE <n>** option in order to display details on core-related configuration registers:

```
PER.view , /CORE 1.
```

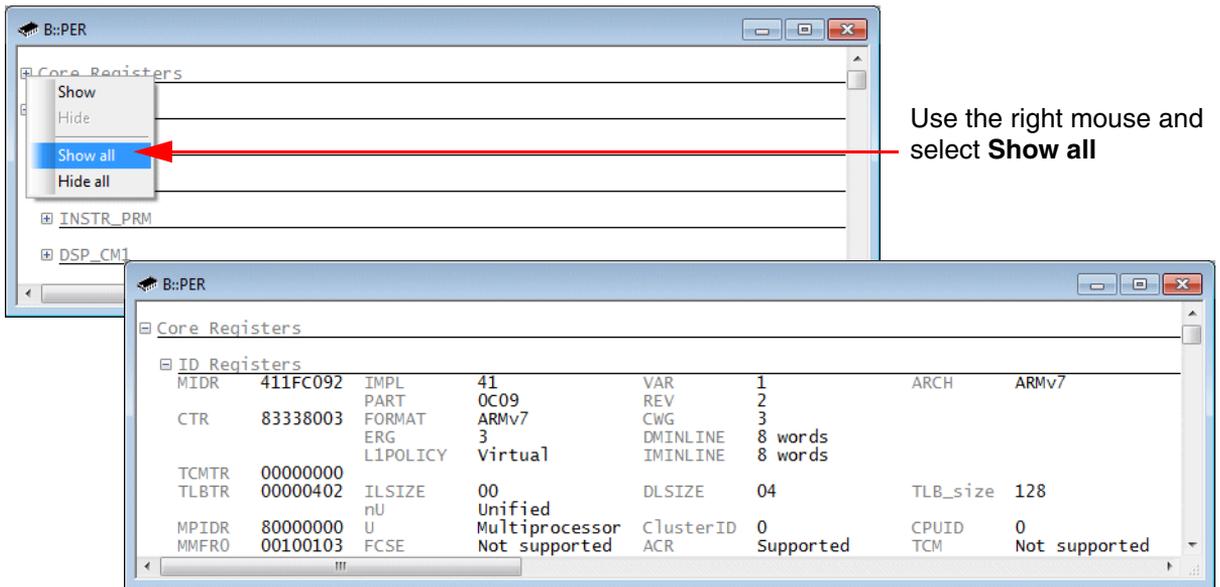
Tree Display

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

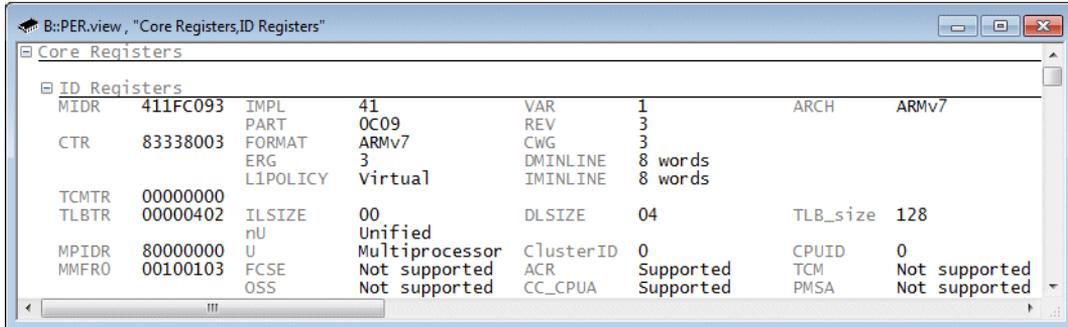
```
; Display all functional units in expanded mode
; , advises TRACE32 PowerView to use the default peripheral file
; * stands for all <tree-items>
PER.view , "*"

```

```

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

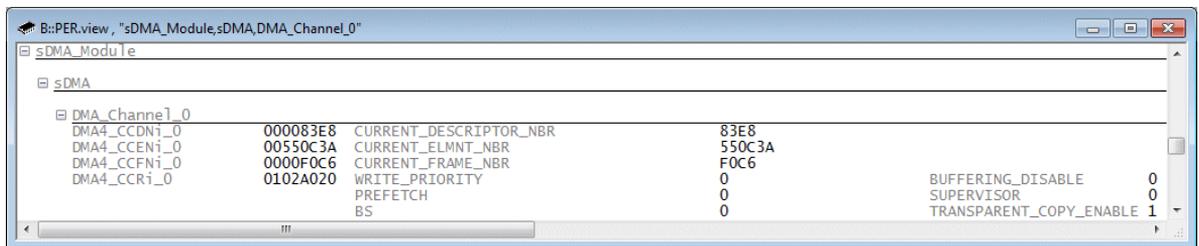
```



```

; 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.

```

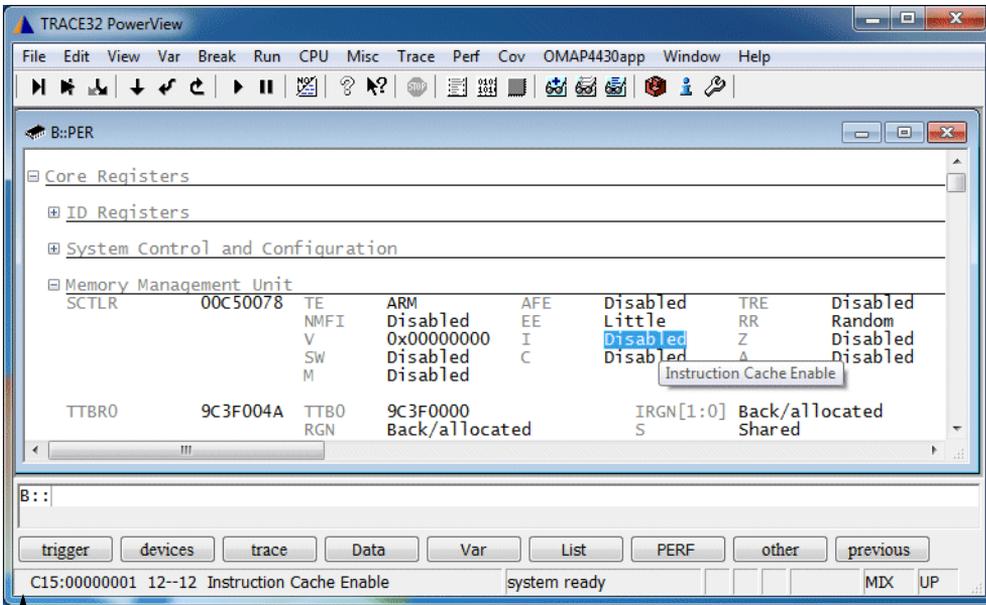
; PRinTer.FileType ASCIIIE ; Select ASCII ENHANCED as output
; format
; (default output format)

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

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

```

Details about a Single Special Function Register

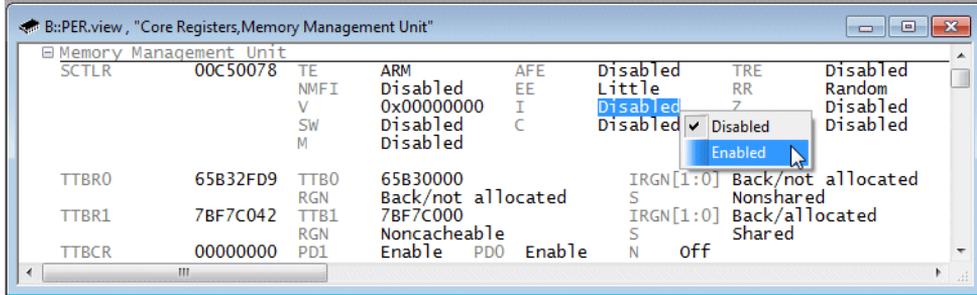


The access class, address, bit position and the full name of the selected item are displayed in the state line; the full name of the selected item is taken from the processor/chip manual.

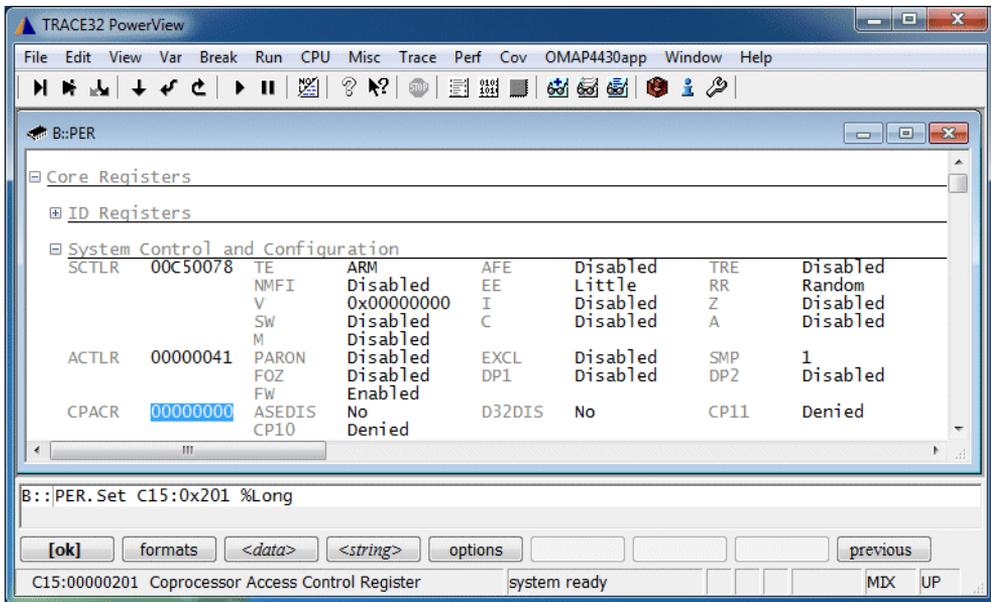
Modify a Special Function Register

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

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



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



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

Modify configuration register/on-chip peripheral

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

Modify memory

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

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

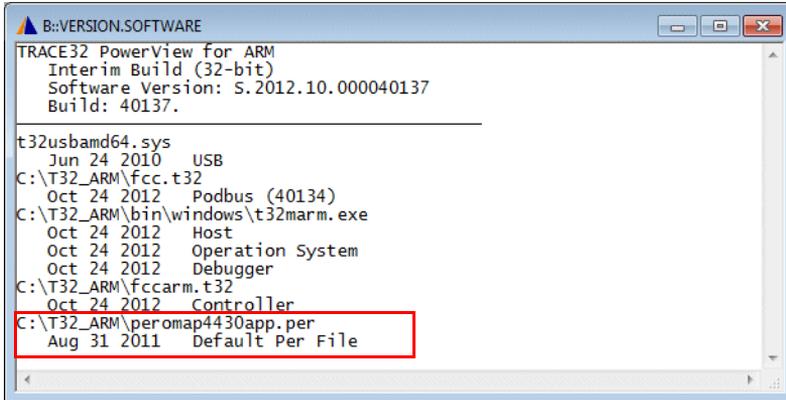
The PER Definition File

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

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

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

VERSION.SOFTWARE



PER.view <filename>

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

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

Memory Display and Modification

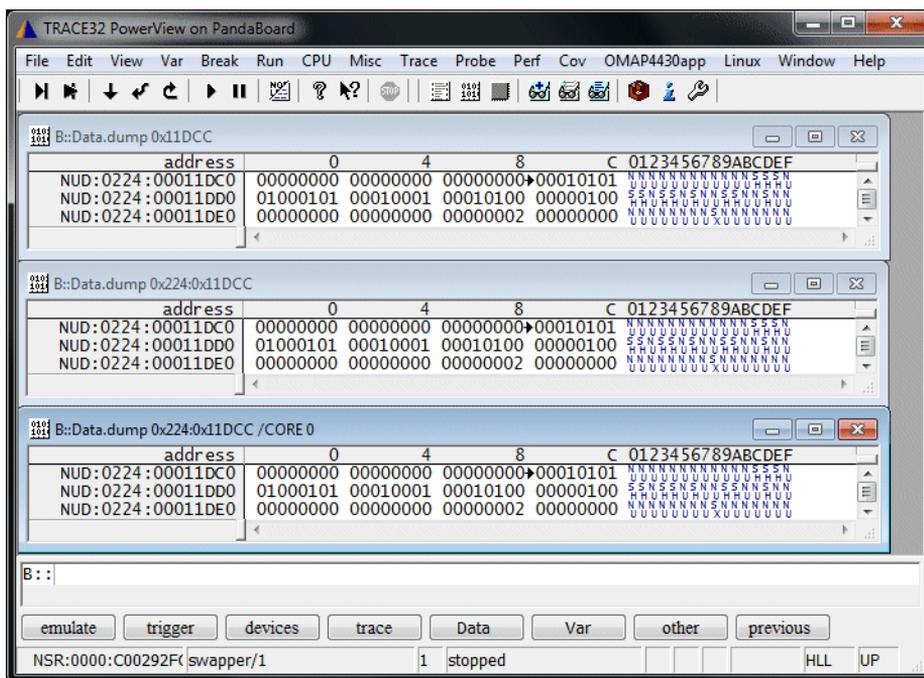
This training section introduces the most often used methods to display and modify memory:

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

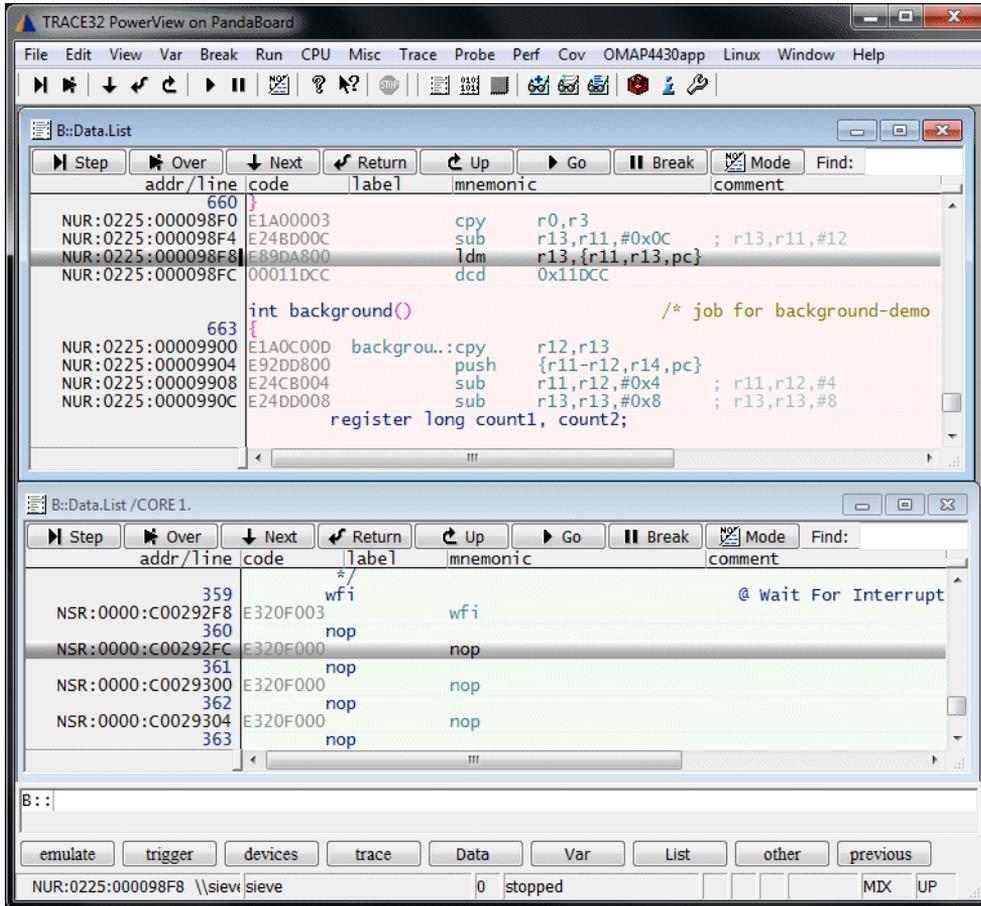
Shared memory is a characteristics of an SMP system. This is the reason why the **Data.dump** window is regarded as common information and is displayed therefore on a white background. TRACE32 PowerView assumes that cache coherency is maintained in an SMP system.

Cache coherency: In a shared memory with a separate cache for each core, it is possible to have many copies of one data: one copy in the main memory and one in each cache. When one copy of this data is changed, the other copies of the data must be changed also. Cache coherence ensures that changes in the values of a data are propagated throughout the system.

To provide flexibility the **CORE <n>** option is provided also for the **Data.dump** command.



Since the **List** (former **Data.List**) window is mainly used to display a source code listing around the current program counter it is regarded as core-specific information and is therefore displayed on a colored background.



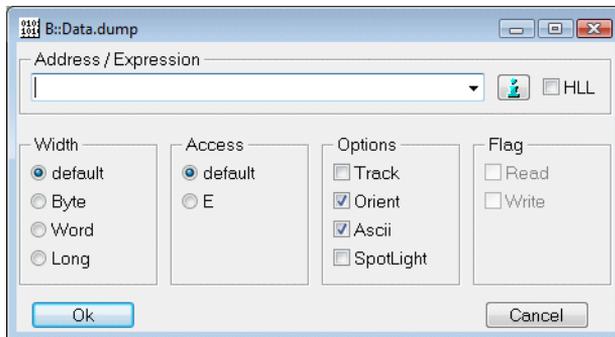
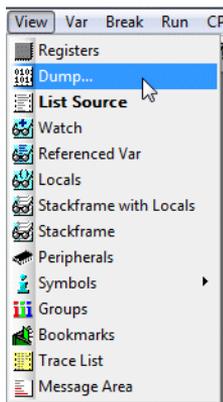
A so-called **access class** is always displayed together with a memory address. The following access classes are available for all processor architectures:

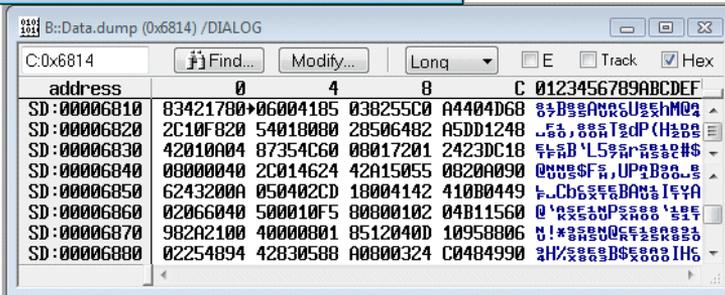
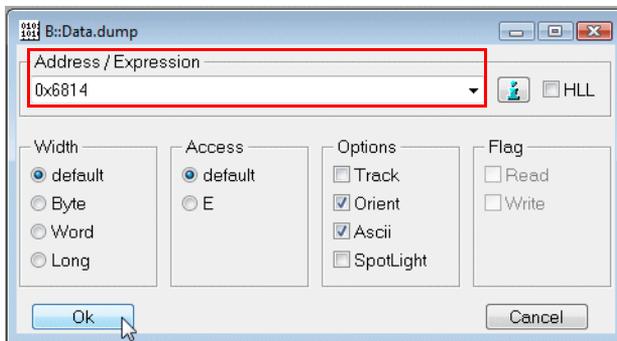
P:1000	Program address 0x1000
D:6814	Data address 0x6814

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

The Data.dump Window

Display the Memory Contents



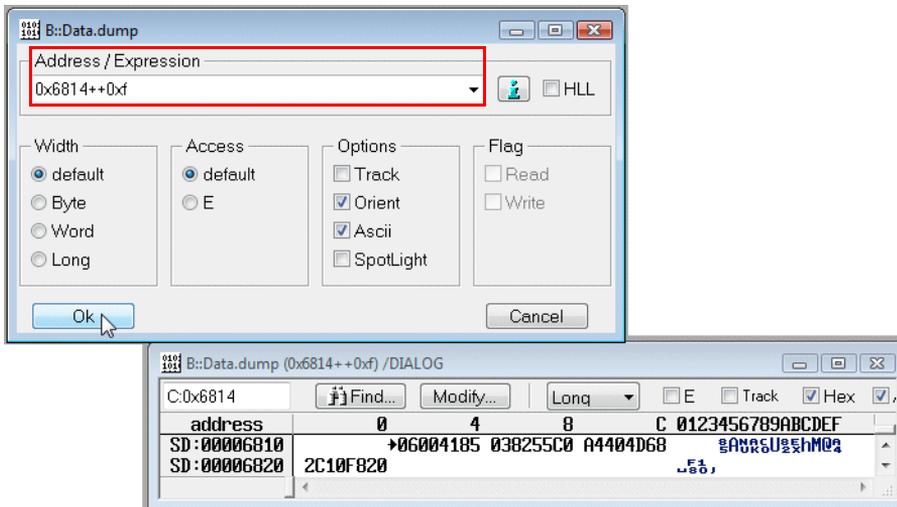


Please be aware, that TRACE32 permanently updates all windows. The default update rate is 10 times per second.

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.

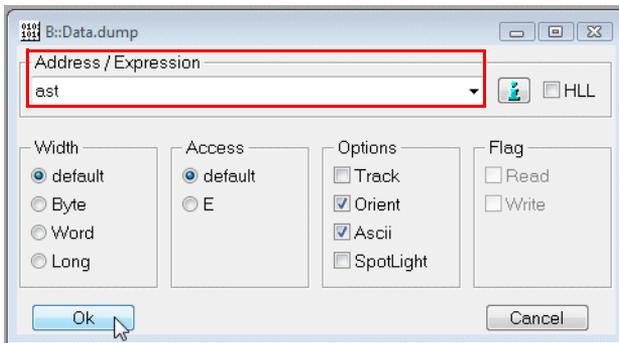
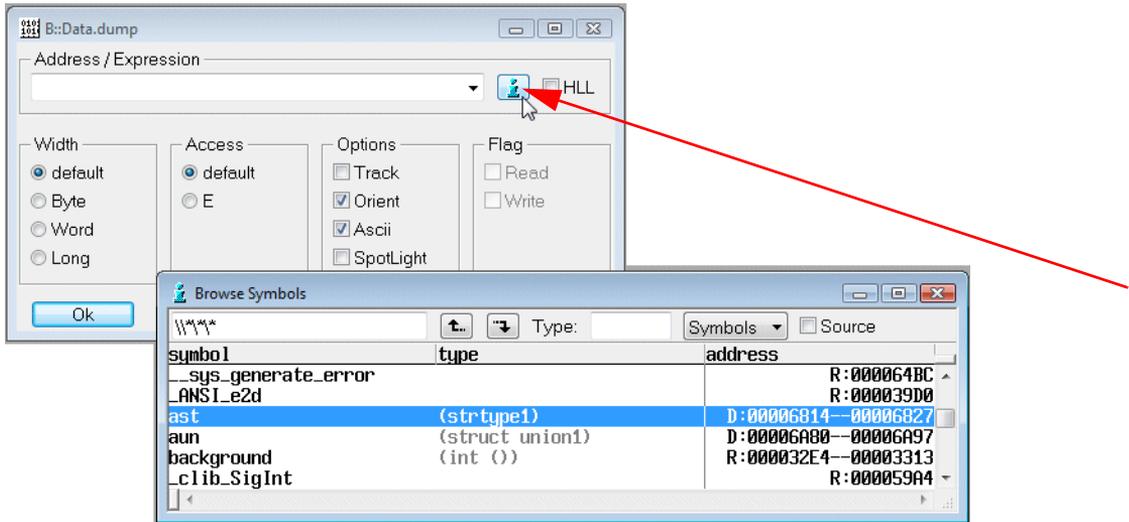
Conventions for address ranges:

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



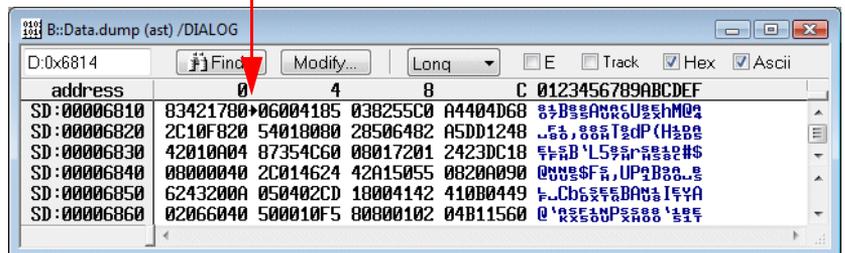
Use a Symbol to Specify the Start Address for the Data.dump Window

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



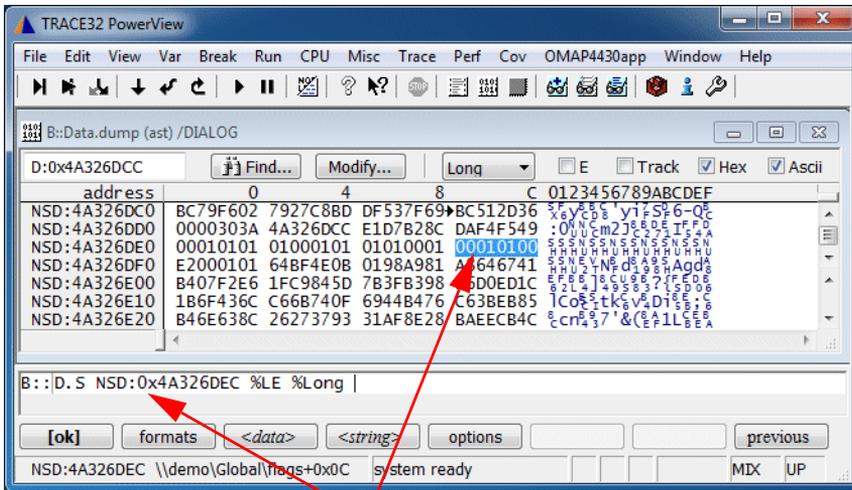
By default an oriented display is used (line break at 2^x).

A small arrow indicates the specified dump address.



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

Modify the Memory Contents



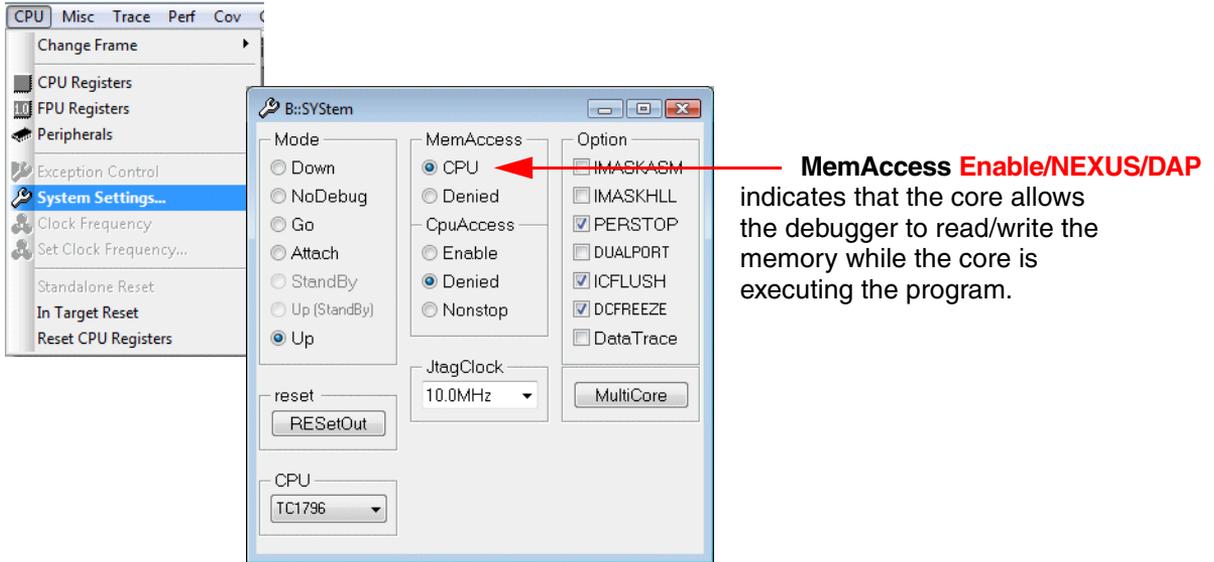
By a left mouse double-click to the memory contents
a **Data.Set** command is automatically
displayed in the command line,
you can enter the new value and
confirm it with return.

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

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


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:



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.

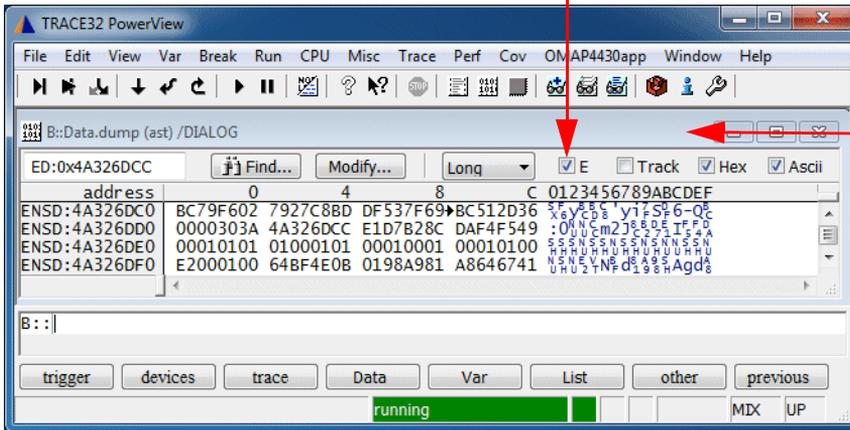
Usage

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:

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

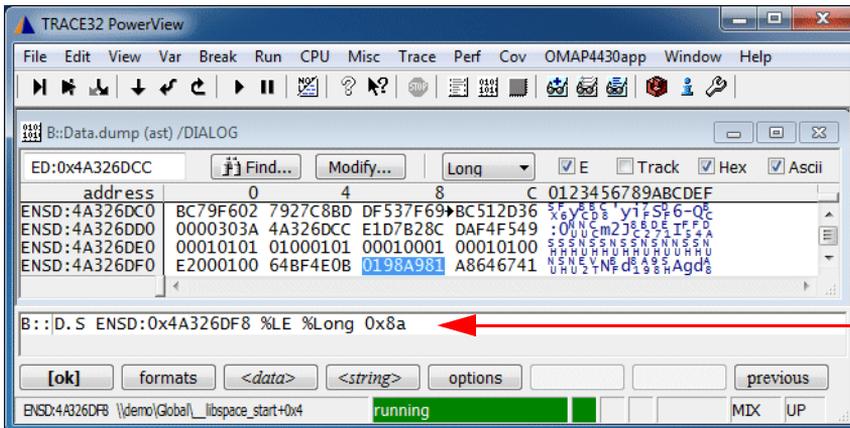


A plain window frame indicates that the information is updated while the core is executing the program

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:



Data.Set via run-time memory access (attribute E)

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

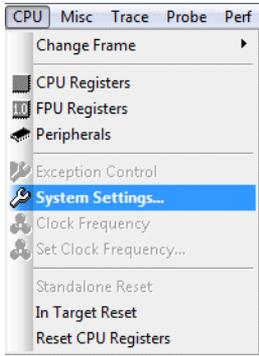
;...

Go                                 ; Start program execution

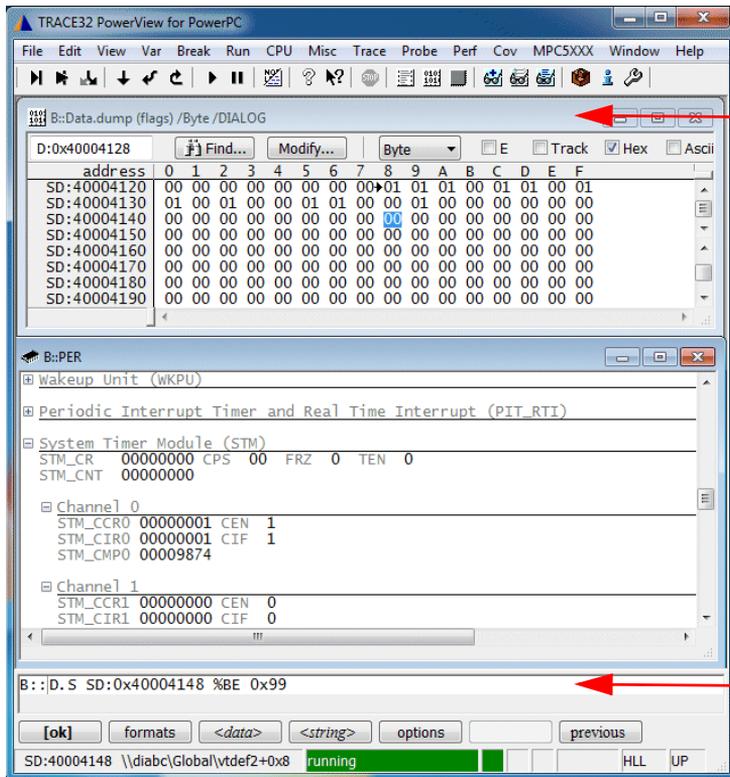
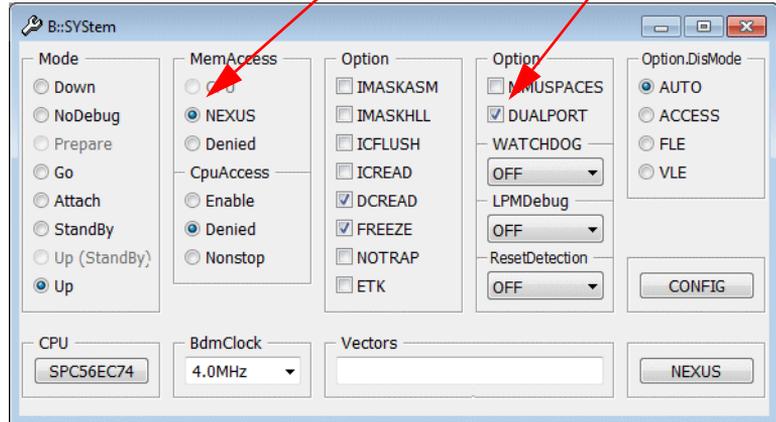
Data.dump E:0x6814                 ; Display a hex dump starting at
                                   ; address 0x6814 via run-time
                                   ; memory access

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

; ...

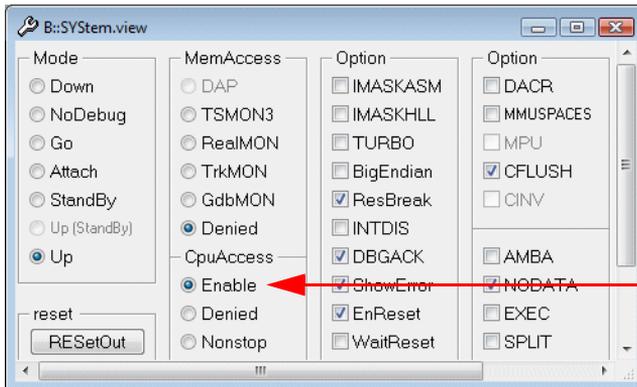
Go                                 ; Start program execution

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
```

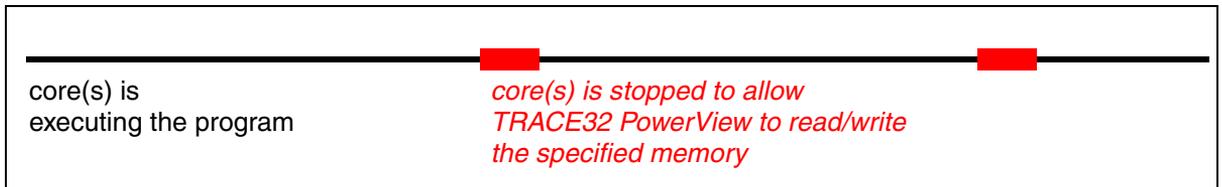
Intrusive Run-Time Memory Access

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**.

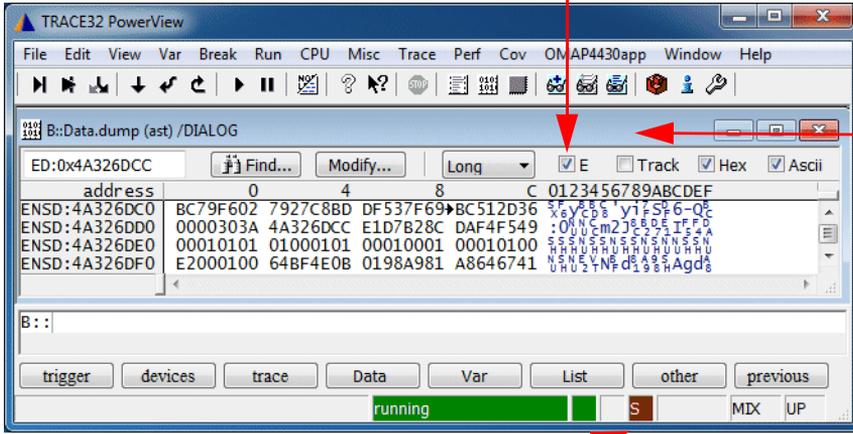


The time taken by a short stop depends on various factors:

- 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 accesses 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**.

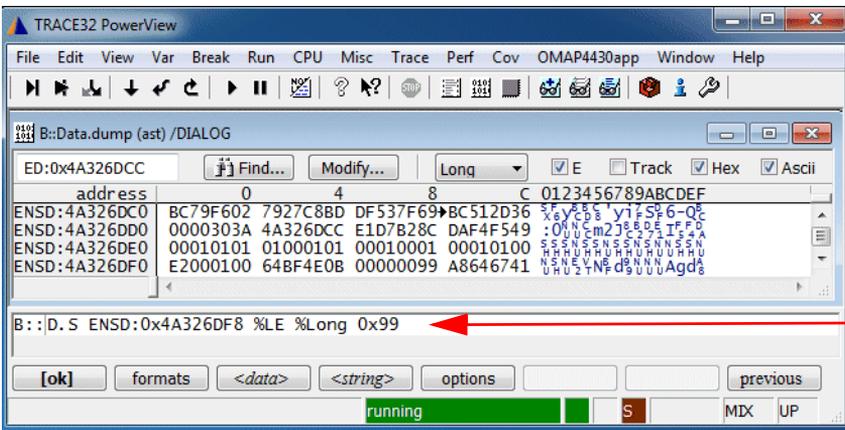
Enable the **E** check box to switch the run-time memory access to ON



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

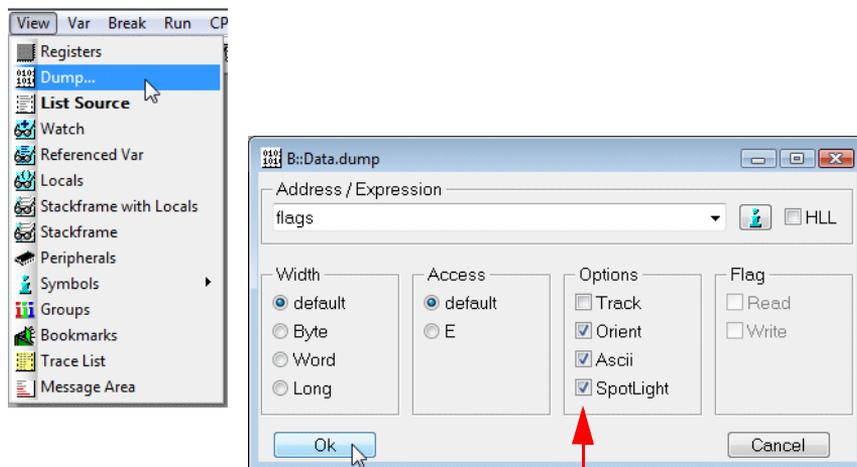
;...

Go                                 ; Start program execution

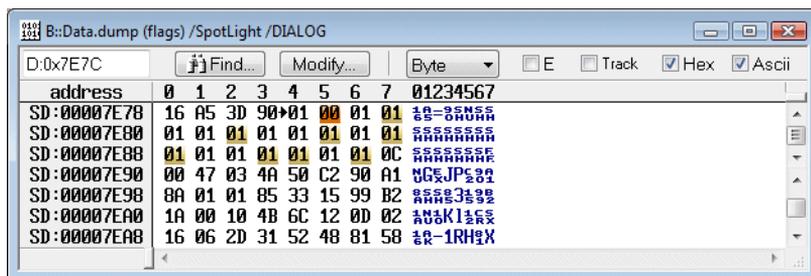
Data.dump E:0x6814                 ; Display a hex dump starting at
                                   ; address 0x6814 via an intrusive
                                   ; run-time memory access

Data.Set E:0x6814 0xAA             ; Write 0xAA to the address
                                   ; 0x6814 via an intrusive
                                   ; run-time memory access
```

Colored Display of Changed Memory Contents



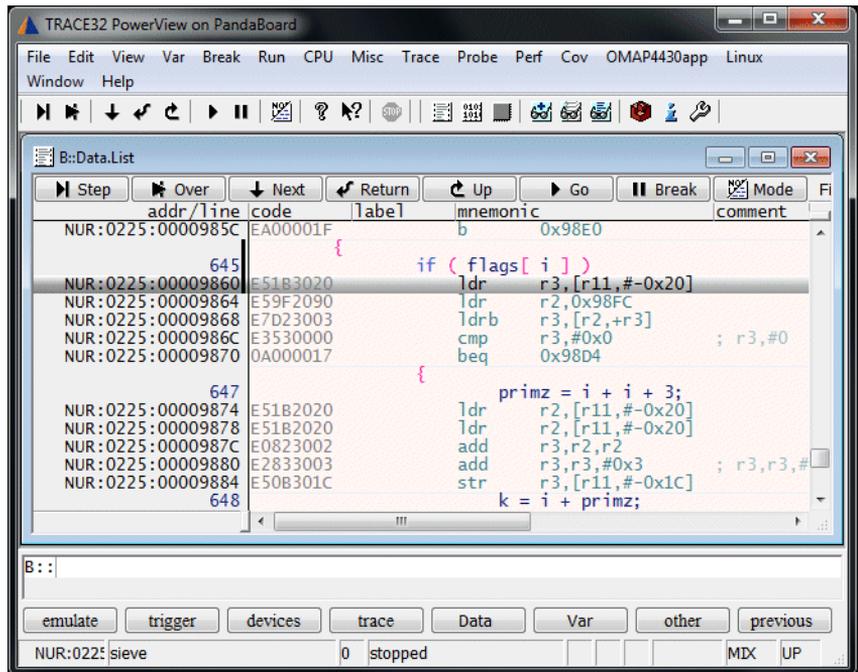
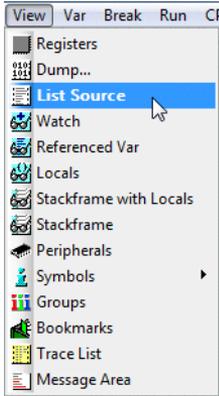
Enable the option **SpotLight** to mark the memory contents changed by the last 4 single steps in orange, older changes being lighter.



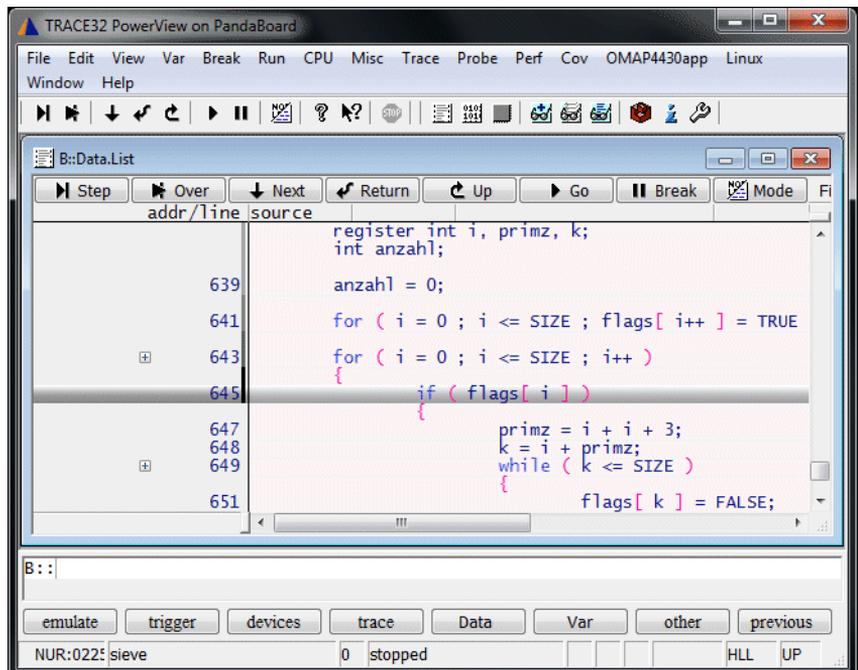
```
Data.dump flags /SpotLight ; Display a hex dump starting at ; the address of the label flags ; Mark changes
```

The List Window

Displays the Source Listing Around the PC

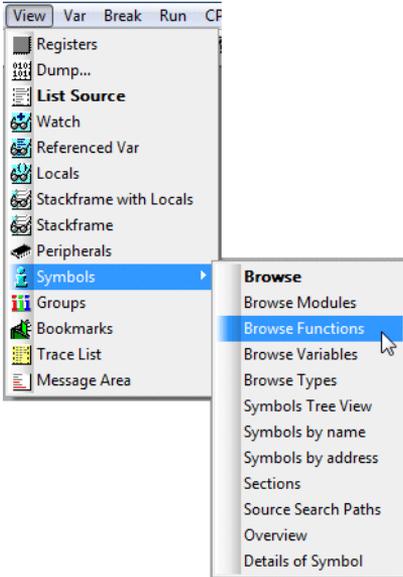


If **MIX** mode is selected for debugging, assembler and HLL information is displayed

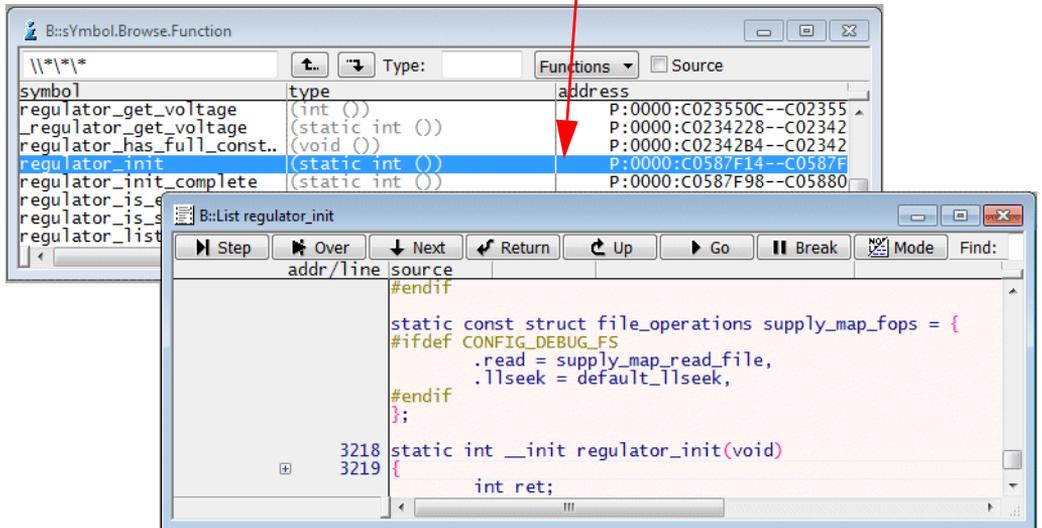


If **HLL** mode is selected for debugging, only hll information is displayed

Displays the Source Listing of a Selected Function



Select the function you want to display



- | | |
|--|------------------------|
| List [<i><address></i>] [<i>/<option></i>] | Display source listing |
| List [<i><address></i>] /CORE <i><n></i> [<i>/<option></i>] | Display source listing |
| Data.List [<i><address></i>] [<i>/<option></i>] | Display source listing |
| Data.List [<i><address></i>] /CORE <i><n></i> [<i>/<option></i>] | 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
```

```
List /CORE 1 ; Display a source listing
; around the PC of core 1
```

Breakpoints

Videos about the breakpoint handling can be found here:

support.lauterbach.com/kb/articles/using-breakpoints-in-trace32

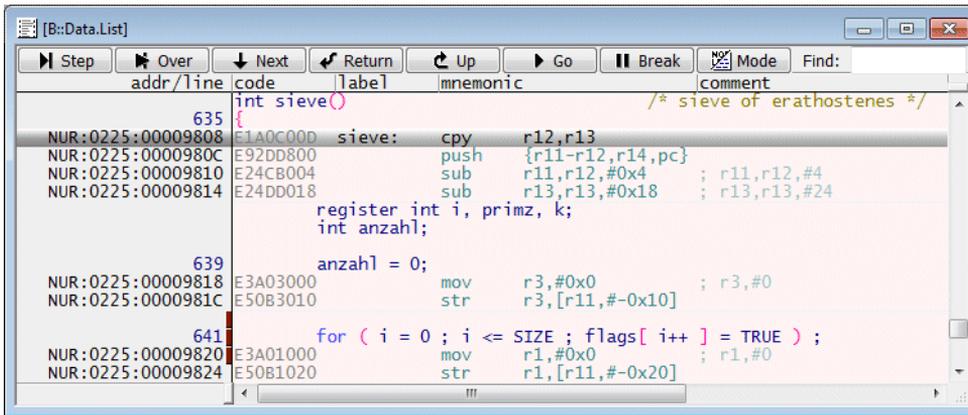
Breakpoint Implementations

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

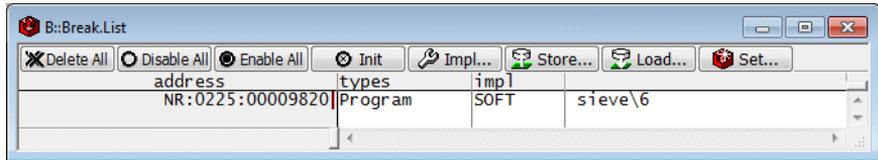
Software Breakpoints in RAM

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.



The screenshot shows a window titled "[B::Data.List]" with a menu bar containing Step, Over, Next, Return, Up, Go, Break, Mode, and Find. The main area displays assembly code for a function named "int sieve()". The code includes instructions like "cpy r12,r13", "push {r11-r12,r14,pc}", "sub r11,r12,#0x4", "sub r13,r13,#0x18", "mov r3,#0x0", "str r3,[r11,#-0x10]", "mov r1,#0x0", and "str r1,[r11,#-0x20]". Comments indicate the function is a sieve of Eratosthenes. Line numbers 635, 639, and 641 are visible on the left.



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

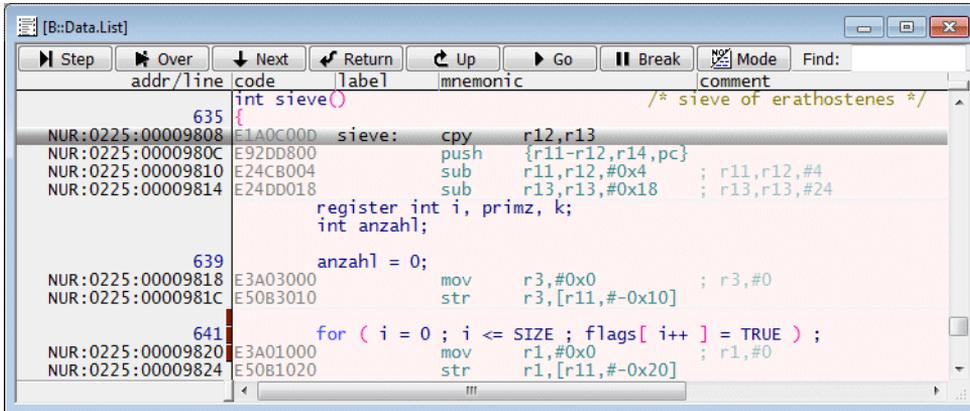
	<p>Please be aware that TRACE32 PowerView always tries to set an Onchip breakpoint, when the setting of a Software Breakpoint fails.</p>
---	--

Onchip Breakpoints in NOR Flash

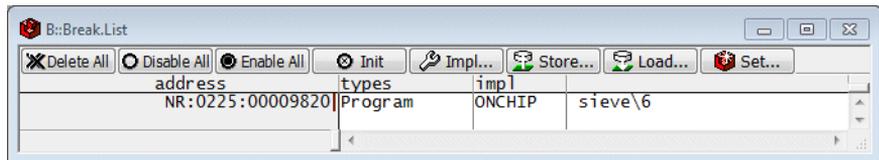
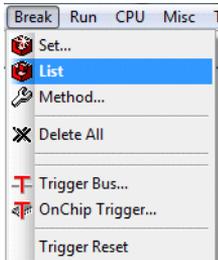
Most core(s) provide a small number of Onchip breakpoints in form of breakpoint registers. These Onchip breakpoints can be used to set breakpoints to instructions in read-only memory like onchip or NOR FLASH.

That fact that the debugger does not know on which core of the SMP system a program section is running, has the consequence that the debugger programs the same on-chip breakpoint to all cores.

So you can say from the debugger perspective there is only one break logic shared by all cores of the SMP system. This is the reason why breakpoints are regarded as common resource and therefore the **Break.List** window has a white background.



```
[B::Data.List]
Step Over Next Return Up Go Break Mode Find:
addr/line code label mnemonic comment
635 { /* sieve of erathostenes */
NUR:0225:00009808 E1A0C00D sieve: cpy r12,r13
NUR:0225:0000980C E92DD800 push {r11-r12,r14,pc}
NUR:0225:00009810 E24CB004 sub r11,r12,#0x4 ; r11,r12,#4
NUR:0225:00009814 E24DD018 sub r13,r13,#0x18 ; r13,r13,#24
register int i, primz, k;
int anzahl;
anzahl = 0;
NUR:0225:00009818 E3A03000 mov r3,#0x0 ; r3,#0
NUR:0225:0000981C E50B3010 str r3,[r11,#-0x10]
641 for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
NUR:0225:00009820 E3A01000 mov r1,#0x0 ; r1,#0
NUR:0225:00009824 E50B1020 str r1,[r11,#-0x20]
```



address	types	impl	
NR:0225:00009820	Program	ONCHIP	sieve\6

If an SMP operating system that uses dynamic memory management to handle processes/tasks (e.g. Linux) is used, the instruction address within TRACE32 PowerView consists of:

- An access class
- A memory-space ID of the process
- A virtual address

```
<access_class>:<space_id>:<virtual_address>  
NUR: 0x225:0x9820
```

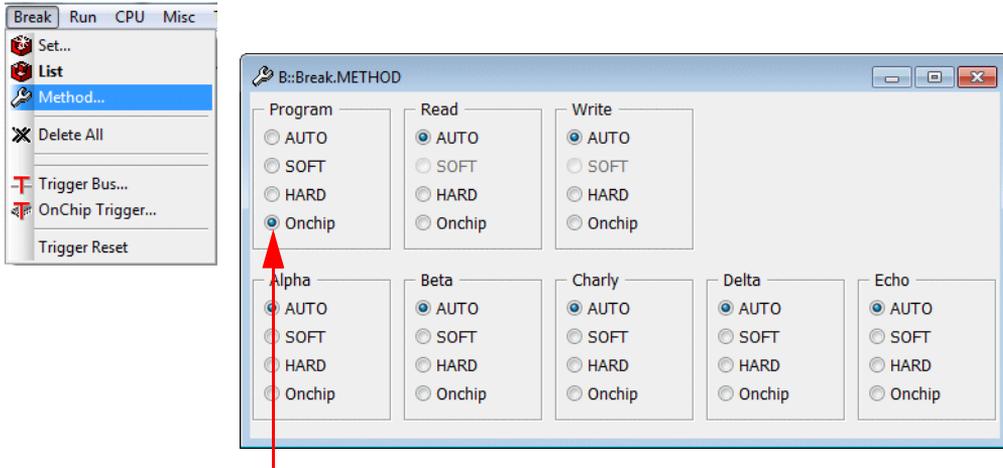
The on-chip break logic of most cores stores only the virtual address, but not the space ID. As a result an identical virtual address within another process can also result in a breakpoint hit.

For details on the TRACE32 PowerView address scheme of operating systems that uses dynamic memory management to handle processes/tasks refer to your **RTOS/OS Debugger Manual**.

Additional details on this issue are provided when task-aware breakpoints are introduced.

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

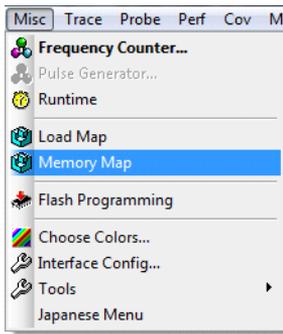
MAP.List

Check your settings

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

```
MAP.BOnchip 0xA0000000++0x1FFFFFFF
```

Check your settings as follows:



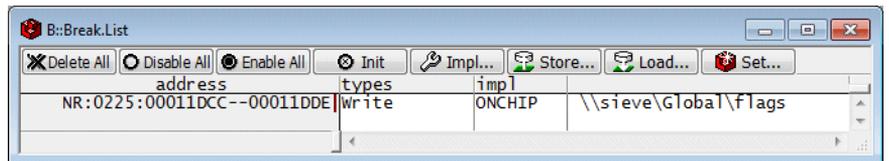
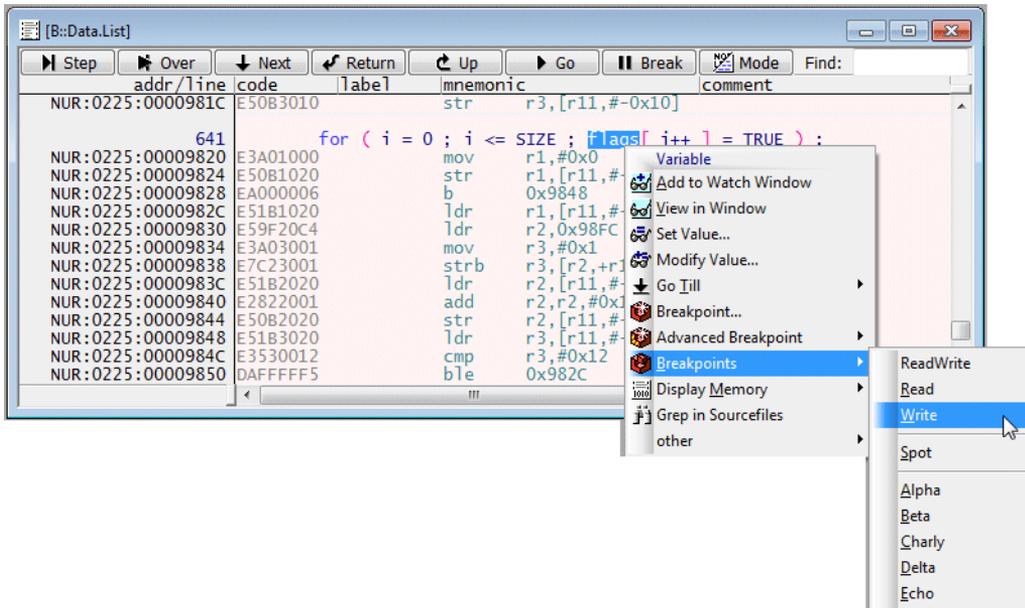
For the specified address ranges Program breakpoints are implemented as Onchip breakpoints. For all other memory areas Software breakpoints are used.

A screenshot of the 'B::Map.List' window in TRACE32. It displays a table with columns for 'address', 'type', 'bus', 'bonchip', and 'attributes'. A red arrow points to the 'bonchip' column. The table contains four rows of memory ranges, all with 'bonchip' in the 'bonchip' column.

address	type	bus	bonchip	attributes
A:00000000--0001FFF			bonchip	
A:00002000--9FFFFFFF				
A:A0000000--A01FFFFF			bonchip	
A:A0200000--FFFFFFF				

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.



Again, this breakpoint is programmed identically in all cores. And again write accesses to an identical virtual address result in a breakpoint hit.

Additional details on this issue are provided when task-aware breakpoints are introduced.

Onchip Breakpoints by Processor Architecture

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

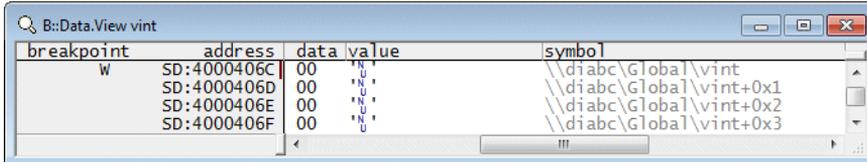
For some processor architectures Onchip breakpoints can only mark **single addresses** (e.g Cortex-A9). Most processor architectures, however, allow to mark **address ranges** with Onchip breakpoints. It is very common that one Onchip breakpoint marks the start address of the address range while the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

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

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

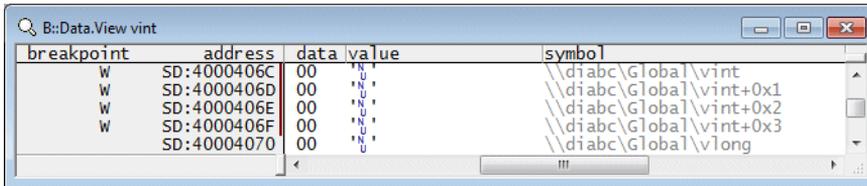
The current setting can be inspected and changed from the **Break.CONFIG** window.

Example: the red line in the [Data.View](#) window shows the range of the Onchip breakpoint.



```
; 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
```

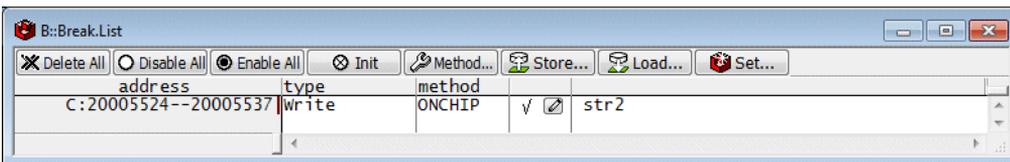


A number of processor architectures provide only **bit masks** or **fixed range sizes** to mark an address range with Onchip breakpoints. In this case the address range is always enlarged to the **smallest bit mask/next allowed range** that includes the address range.

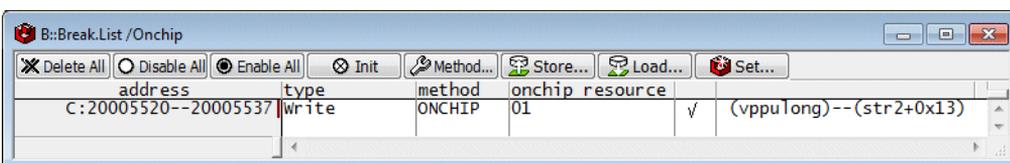
It is recommended to control which addresses are actually marked with breakpoints by using the **Break.List /Onchip** command:

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

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

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

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.

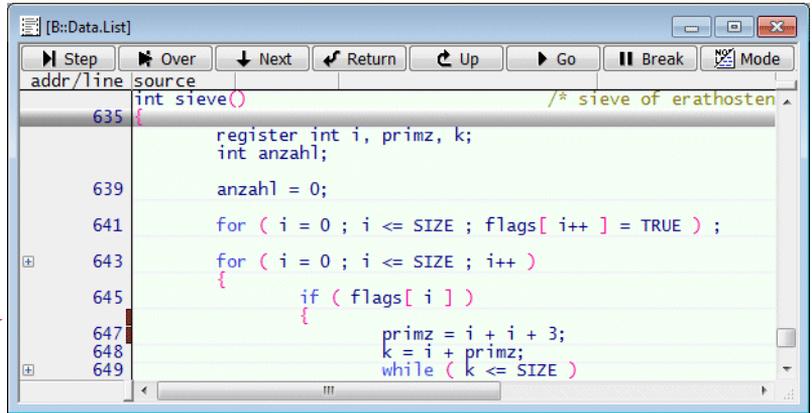
Breakpoint Types

TRACE32 PowerView provides the following breakpoint types for standard debugging.

Breakpoint Types	Possible Implementations
Program	Software (Default) Onchip
Read, Write, ReadWrite	Onchip (Default)

Program Breakpoints

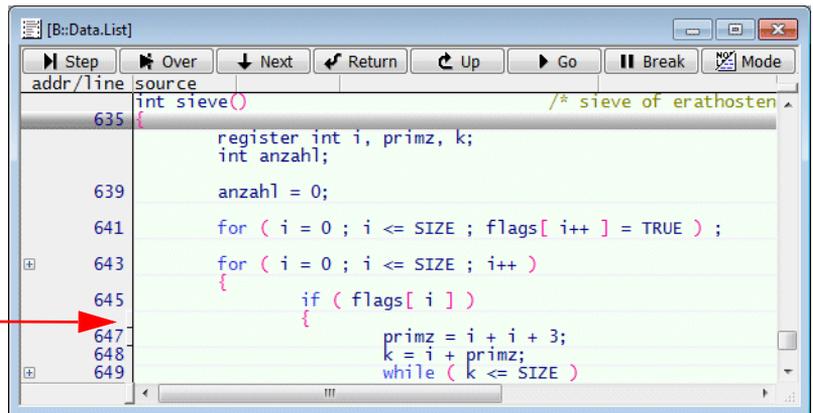
Set a Program breakpoint by a left mouse double-click to the instruction



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)

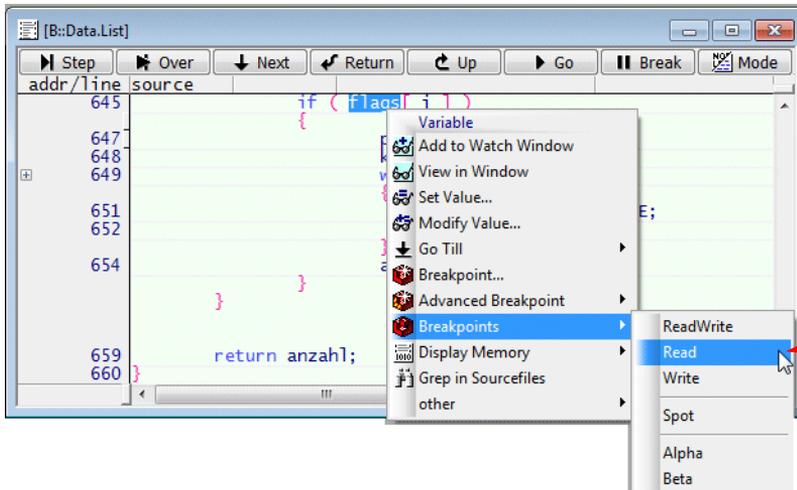
Break.Set func1+0x1c /Program      ; set a Program breakpoint to the
                                   ; instruction at address
                                   ; func1 plus 28 bytes
                                   ; (assuming that byte is the
                                   ; smallest addressable unit)

Break.Set func11\7                 ; set a Program breakpoint to the
                                   ; 7th line of code of the function
                                   ; func11
                                   ; (line in compiled program)

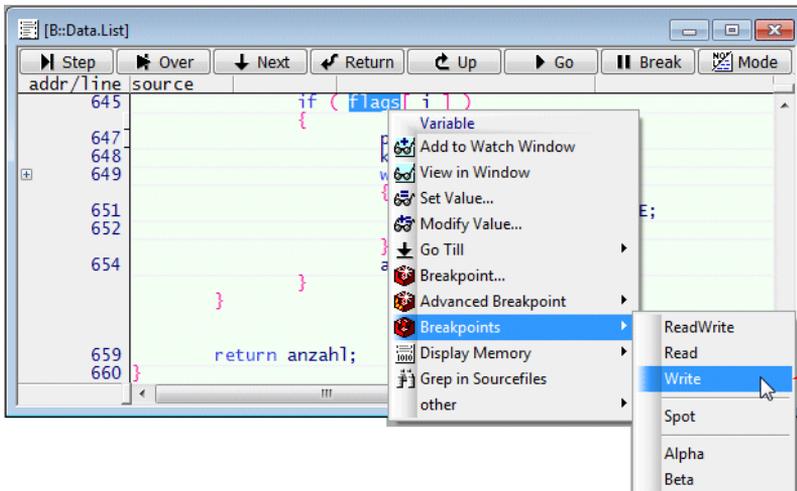
Break.Set func17 /Program /DISable ; set a Program breakpoint to the
                                   ; entry of func17
                                   ; diable Program breakpoint

Break.List                          ; list all breakpoints
```

Read/Write Breakpoints

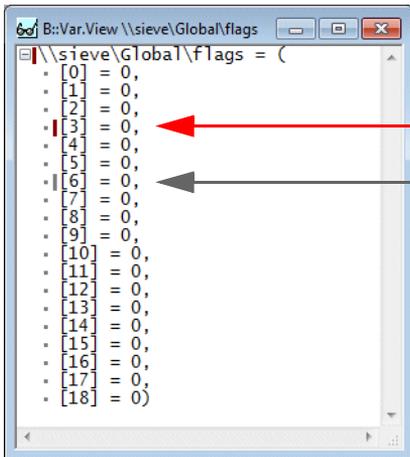


All cores are stopped at a read access to the variable



All cores are stopped at a write access to the variable

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



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

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

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

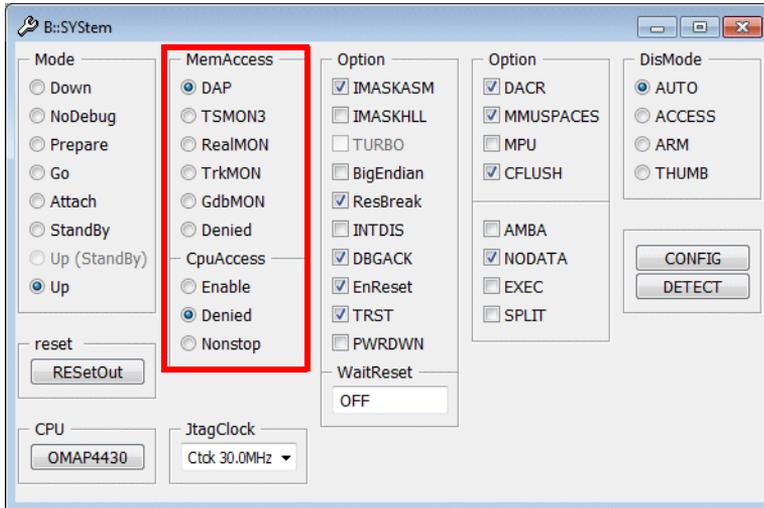
; allow HLL expression to specify breakpoint

Var.Break.Set <hll_expression> /Read | /Write | /ReadWrite [/DISable]

```
Break.Set 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 Handling

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.

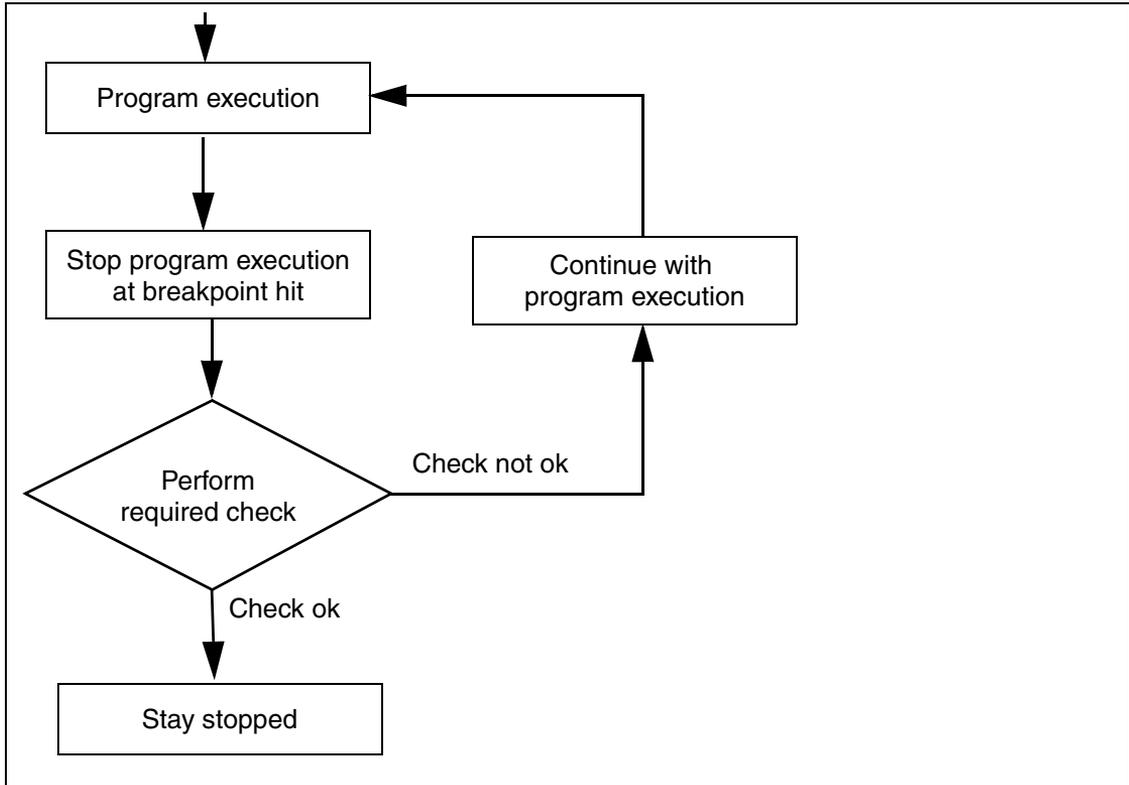
Real-time Breakpoints vs. Intrusive Breakpoints

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.

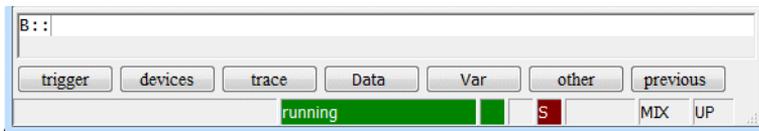
Real-time breakpoints do not disturb the real-time program execution on the core(s), but they require a complex on-chip break logic.

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.

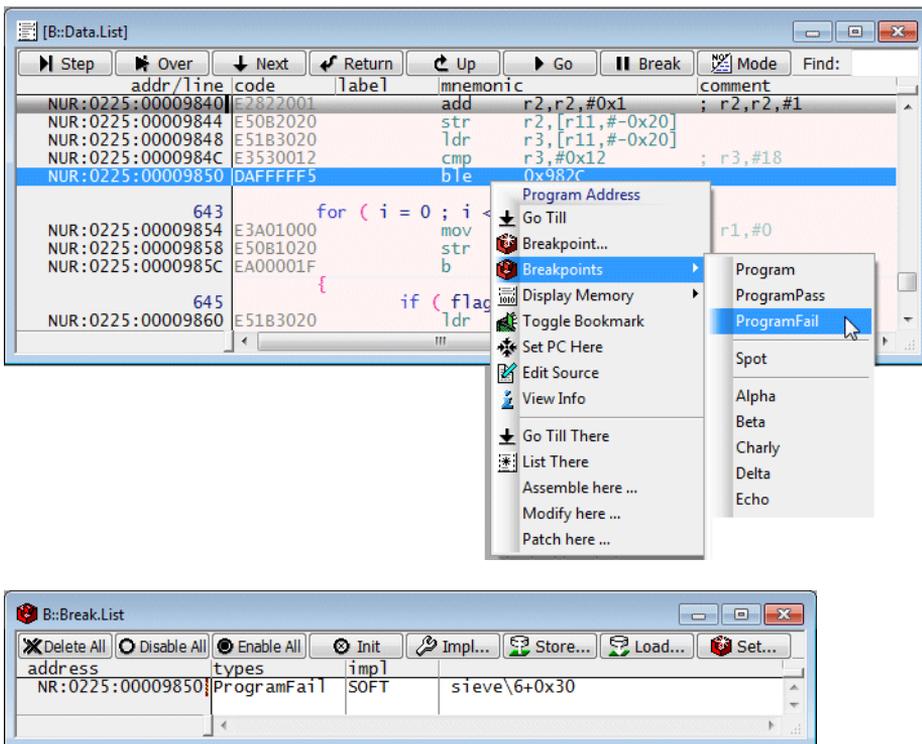
Intrusive breakpoints are marked with a special breakpoint indicator:



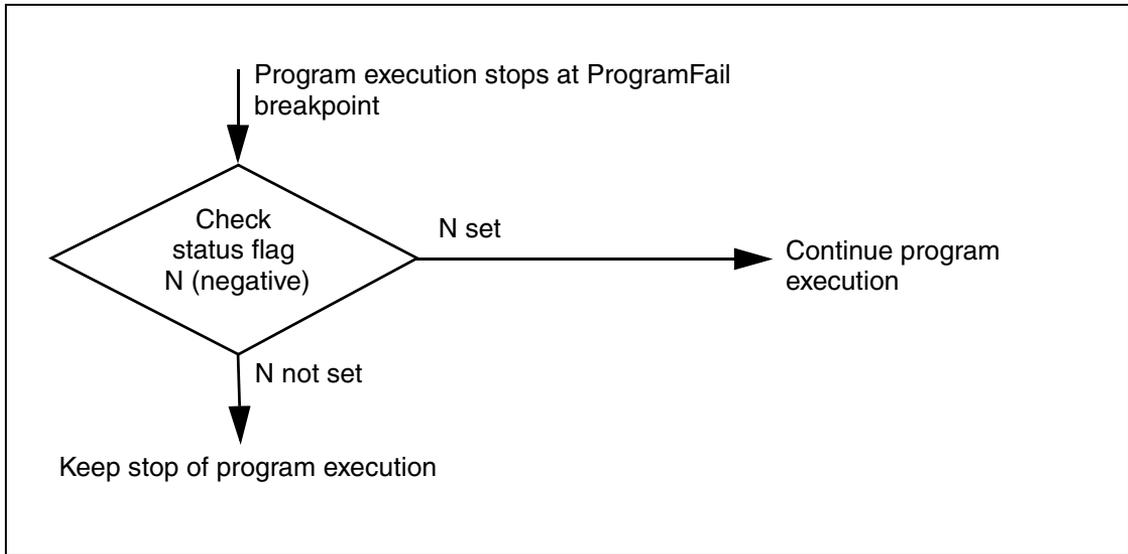
Example for intrusive breakpoint (Cortex-A9): ProgramPass/ProgramFail breakpoint

<p>ProgramPass</p>	<p>If a breakpoint is set to a conditional instruction, the program execution is only stopped, if the condition is satisfied (pass).</p>
<p>ProgramFail</p>	<p>If a breakpoint is set to a conditional instruction, the program execution is only stopped, if the condition fails.</p>

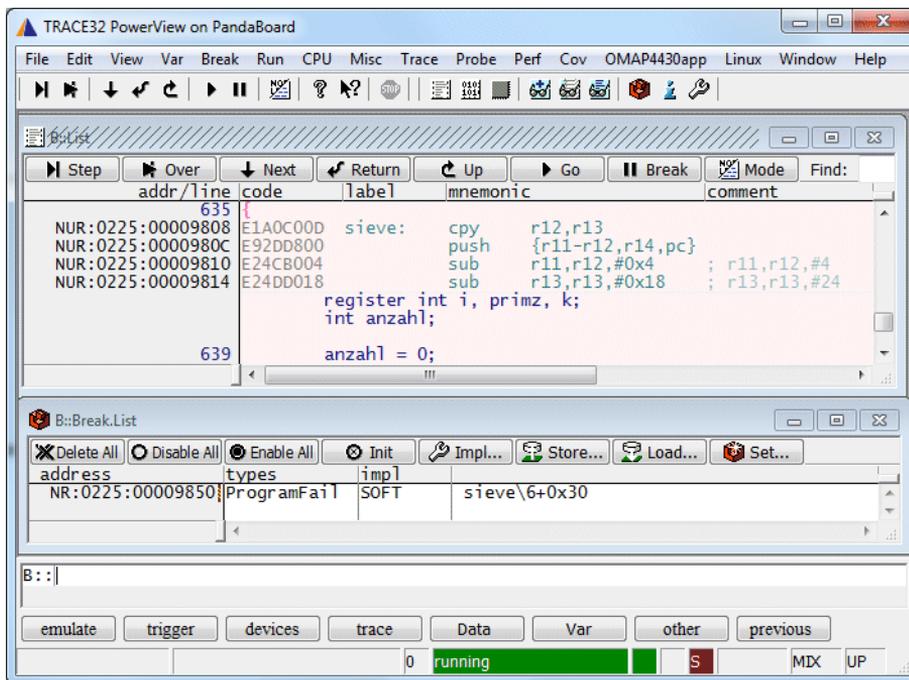
Stop the program execution, when the **ble** instruction fails.



The ProgramFail breakpoint behaves as follows:



Each stop to check the status flag takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line.



TRACE32 PowerView on PandaBoard

File Edit View Var Break Run CPU Misc Trace Probe Perf Cov OMAP4430app Linux Window Help

B::Data.List

Step	Over	Next	Return	Up	Go	Break	Mode	
addr/line	code	label	mnemonic		comment			
NUR:0225:00009840	E2822001		add	r2,r2,#0x1	; r2,r2			
NUR:0225:00009844	E50B2020		str	r2,[r11,#-0x20]				
NUR:0225:00009848	E51B3020		ldr	r3,[r11,#-0x20]				
NUR:0225:0000984C	E3530012		cmp	r3,#0x12	; r3,#1			
NUR:0225:00009850	DAFFFFFF5		bne	0x982C				
643			for (i = 0 ; i <= SIZE ; i++)					
NUR:0225:00009854	E3A01000		mov	r1,#0x0	; r1,#0			
NUR:0225:00009858	E50B1020		str	r1,[r11,#-0x20]				
NUR:0225:0000985C	EA00001F		b	0x98E0				
645			{					
			if (flags[i])					
NUR:0225:00009860	E51B3020		ldr	r3,[r11,#-0x20]				
NUR:0225:00009864	E59F2090		ldr	r2,0x98FC				

B::Break.List

Delete All Disable All Enable All Init Impl... Store... Load... Set...

address	types	impl	
NR:0225:00009850	ProgramFail	SOFT	sieve\6+0x30

B::Register

N	-	R0	0	R8	0	SP	00000001
Z	-	R1	12	R9	0	-24	00000013
C	C	R2	13	R10	B6F08000	-20	00000025
V	-	R3	13	R11	BE9AAD14	-1C	00011B84
Q	-	R4	00011B84	R12	BE9AAD18	-18	00000000
R5	-	R5	0	R13	BE9AACF0	-14	00000000
0	-	R6	0	R14	97E8	-10	BE9AAD74
1	-	R7	0	PC	9850	-0C	BE9AAD18
2	-	SPSR		CPSR	20000010	-08	000097E8

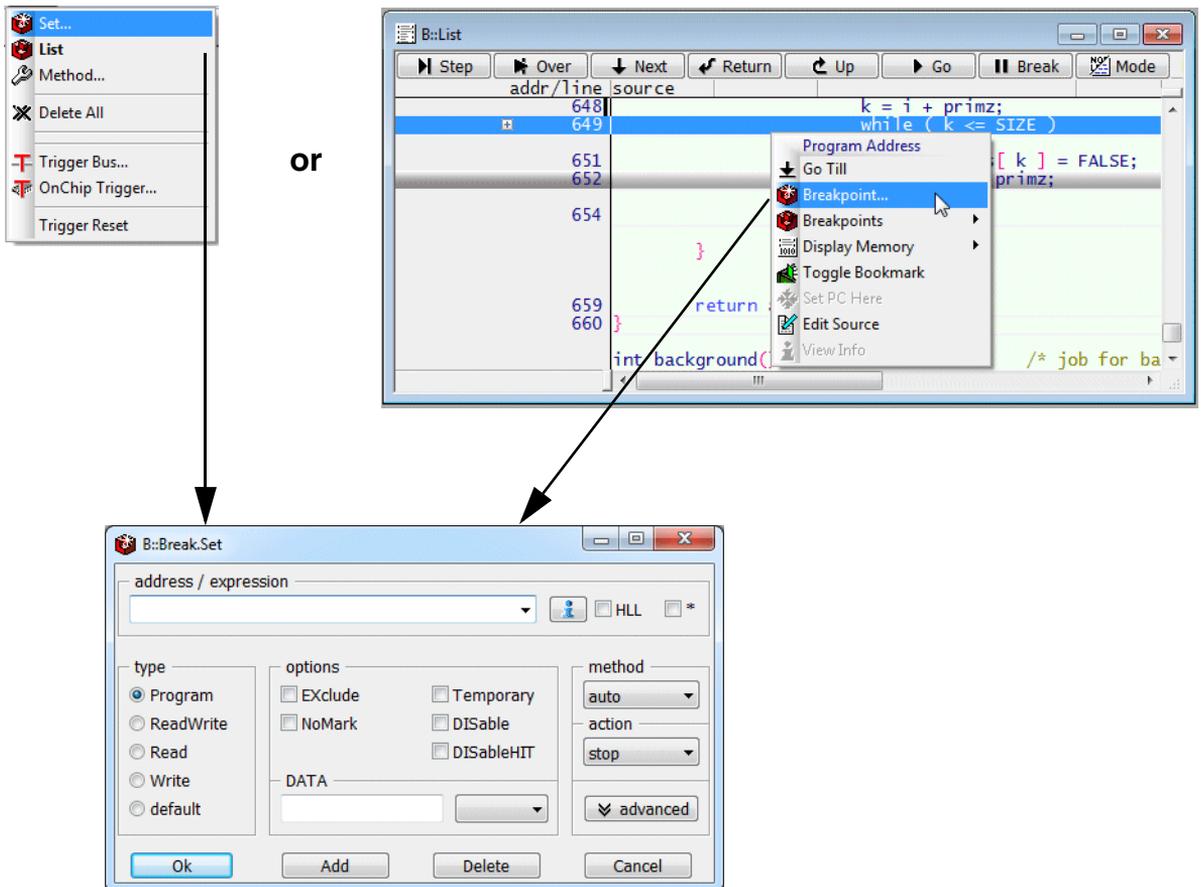
B::

emulate trigger devices trace Data Var other previous

NUR:0225:00009850 sieve 0 stopped at breakpoint MDX UP

Break.Set Dialog Box

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



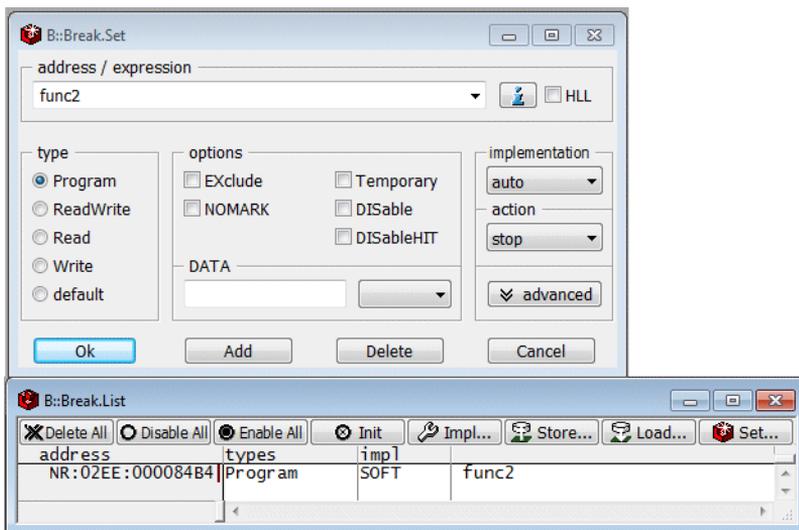
The HLL Check Box - Function Name

```
sYmbol.INFO func2                ; display symbol information
                                   ; for function func2
```

Function Name/HLL Check Box OFF

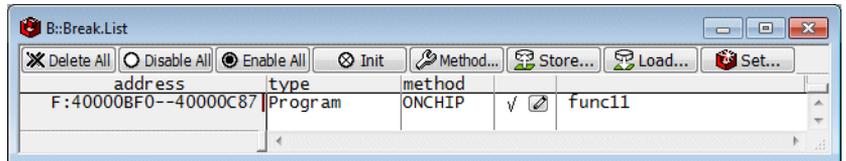
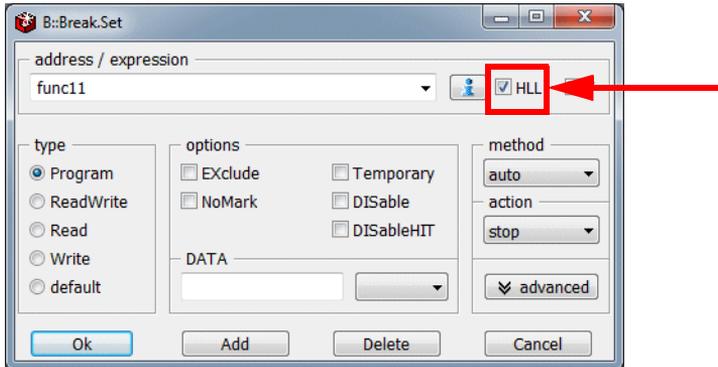
Program breakpoint is set to the function entry (first address of the function).

```
Break.Set func11
```



Function name/HLL Check Box ON (only for special use cases)

- 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
```

The HLL Check Box - Program Line Number

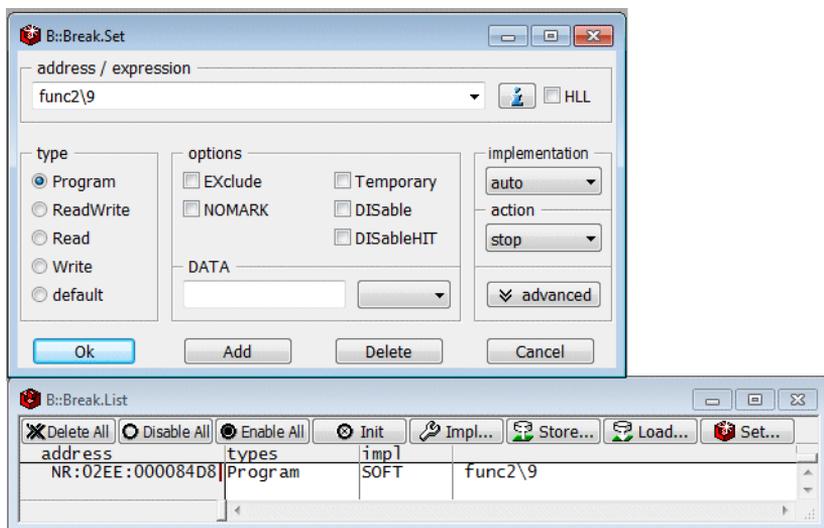
```
sYmbol.INFO func2\9 ; display symbol information  
; for 9th program line in  
; function func2
```



Program Line Number/HLL Check Box OFF

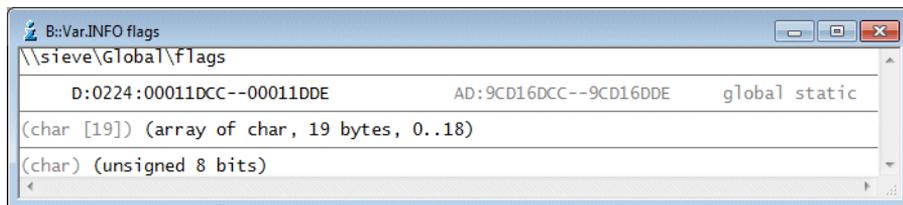
Program breakpoint is set to the first assembler instruction generated for the program line number.

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



The HLL Check Box - Variable

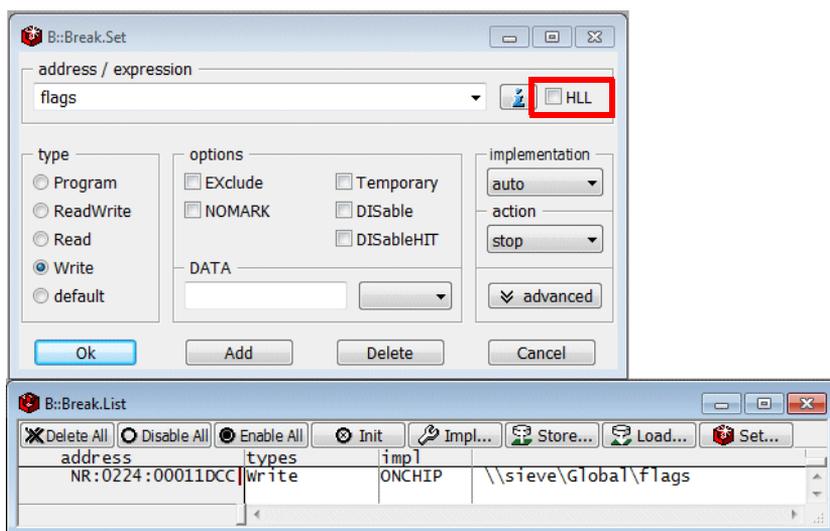
```
sYmbol.INFO flags ; display symbol information  
; for variable flags
```



Variable/HLL Check Box OFF

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

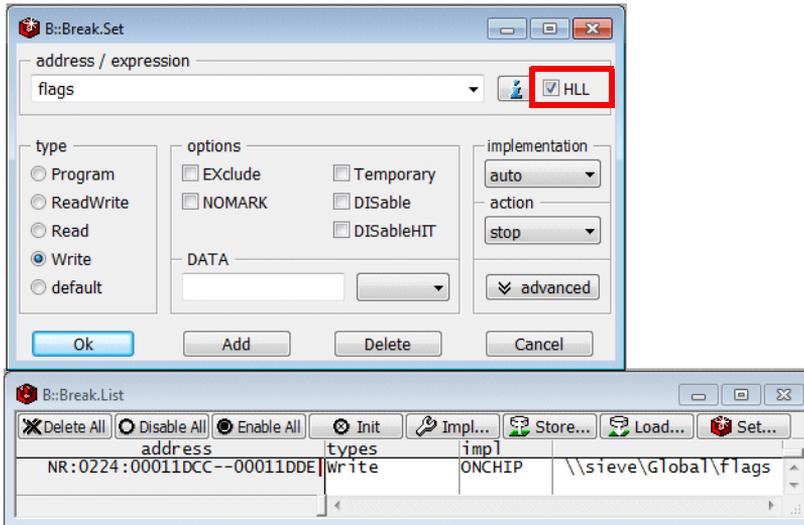
```
Break.Set flags
```



Variable/HLL Check Box ON

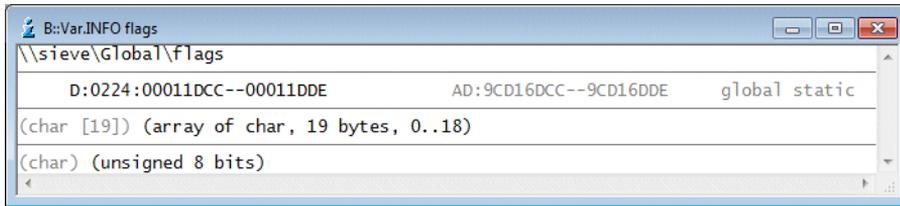
- If the on-chip break logic supports ranges for Read/Write breakpoints, the specified breakpoint is set to the complete address range covered by the variable.
- If the on-chip break logic provides only bitmasks to realize Read/Write breakpoints on address ranges, the specified breakpoint is set by using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

Var.Break.Set flags



The HLL Check Box - HLL Expression

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

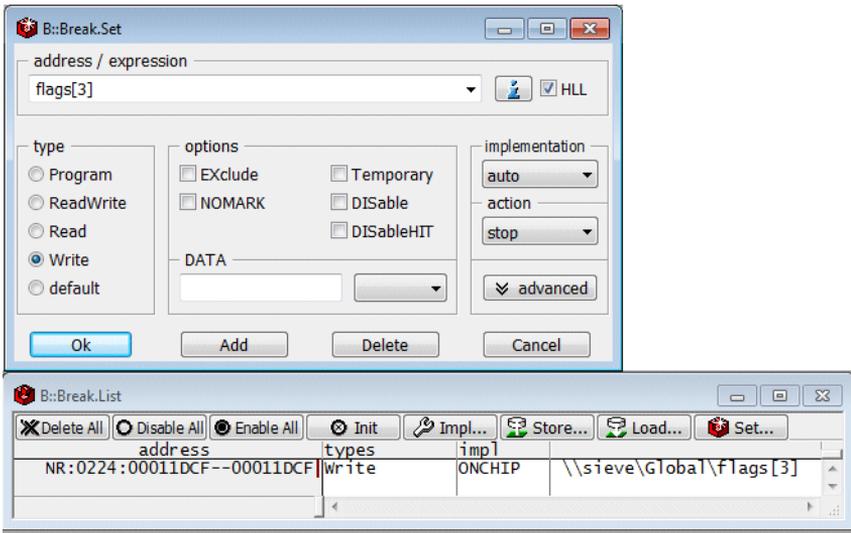


Variable/HLL Check Box Must Be ON

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

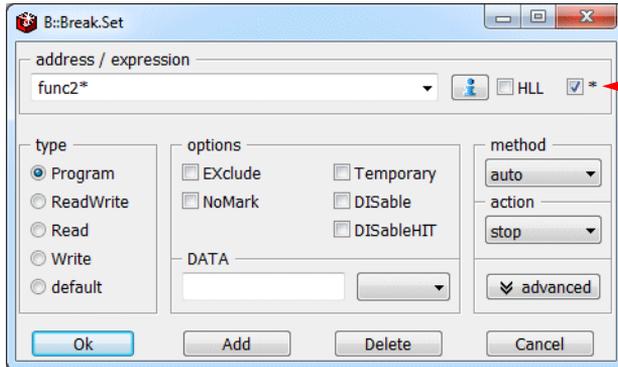
- If the on-chip break logic supports ranges for Read/Write breakpoints, the specified breakpoint is set to the complete address range covered by the HLL expression.
- If the on-chip break logic provides only bitmasks to realizes Read/Write breakpoints on address ranges, the specified breakpoint is set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

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

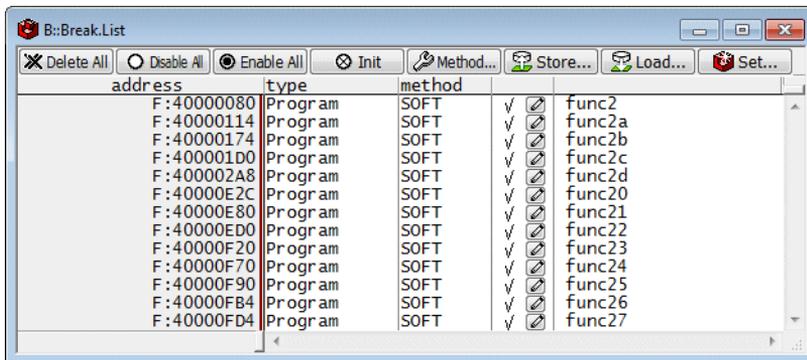


Allow Wildcards in address/expression

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



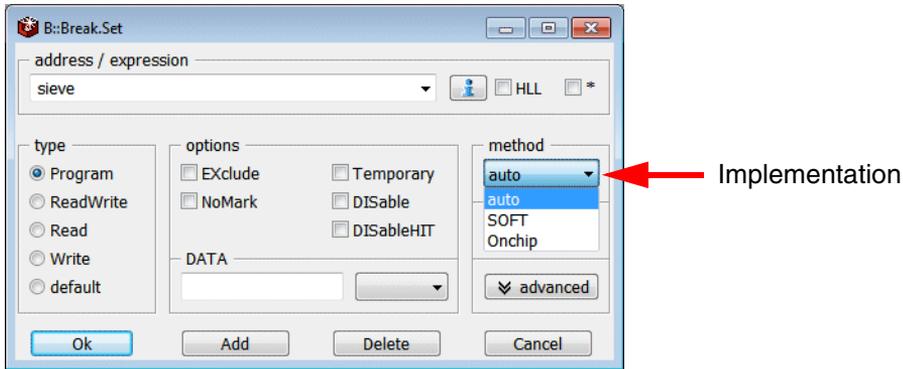
Check * to enable wildcard usage



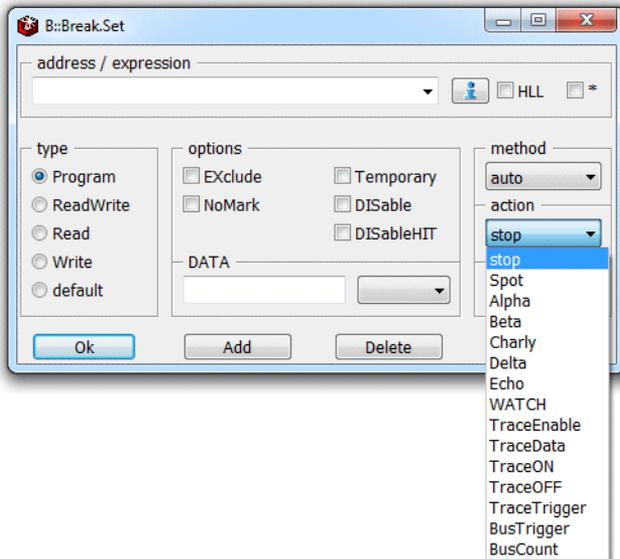
Requires sufficient resources if Onchip breakpoints are used.

```
Break.SetPATtern func2*
```

Implementations



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 CoreSight ETM Tracing”](#) (training_arm_etm.pdf).
- [“Training Cortex-M Tracing”](#) (training_cortexm_etm.pdf).
- [“Training AURIX Tracing”](#) (training_aurix_trace.pdf).
- [“Training Hexagon ETM Tracing”](#) (training_hexagon_etm.pdf).
- [“Training Nexus Tracing”](#) (training_nexus.pdf).

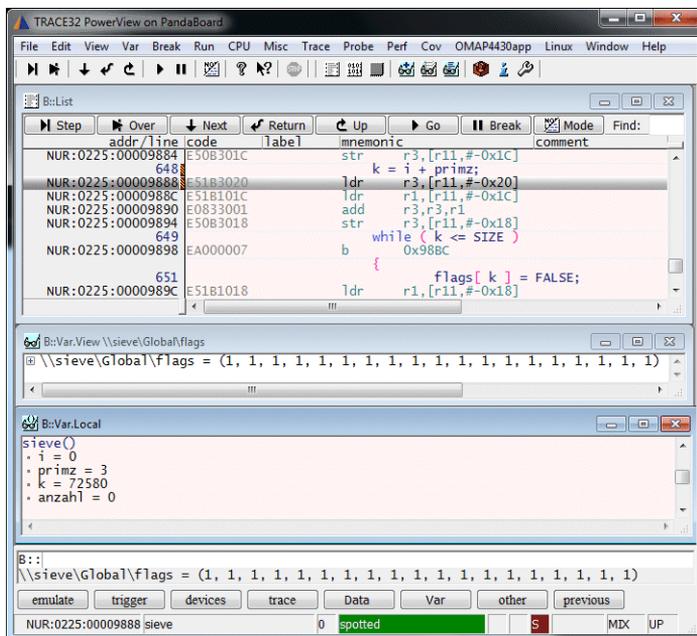
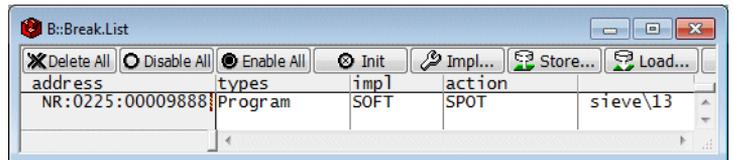
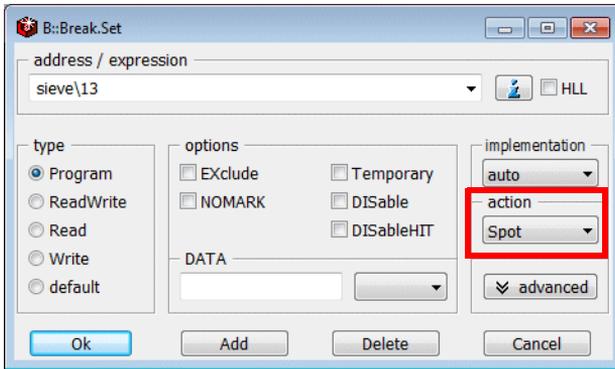
or with the description of the [Break.Set](#) command.

Example for the Action Spot

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

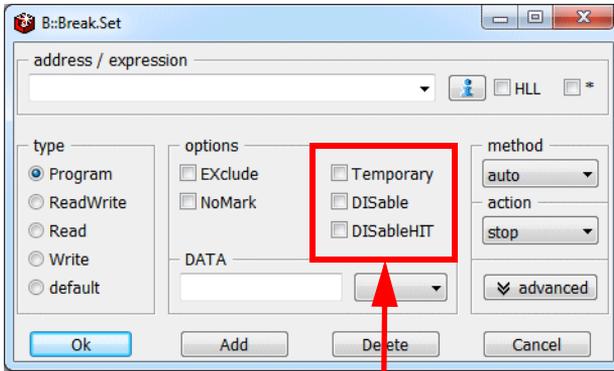
The action Spot can be used to turn a breakpoint into a watchpoint. The core stops the program execution at the watchpoint, updates the screen and restarts the program execution automatically. Each stop takes **50 ... 100 ms** depending on the speed of the debug interface and the amount of information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

Example: Update the screen whenever the program executes the instruction sieve\13.



A spotpoint is active and the system is no longer running in real-time

Options



Options

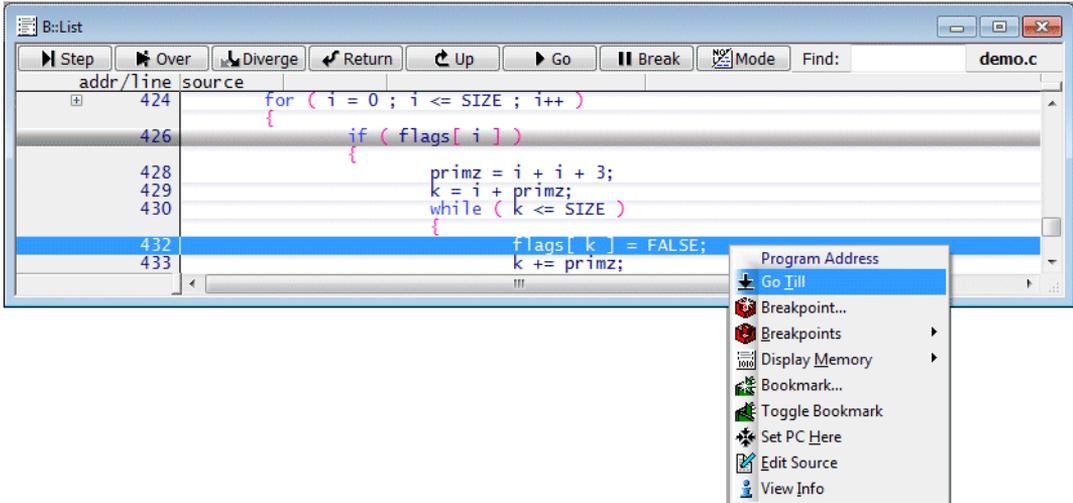
Temporary	OFF: Set a permanent breakpoint (default). ON: Set a temporary breakpoint. All temporary breakpoints are deleted the next time the core(s) stops the program execution.
DISable	OFF: Breakpoint is enabled (default). ON: Set breakpoint, but disabled.
DISableHIT	ON: Disable the breakpoint after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

Example for the Option Temporary

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

Examples:

- **Go Till**



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

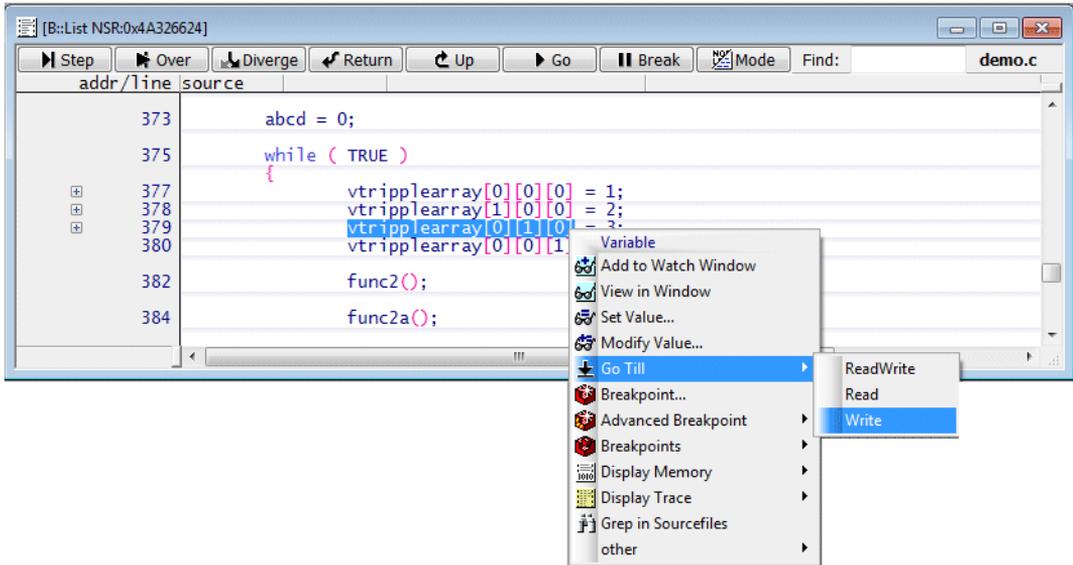
```
; set a temporary Program breakpoint to  
; the entry of the function func4  
; and start the program execution
```

Go func4

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

Go func4 func8 func9

- **Go Till -> Write**



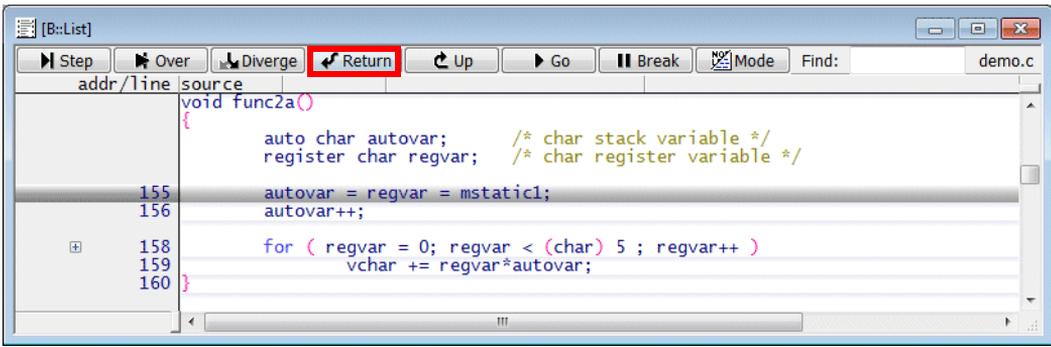
Var.Go <hl_expression> [/Write]

```

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

```

- **Go.Return and similar commands**



Go.Return

```

; first Go.Return
; set a temporary breakpoint to the start of the function epilogue
; and start the program execution

```

Go.Return

```

; stopping at the function epiloug first has the advantage that the
; local variables are still valid at this point.

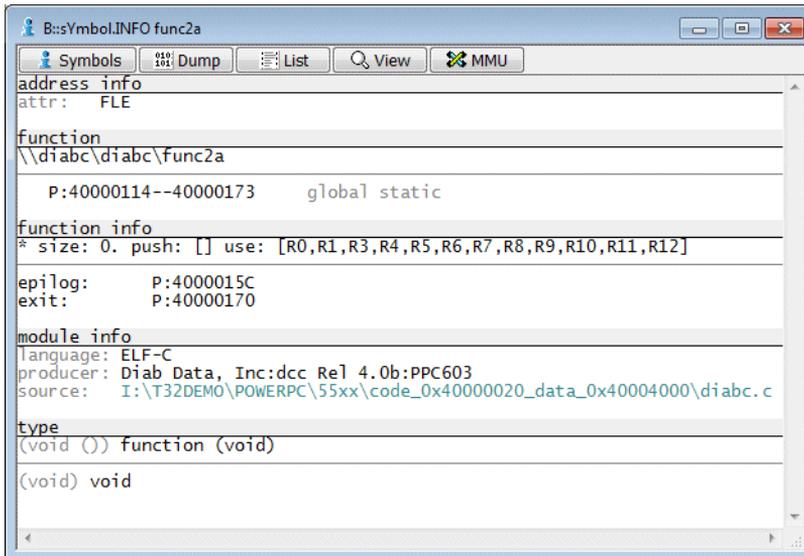
```

```

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

```

Go.Return



DATA Breakpoints

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

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

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



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

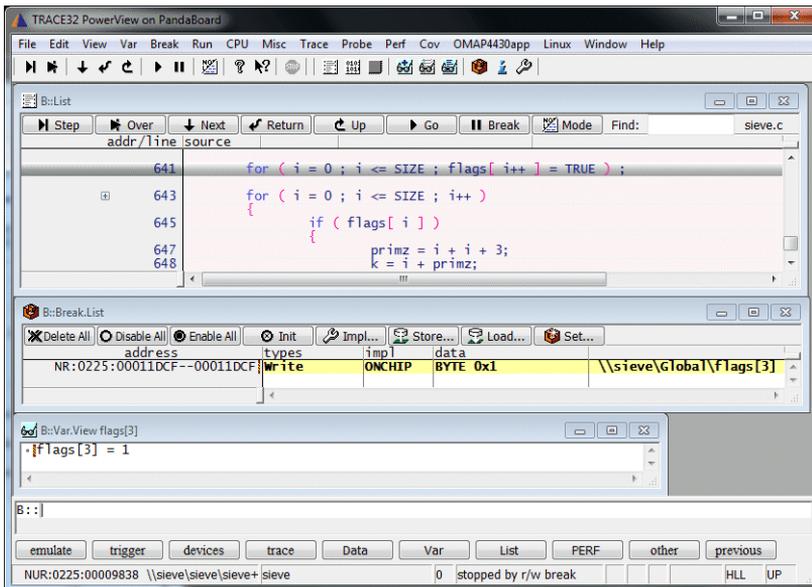
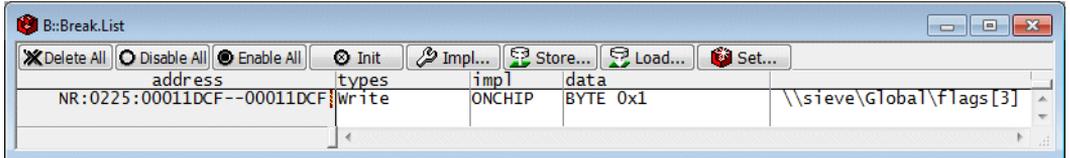
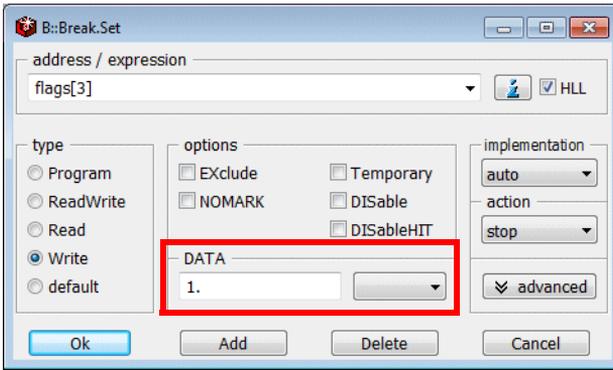
- DATA breakpoints are implemented as intrusive breakpoints if the core does not support Data Value Breakpoints. For details on the intrusive DATA breakpoints refer to the description of the [Break.Set](#) command.

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



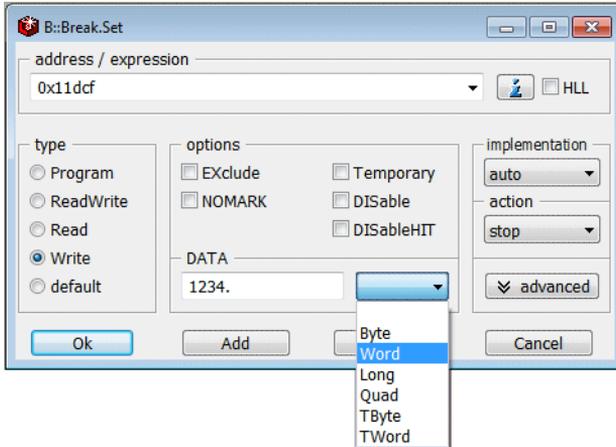
TRACE32 PowerView allows inverted data values for intrusive DATA breakpoints.

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



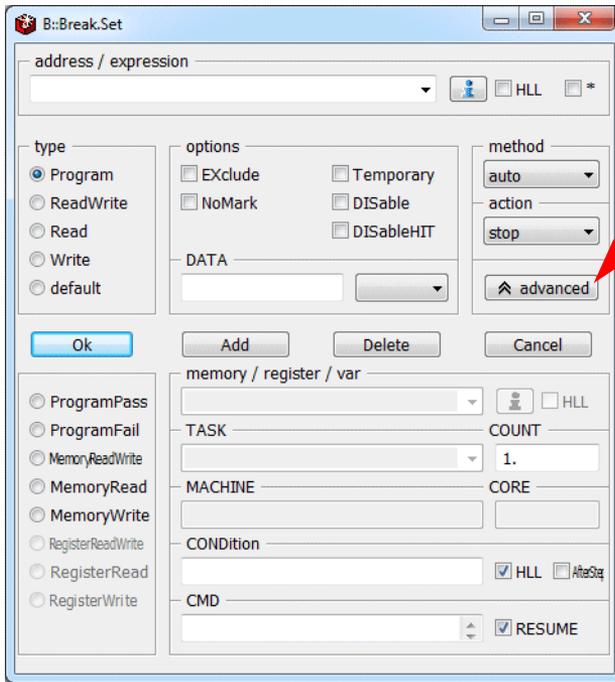
If an HLL expression is used TRACE32 PowerView gets the information if the data is written via a byte, word or long access from the symbol information.

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



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

Advanced Breakpoints



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

Advanced breakpoint input fields

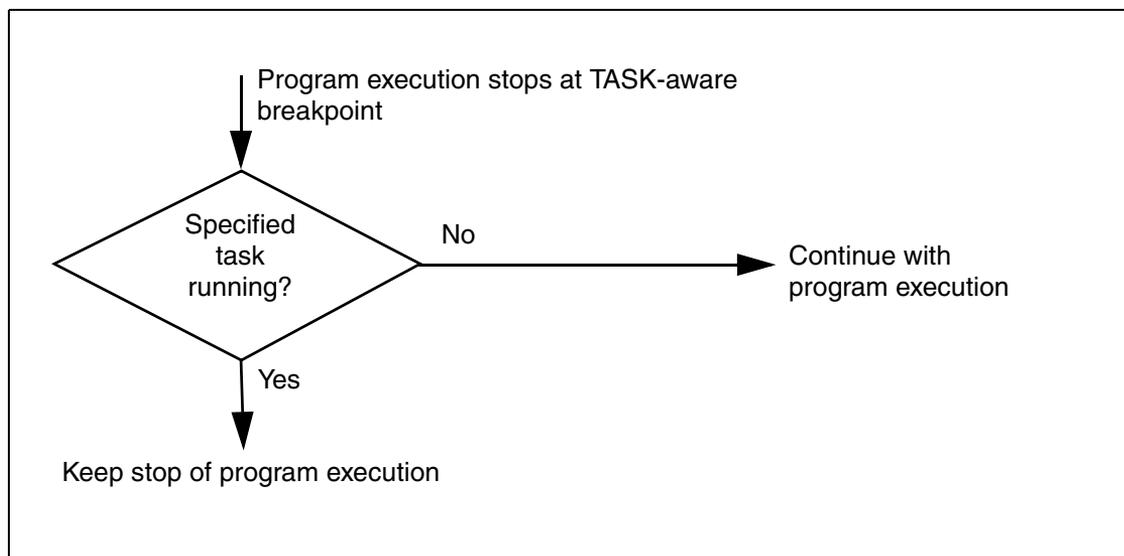
TASK-aware Breakpoints

If OS-aware debugging is configured (refer to “[OS-aware Debugging](#)” in TRACE32 Glossary, page 31 (glossary.pdf)), TASK-aware breakpoints allow to stop the program execution at a breakpoint if the specified task/process is running.

TASK-aware breakpoints are implemented on most cores as intrusive breakpoints. A few cores support real-time TASK-aware breakpoints (e.g. ARM/Cortex). For details on the real-time TASK-aware breakpoints refer to the description of the **Break.Set** command.

Intrusive TASK-aware Breakpoint

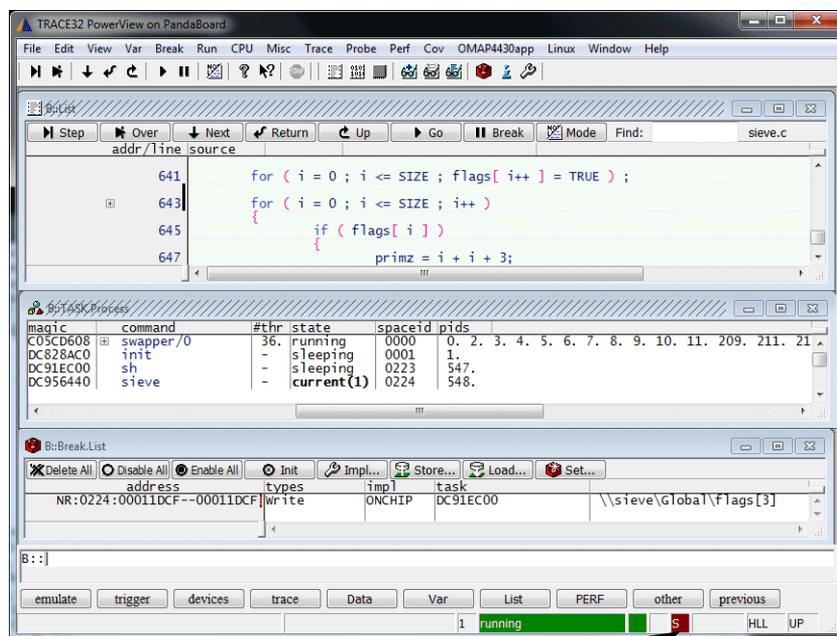
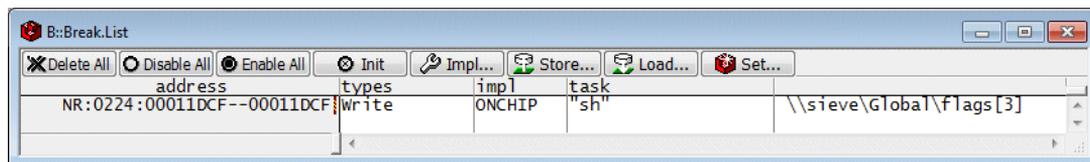
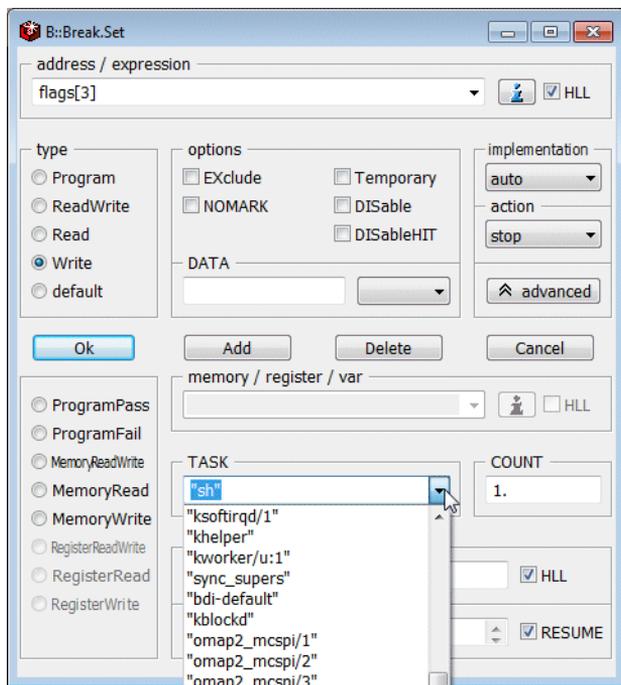
Processing:



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

EE_oo_TerminateTask

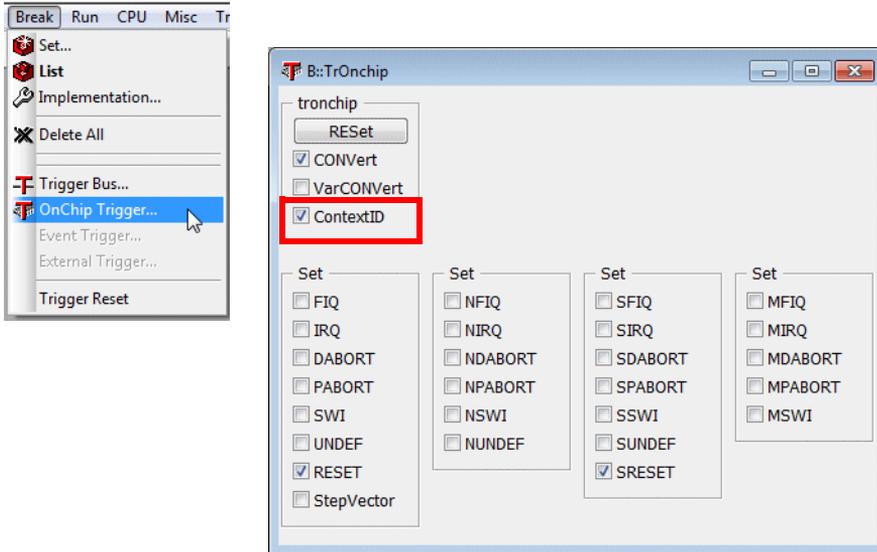
Example: Stop the program execution at a write access to the variable flags[3] only when the task/process “sh” is performing this write access.



The red S indicates an intrusive breakpoint

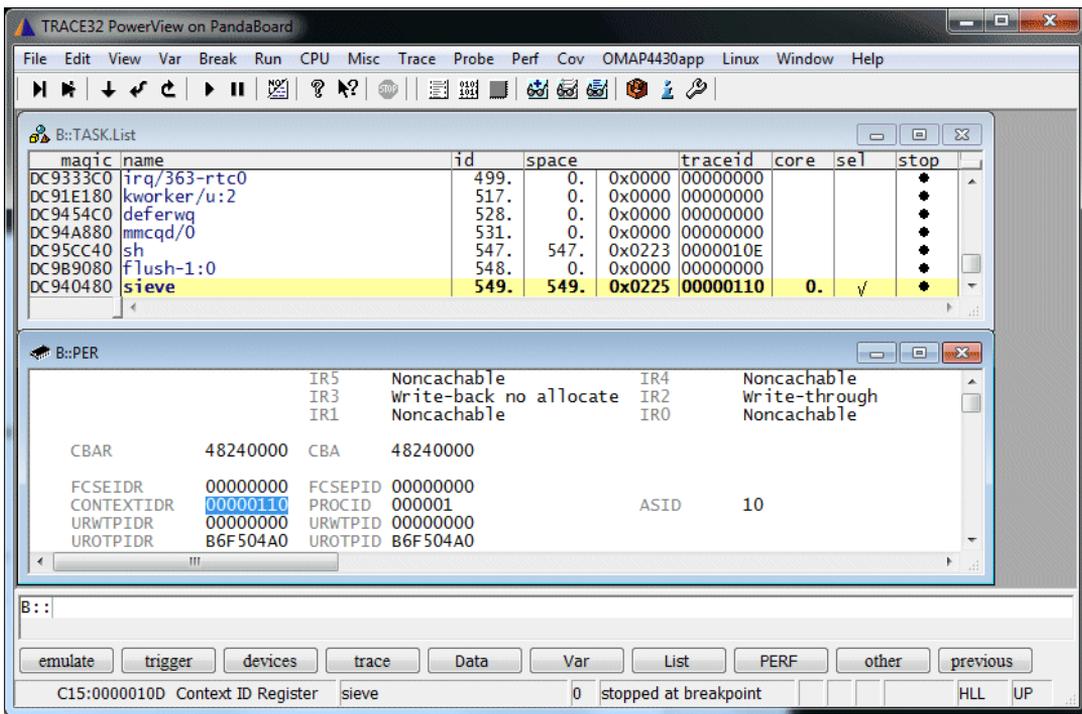
Example for Cortex-A9: Stop the program execution at a write access to the variable flags[3] only when the task/process “sh” is performing this write access (here Cortex-A9).

TRACE32 PowerView can realize real-time TASK-aware breakpoints, if the operating system updates the **Context ID Register** (CONTEXTIDR) on every process/task switch. Set the **ContextID** check-box in the **TrOnchip** window to ON to inform TRACE32 PowerView about this update.

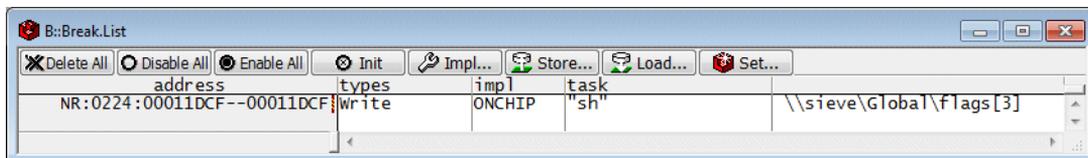
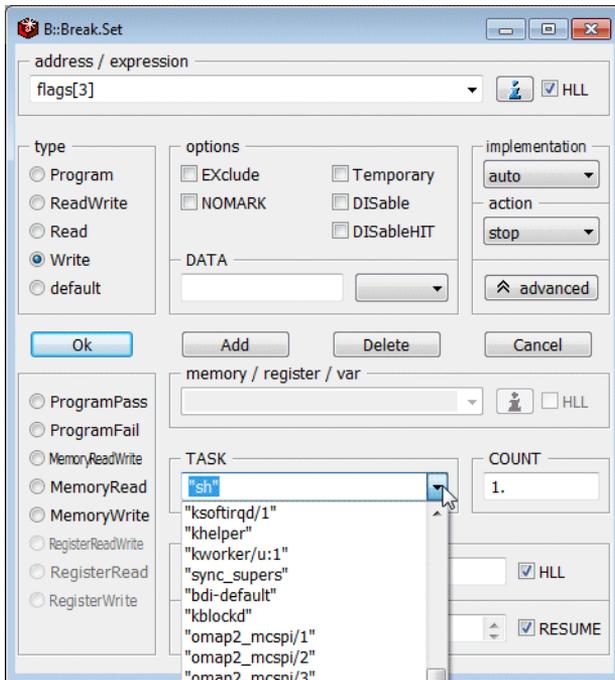


TrOnchip ContextID ON

Enable TASK-aware breakpoints (Onchip and ETM)



The **traceid** in the **TASK.List** window helps you to decode the contents of the **CONTEXTIDR**.



TRACE32 PowerView on PandaBoard

File Edit View Var Break Run CPU Misc Trace Probe Perf Cov OMAP4430app Linux Window Help

Step Over Next Return Up Go Break Mode Find: sleep44x

addr	line	source
359		wf1
360		nop
361		nop
362		nop
363		nop
364		nop

@ Wait For Interrupt

86 Task Process

magic	command	#thr	state	spaceid	pids
C05CD608	swapper/0	36	current	0000	0. 2. 3. 4. 5. 6. 7. 8. 9. 10. 11. 209. 211.
DC828AC0	init	-	sleeping	0001	1.
DC95CC40	sh	-	sleeping	0223	547.
DC940480	stieve	-	sleeping	0225	549.

B::Break.List

Delete All Disable All Enable All Init Impl... Store... Load... Set...

address	types	impl	task
NR:0225:00011DCF--00011DCF	write	ONCHIP	DC95CC40

B:::

emulate trigger devices trace Data Var List PERF other previous

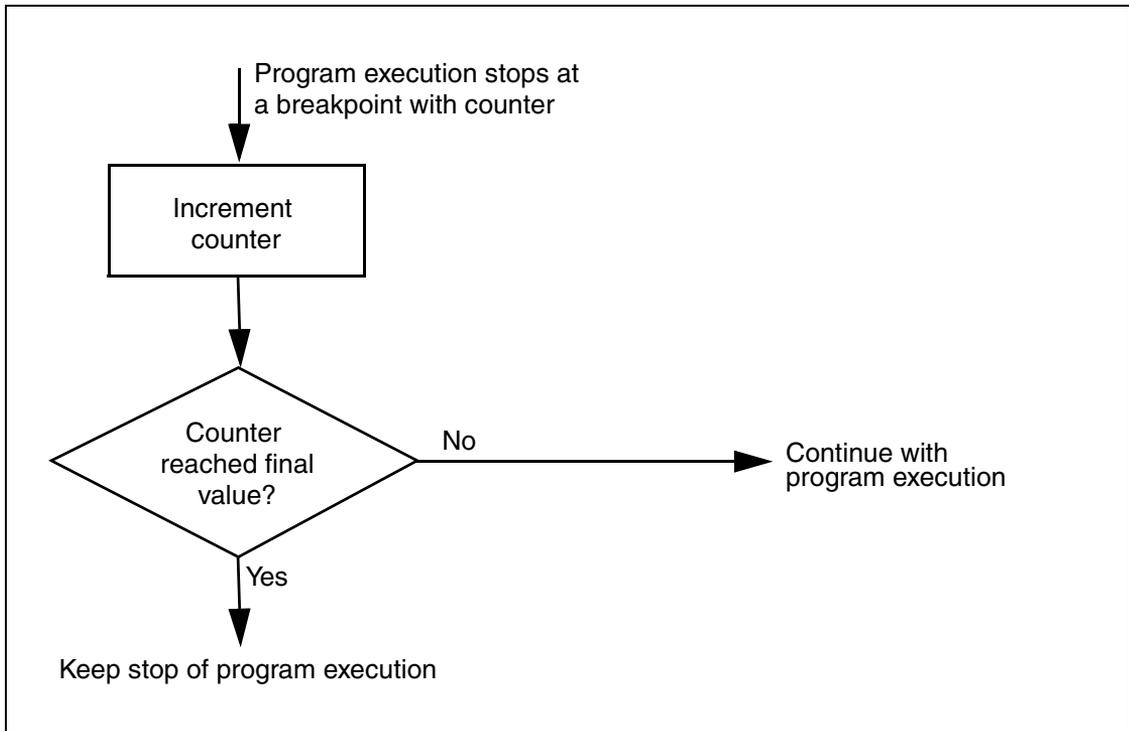
0 running HLL UP

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

Software Counter

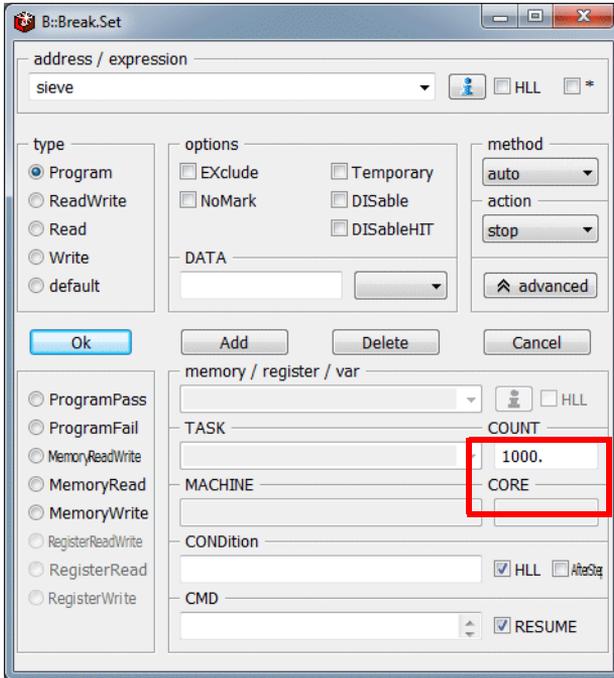
If the on-chip break logic of the core does not provide counters or if a Software breakpoint is used, counters are implemented as software counters.

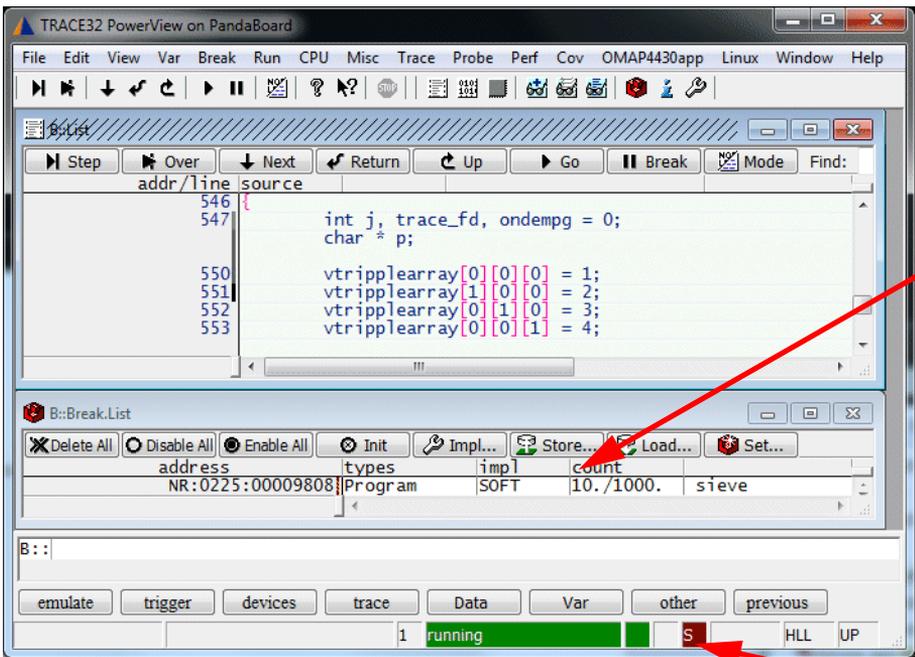
Processing:



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.

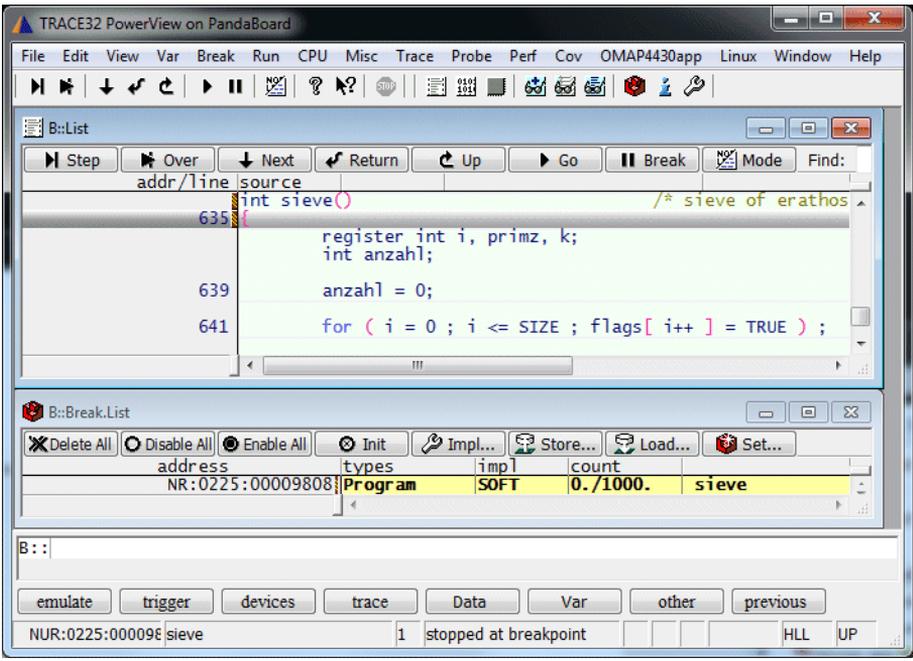
Example: Stop the program execution after the function sieve was entered 1000. times.





The current counter contents is permanently updated

The red S indicates an intrusive breakpoint

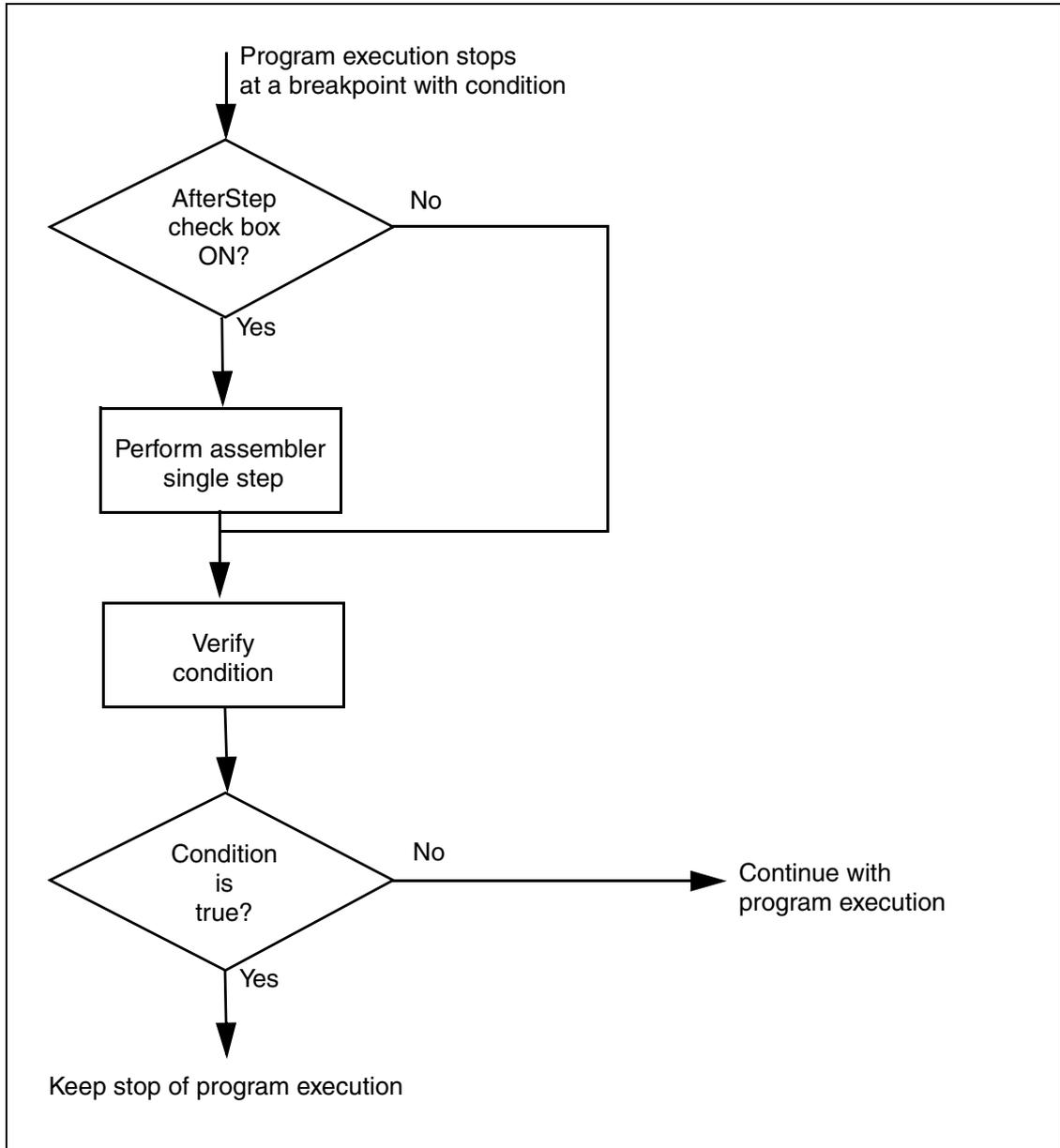


CONDition

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

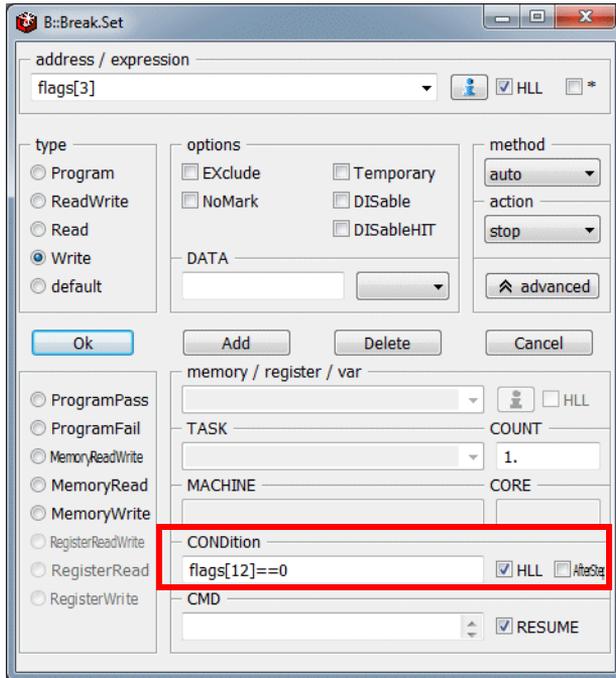
CONDition breakpoints are always intrusive.

Processing:

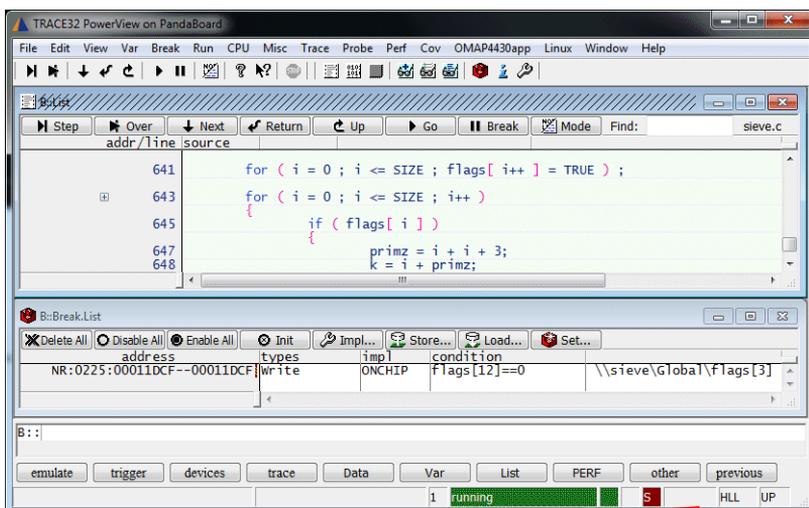
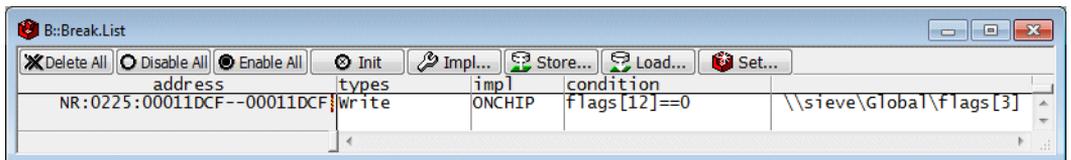


Each stop at a CONDition breakpoint takes at least 1.ms. This is why the red S is displayed in the TRACE32 PowerView state line whenever the breakpoint is hit.

Example: Stop the program execution on a write to flags[3] only if flags[12] is equal to 1 when the breakpoint is hit.



When the breakpoint is reached, the core(s)/CPU is stopped for a moment, the condition is checked and the program execution continues when the condition is not true.

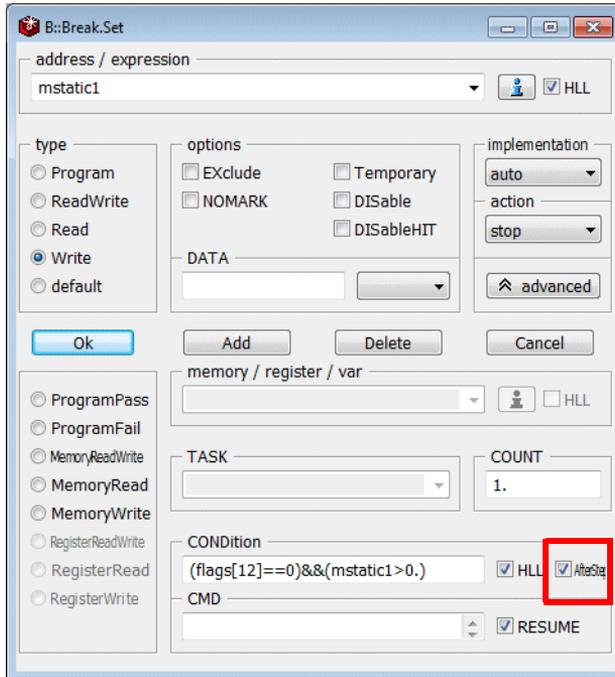


The red S indicates an intrusive breakpoint

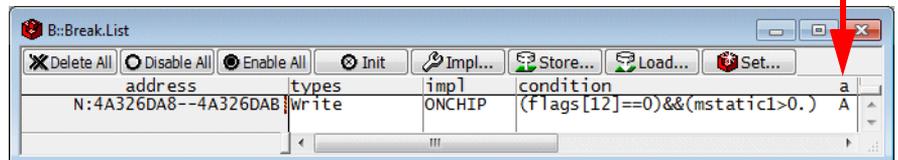
Example: “Break-before-make” Read/Write breakpoints only

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

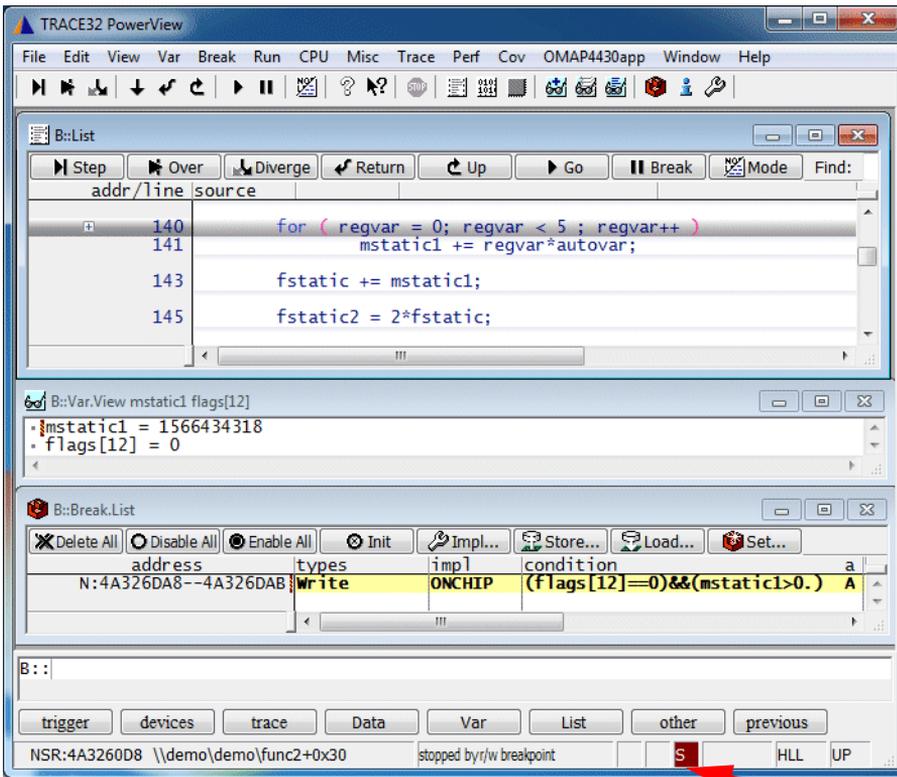
Perform an assembler single step because the processor architecture stops before the write access is performed.



AfterStep checked



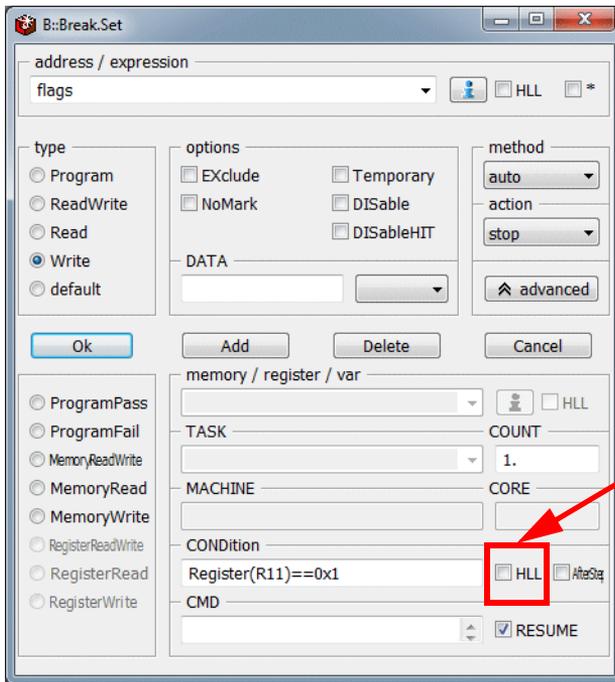
```
Var.Break.Set mstatic1 /Write /VarCONDition (flags[12]==0)&&(mstatic1>0) /AfterStep
```



The red S indicates an intrusive breakpoint

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

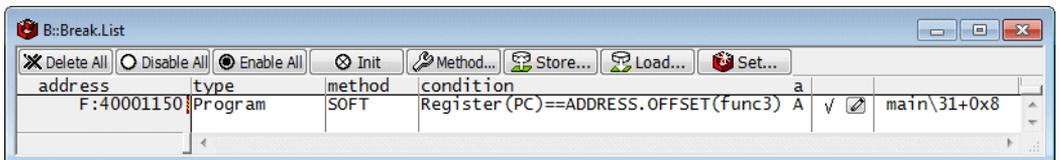
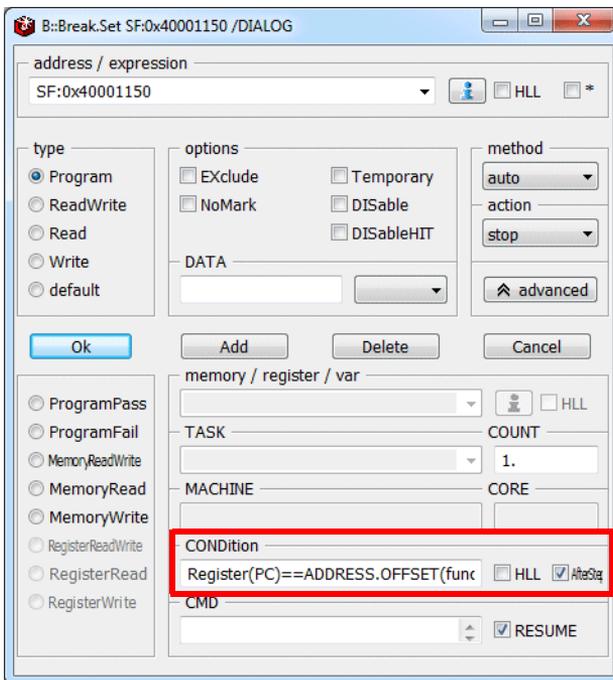
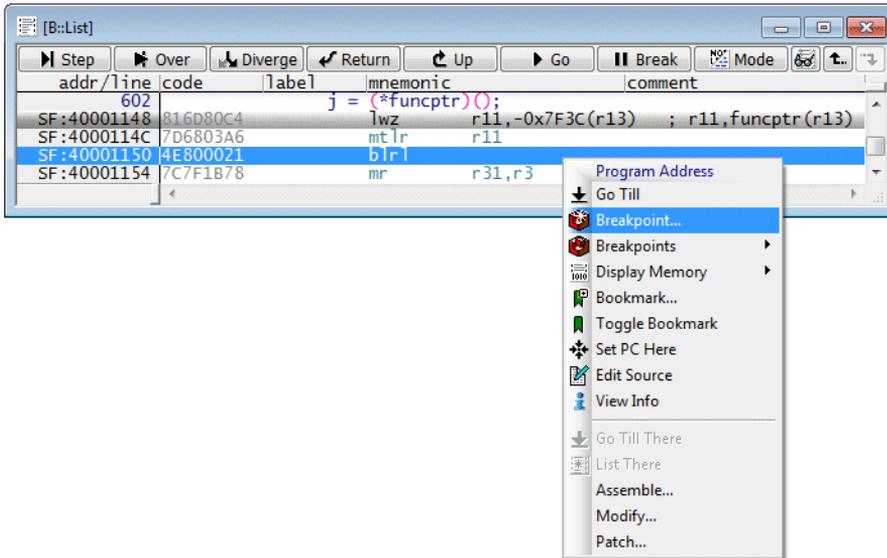
Examples: Stop the program executions on a write to the address flags if Register R11 is equal to 1.



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

addr/line	code	label	mnemonic	comment
232				/* simple function */
	{			
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

address	type	method	condition	a		
F:40001150	Program	SOFT	Register(PC)==ADDRESS.OFFSET(func3)	A	✓	main\31+0x8

B:::

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
```

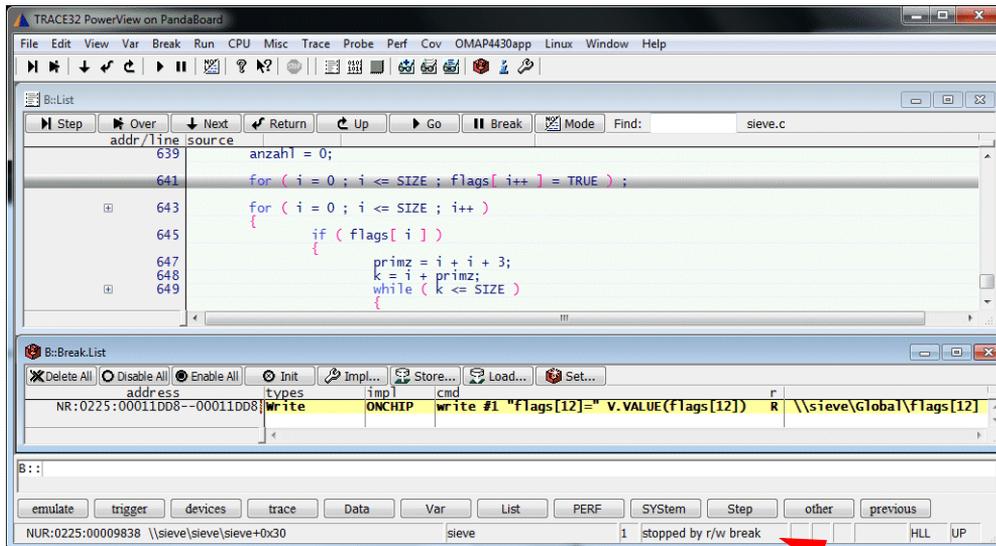
```
Var.Break.Set flags[12] /Write /CMD "WRITE #1 ""flags[12]="" %Decimal  
Var.VALUE(flags[12])" /RESUME
```



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.

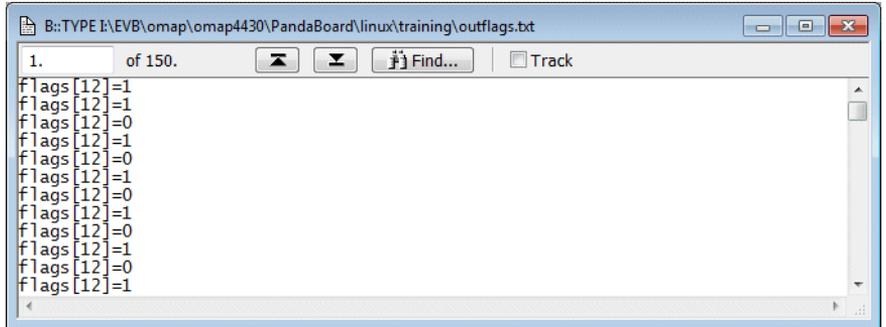
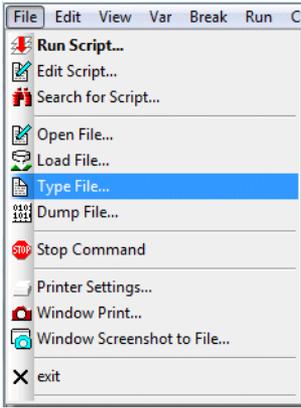


The state of the debugger toggles between **going** and **stopped**

CLOSE #1

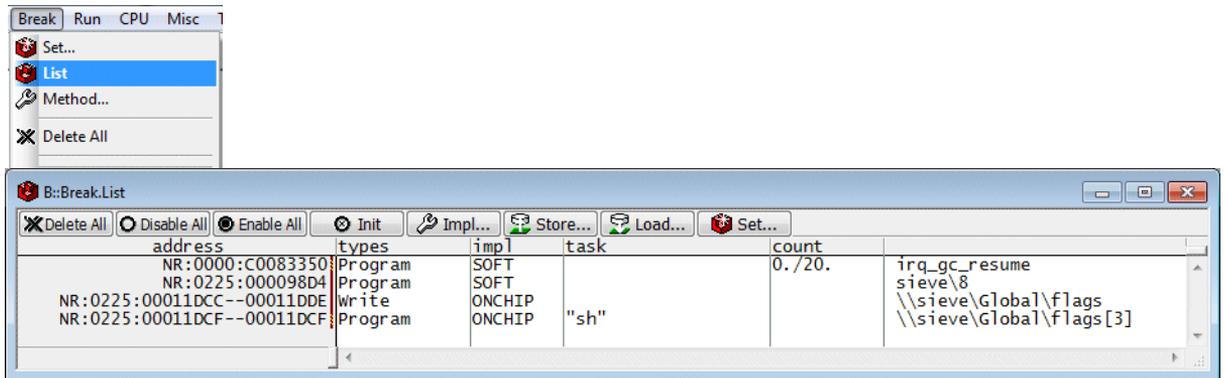
; close the file when you are done

Display the result:



•

Display a List of all Set Breakpoints

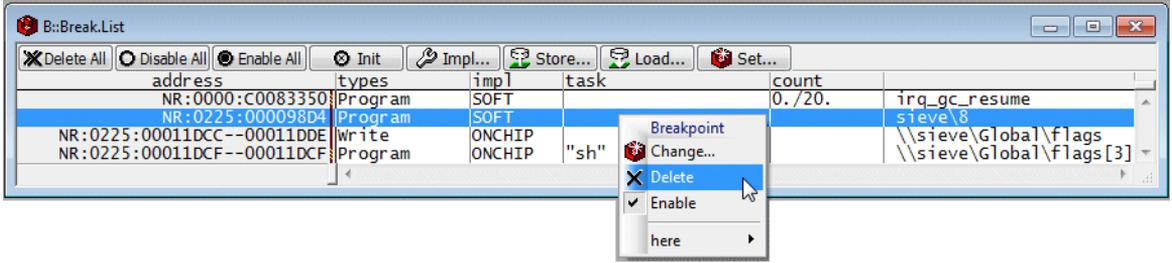


address	Address of the breakpoint
types	Type of the breakpoint
impl	Implementation of the breakpoint or disabled
action	Action selected for the breakpoint (if not stop)
options	Option defined for the breakpoint
data	Data value that has to be read/written to stop the program execution by the breakpoint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

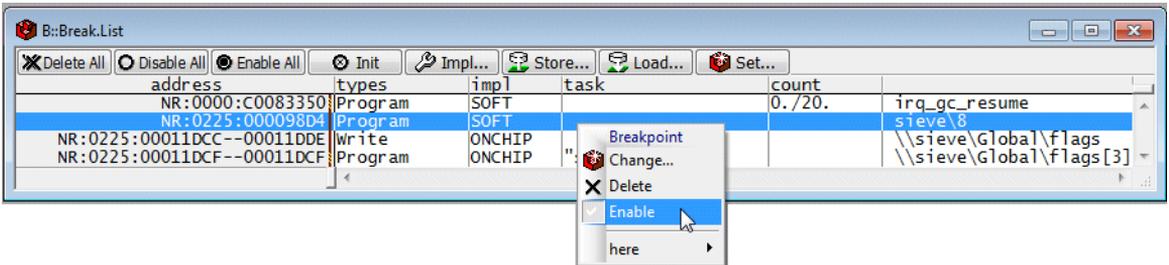
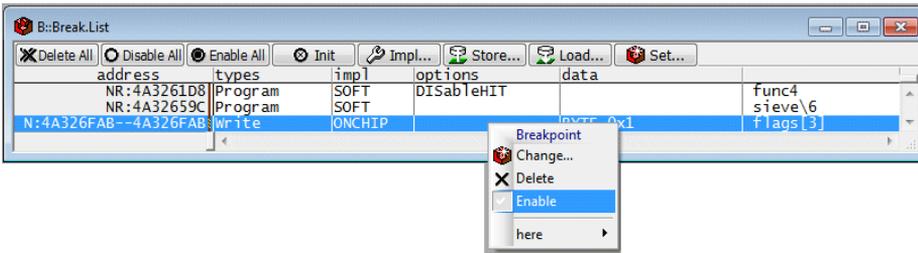
Delete Breakpoints



Break.Delete <address>|<address_range> [/<type>] [/<implem.>] [/<option>] Delete breakpoint

Var.Break.Delete <hll_expression> [/<type>] [/<implem.>] [/<option>] Delete HLL breakpoint

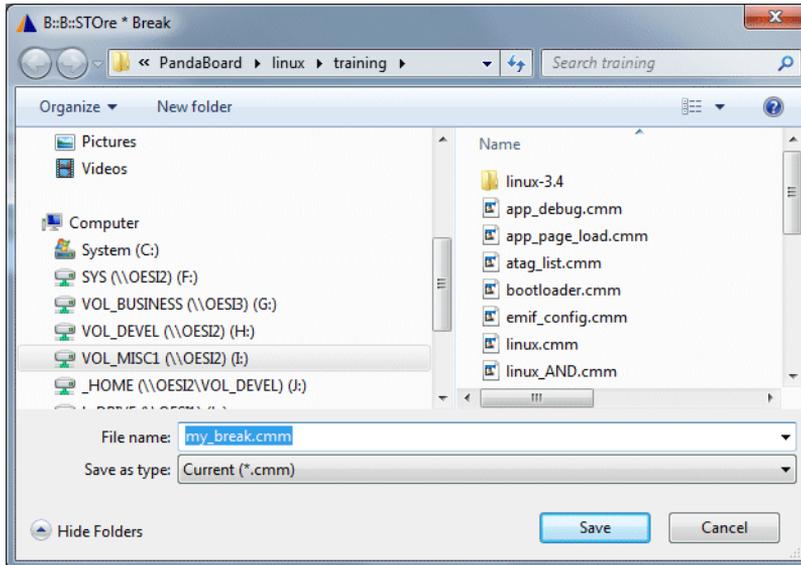
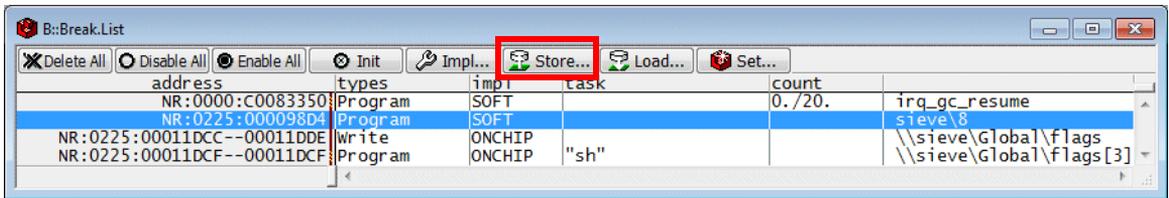
Enable/Disable Breakpoints



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

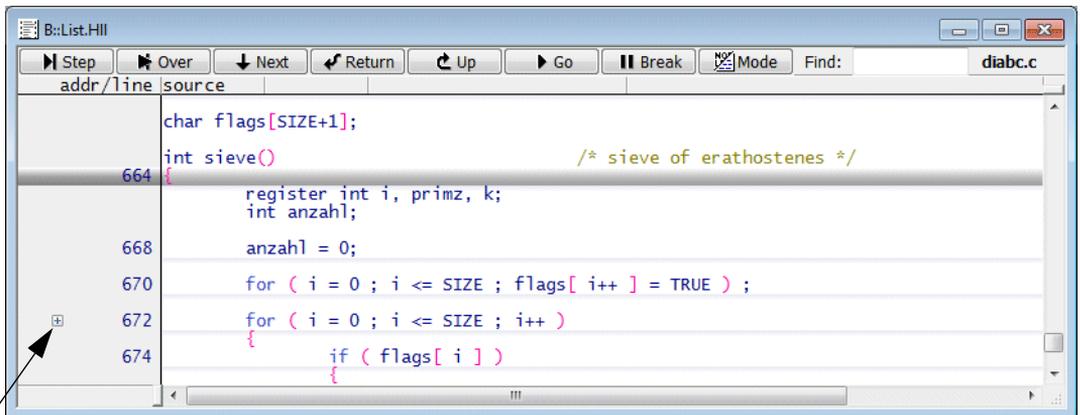
A video tutorial about debugging optimized code can be found here:

support.lauterbach.com/kb/articles/debugging-optimized-code-in-trace32

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.



Drill-down button

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

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

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

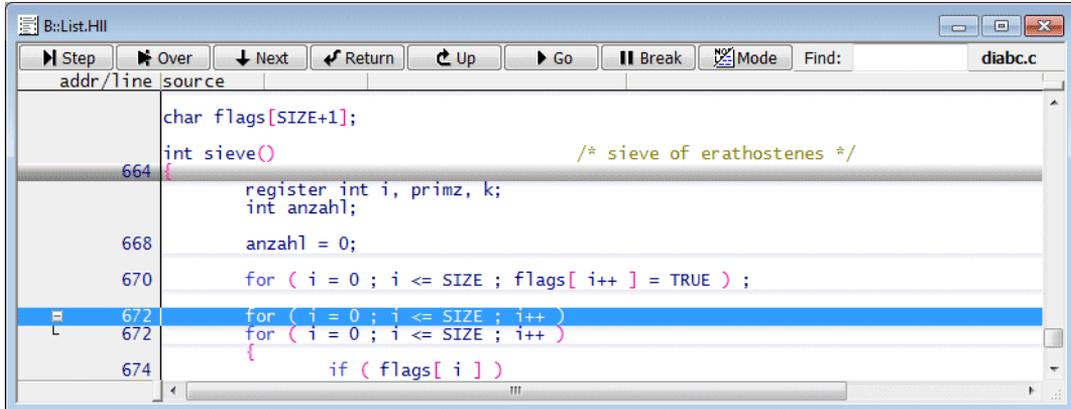
List.Hll

Display source listing, display HLL code lines only.

List.Mix /Track

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

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

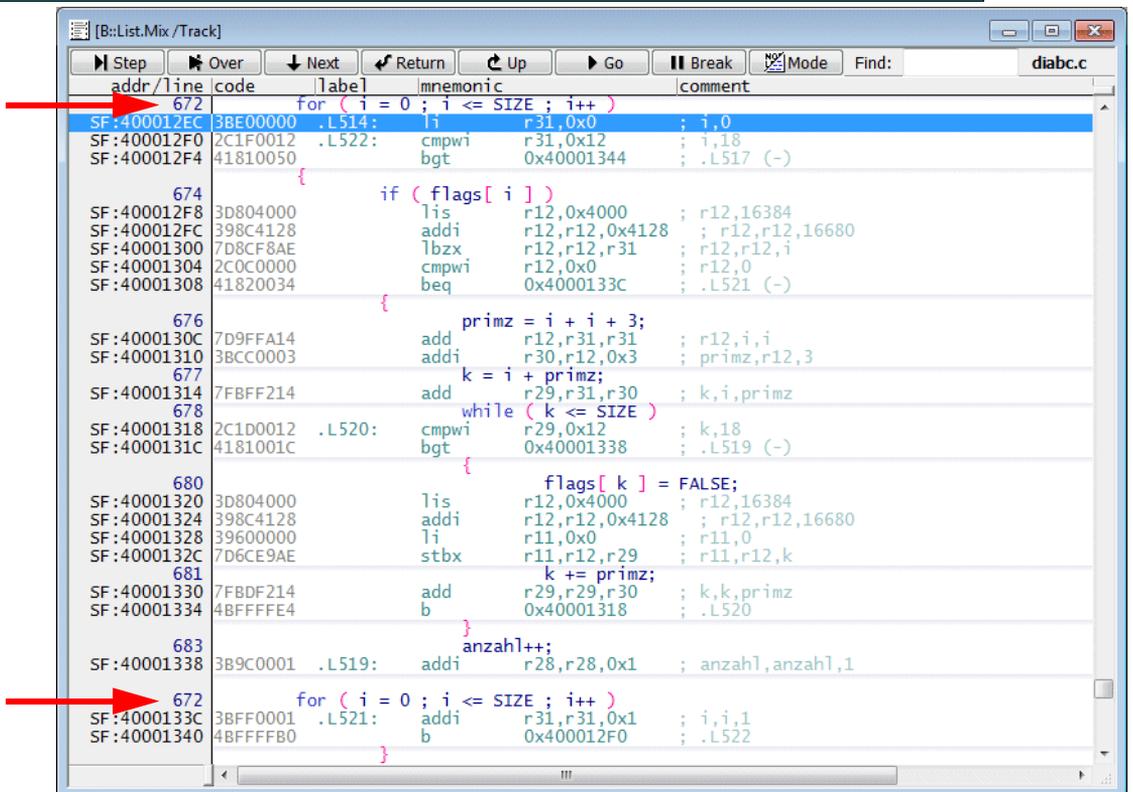


```
char flags[SIZE+1];
int sieve() /* sieve of erathostenes */
{
    register int i, primz, k;
    int anzahl;

    anzahl = 0;

    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

    for ( i = 0 ; i <= SIZE ; i++ )
    for ( i = 0 ; i <= SIZE ; i++ )
    {
        if ( flags[ i ] )
```



addr/line	code	label	mnemonic	comment
672	for (i = 0 ; i <= SIZE ; i++)			
SF:400012EC	3BE00000	.L514:	li	r31,0x0 ; i,0
SF:400012F0	2C1F0012	.L522:	cmpwi	r31,0x12 ; i,18
SF:400012F4	41810050		bgt	0x40001344 ; .L517 (-)
674	if (flags[i])			
SF:400012F8	3D804000		lis	r12,0x4000 ; r12,16384
SF:400012FC	398C4128		addi	r12,r12,0x4128 ; r12,r12,16680
SF:40001300	7D8CF8AE		lbzx	r12,r12,r31 ; r12,r12,i
SF:40001304	2C0C0000		cmpwi	r12,0x0 ; r12,0
SF:40001308	41820034		beq	0x4000133C ; .L521 (-)
676	primz = i + i + 3;			
SF:4000130C	7D9FFA14		add	r12,r31,r31 ; r12,i,i
SF:40001310	3BCC0003		addi	r30,r12,0x3 ; primz,r12,3
677	k = i + primz;			
SF:40001314	7FBFF214		add	r29,r31,r30 ; k,i,primz
678	while (k <= SIZE)			
SF:40001318	2C1D0012	.L520:	cmpwi	r29,0x12 ; k,18
SF:4000131C	4181001C		bgt	0x40001338 ; .L519 (-)
680	flags[k] = FALSE;			
SF:40001320	3D804000		lis	r12,0x4000 ; r12,16384
SF:40001324	398C4128		addi	r12,r12,0x4128 ; r12,r12,16680
SF:40001328	39600000		li	r11,0x0 ; r11,0
SF:4000132C	7D6CE9AE		stbx	r11,r12,r29 ; r11,r12,k
681	k += primz;			
SF:40001330	7FBDF214		add	r29,r29,r30 ; k,k,primz
SF:40001334	4BFFFFE4		b	0x40001318 ; .L520
683	anzahl++;			
SF:40001338	3B9C0001	.L519:	addi	r28,r28,0x1 ; anzahl,anzahl,1
672	for (i = 0 ; i <= SIZE ; i++)			
SF:4000133C	3BFF0001	.L521:	addi	r31,r31,0x1 ; i,i,1
SF:40001340	4BFFFFB0		b	0x400012F0 ; .L522

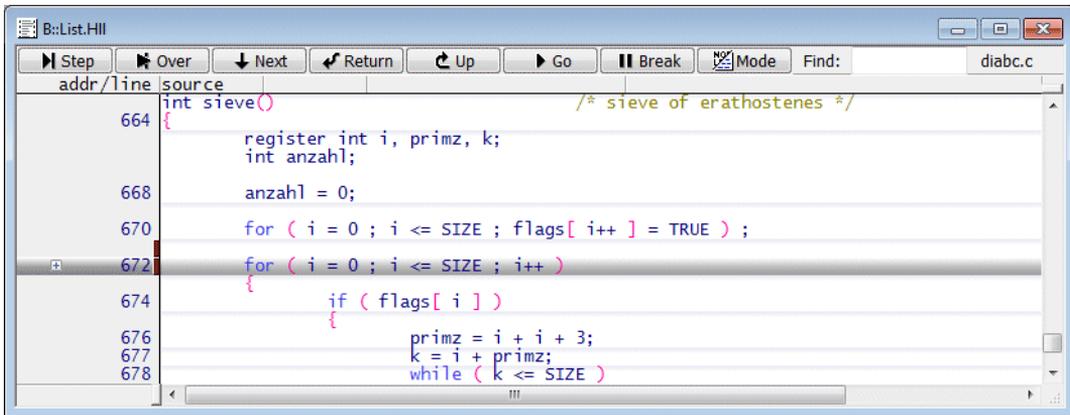
To keep track when debugging optimized code, it is recommended to work with an HLL mode and a MIX mode display of the source listing in parallel.

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.

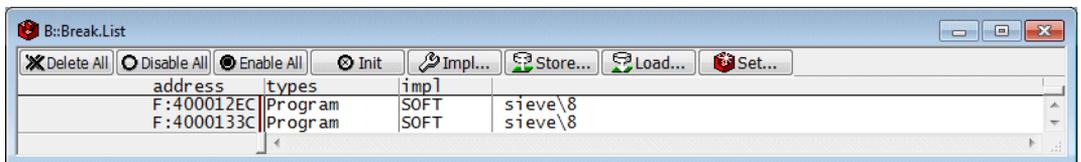


The screenshot shows a debugger window titled "B::List.Hll" with a toolbar containing Step, Over, Next, Return, Up, Go, Break, and Mode buttons. The main area displays C source code for a function named "sieve". The code is as follows:

```
int sieve() /* sieve of erathostenes */
{
    register int i, primz, k;
    int anzahl;

    anzahl = 0;

    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
    for ( i = 0 ; i <= SIZE ; i++ )
    {
        if ( flags[ i ] )
        {
            primz = i + i + 3;
            k = i + primz;
            while ( k <= SIZE )
            {}
        }
    }
}
```

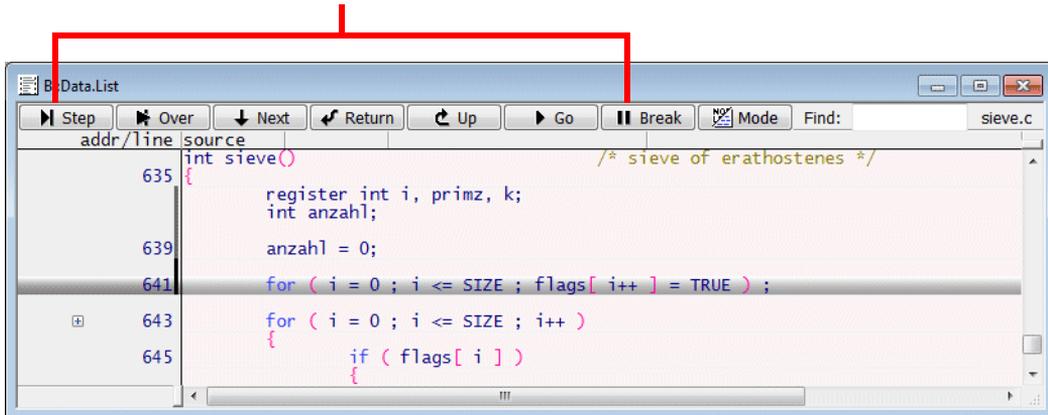


The screenshot shows a debugger window titled "B::Break.List" with a toolbar containing Delete All, Disable All, Enable All, Init, Impl..., Store..., Load..., and Set... buttons. The main area displays a table of breakpoints:

address	types	impl	
F:400012EC	Program	SOFT	sieve\8
F:4000133C	Program	SOFT	sieve\8

Basic Debug Control

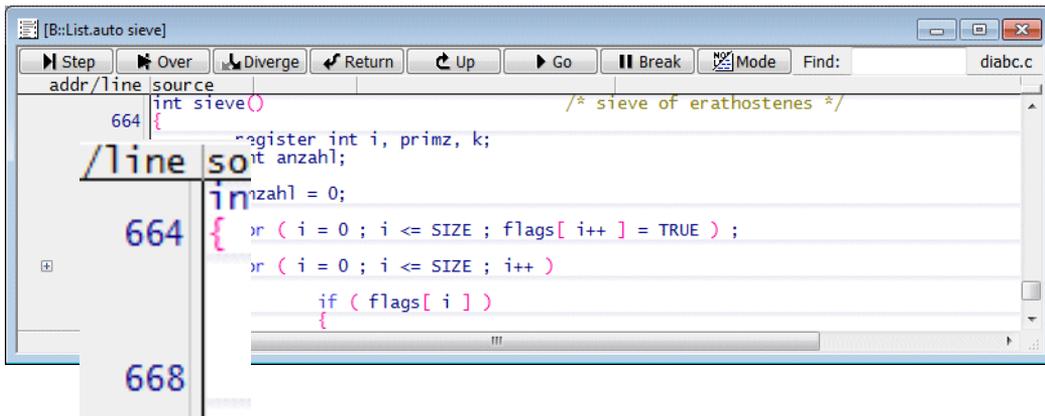
There are local buttons in the **Data.List** window for all basic debug commands



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.

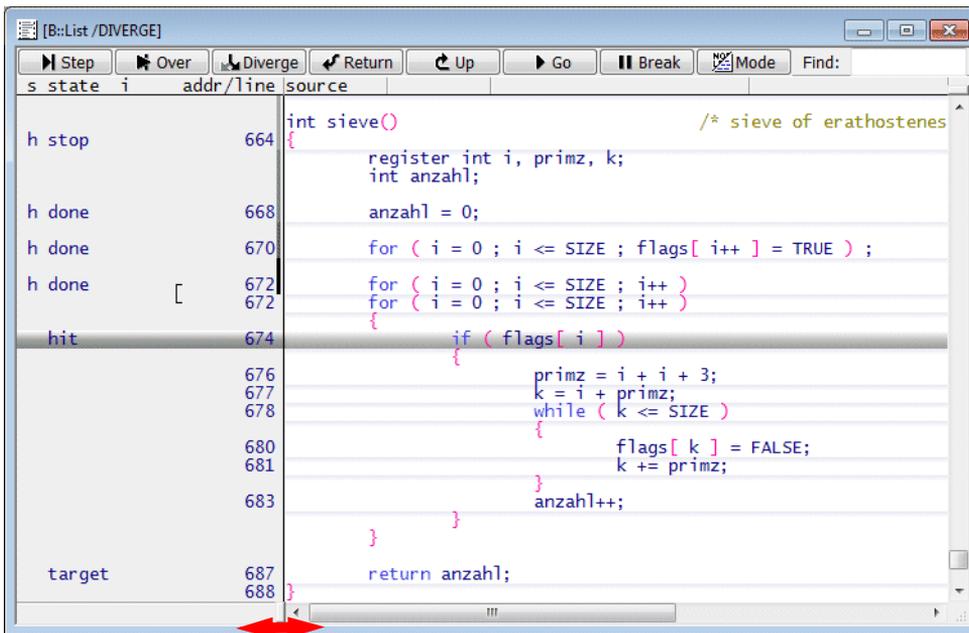
More details on Step.Diverge

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.

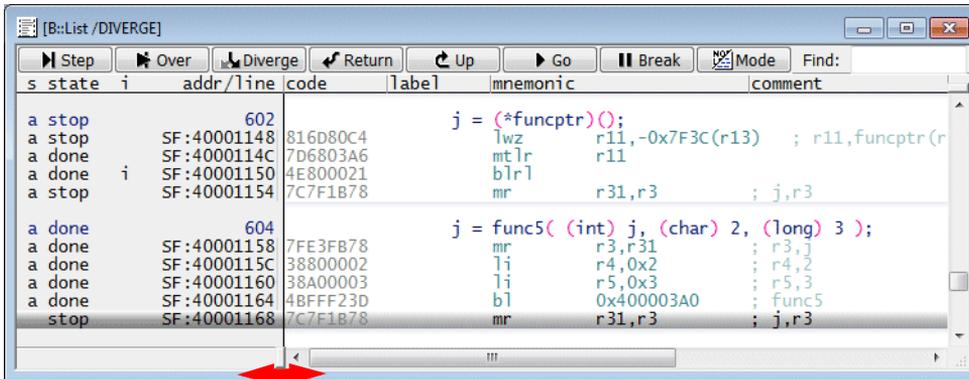


The following command allows you to get more details:

```
List.auto /DIVERGE
```



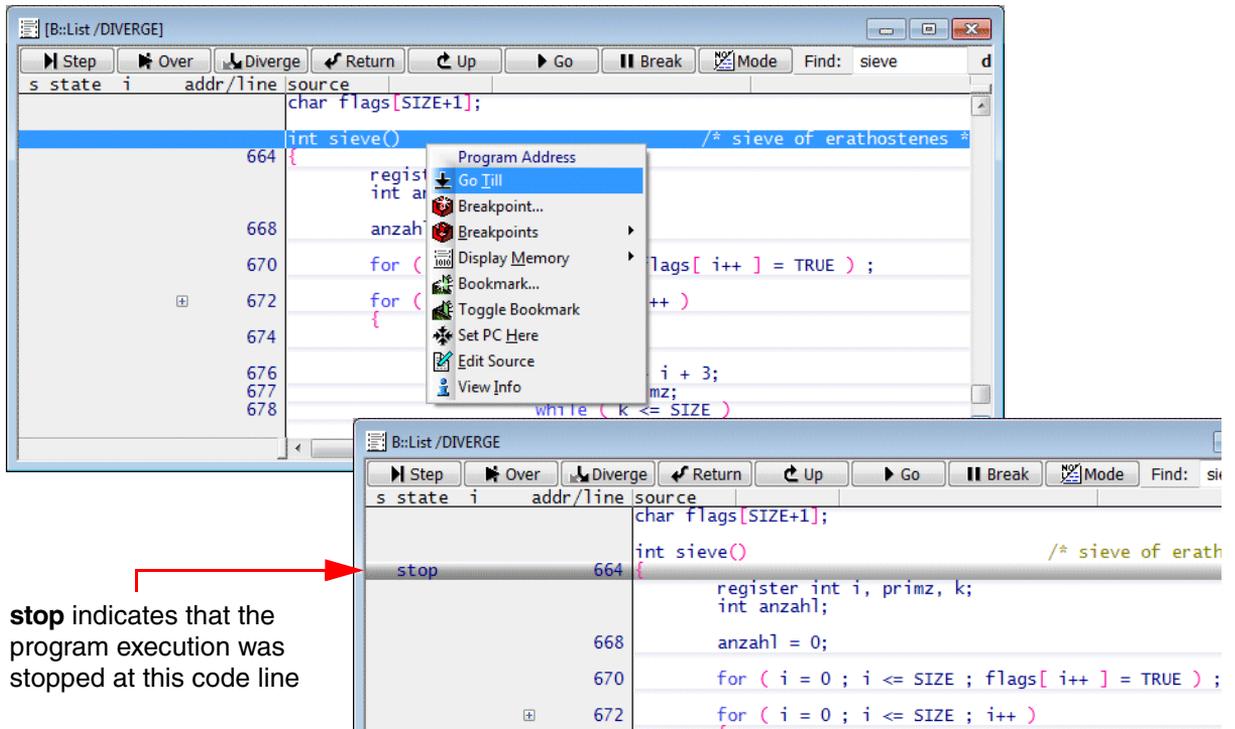
Drag this handle to see the DIVERGE details



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

Example 1: Diverge through function sieve.

1. Run program execution until entry to function sieve.

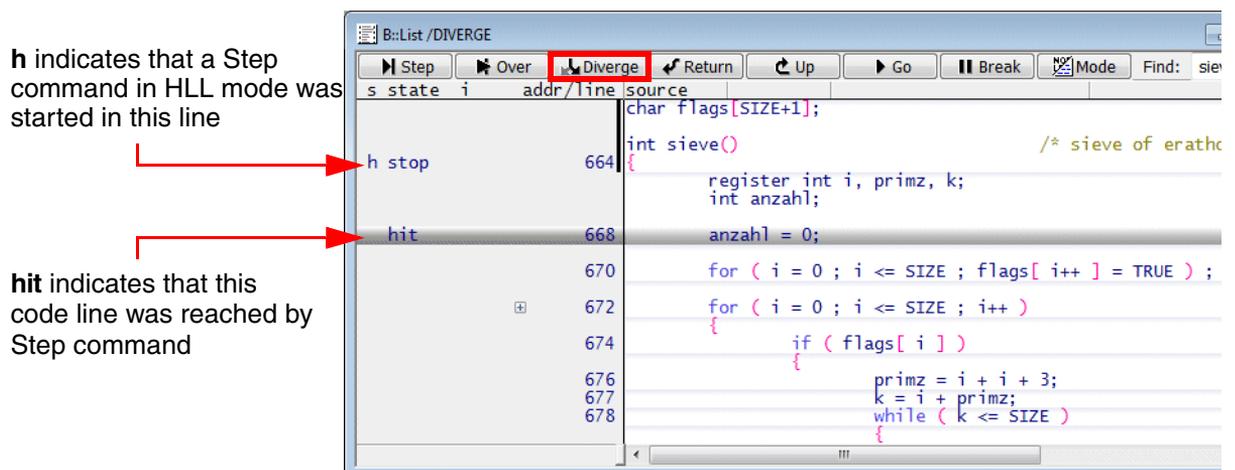


The screenshot shows a debugger window titled "B::List / DIVERGE". The main window displays the source code of a function `sieve()` starting at line 664. A context menu is open over line 664, with "Go To" selected. A second window below shows the same code, but with a "stop" command entered in the "s state" column next to line 664. A red arrow points from the text "stop indicates that the program execution was stopped at this code line" to the "stop" command.

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

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

2. Start a Step.Diverge command.



The screenshot shows the same debugger window. The "Step.Diverge" button in the toolbar is highlighted with a red box. The "s state" column now shows "h stop" next to line 664 and "hit" next to line 668. Red arrows point from the text "h indicates that a Step command in HLL mode was started in this line" to "h stop" and from "hit indicates that this code line was reached by Step command" to "hit".

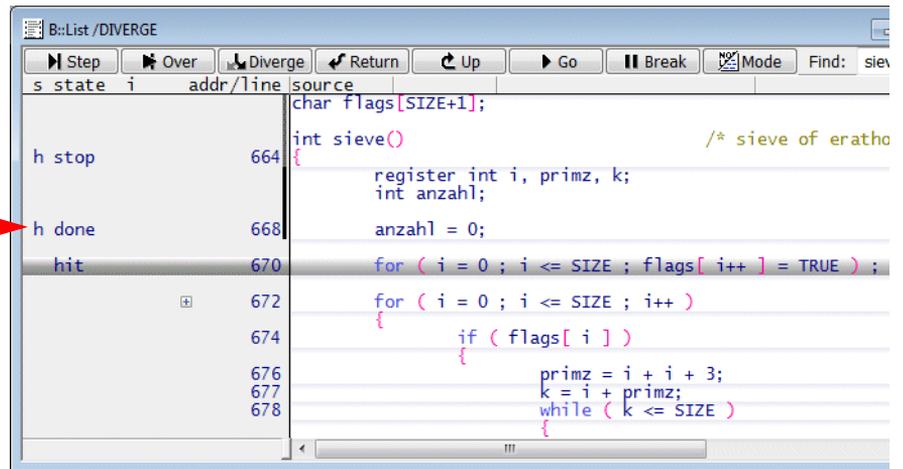
```
char flags[SIZE+1];
int sieve() /* sieve of erathostenes */
{
    register int i, primz, k;
    int anzahl;
    anzahl = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
    for ( i = 0 ; i <= SIZE ; i++ )
    {
        if ( flags[ i ] )
        {
            primz = i + i + 3;
         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

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 screenshot shows a debugger window titled "B::List / DIVERGE". The interface includes a toolbar with buttons for Step, Over, Diverge, Return, Up, Go, Break, and Mode. Below the toolbar is a table with columns for state, address/line, and source code. The source code is a C function named "sieve()".

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

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

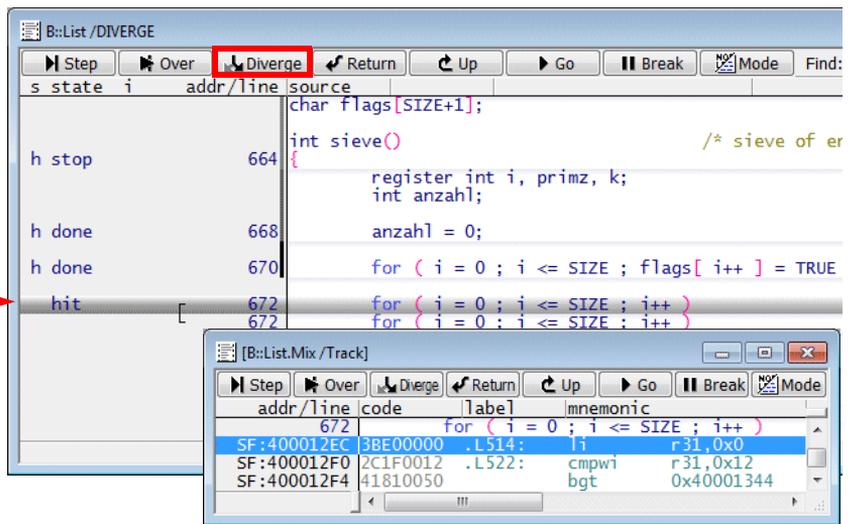
s	state	i	addr/line	source
			664	char flags[SIZE+1];
	h stop		664	int sieve() /* sieve of eratho
			668	{ register int i, primz, k;
	h done		668	int anzahl;
			670	anzahl = 0;
	hit		670	for (i = 0 ; i <= SIZE ; flags[i++] = TRUE) ;
			672	for (i = 0 ; i <= SIZE ; i++)
			674	{ if (flags[i])

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**

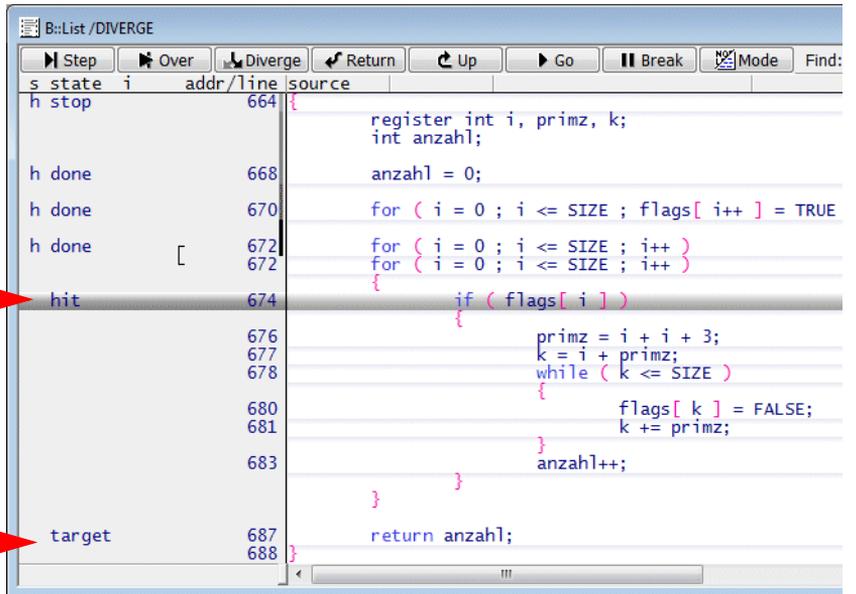
s	state	i	addr/line	source
			664	char flags[SIZE+1];
	h stop		664	int sieve() /* sieve of erathos
			668	{ register int i, primz, k;
			668	int anzahl;
	h done		668	anzahl = 0;
	h done		670	for (i = 0 ; i <= SIZE ; flags[i++] = TRUE) ;
	hit		672	for (i = 0 ; i <= SIZE ; i++)
			672	for (i = 0 ; i <=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

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**

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

7. Continue with Step.Diverge.

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

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

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

```

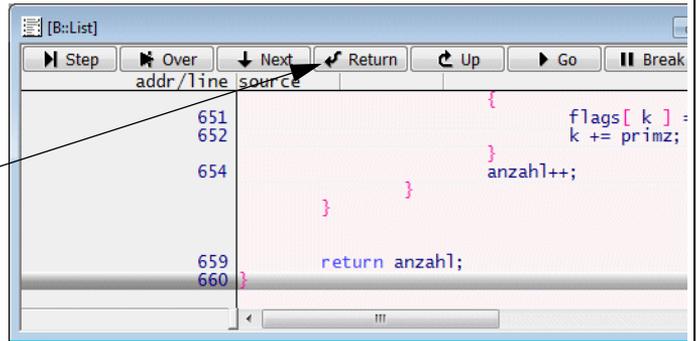
B::
no more reachable targets from this code
trigger devices trace Data Var List
SF:40001284 \\diabc\diabc\main+0x228
    
```

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

Return

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

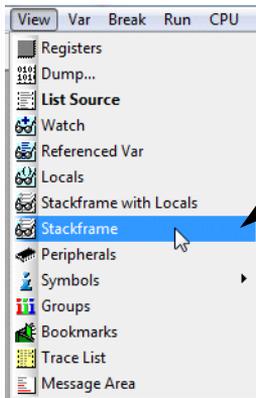
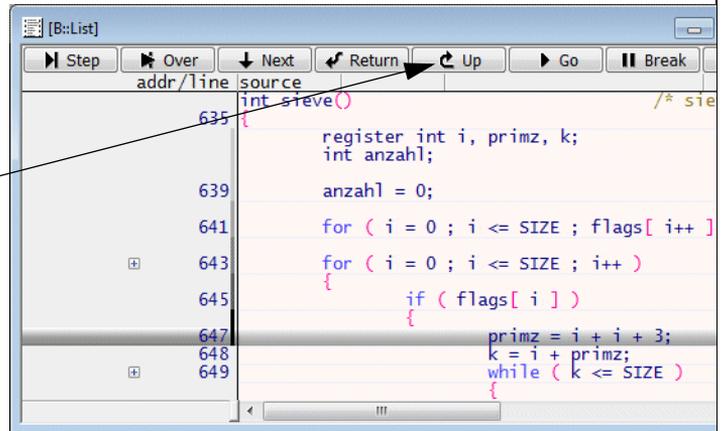
After pressing **Return** the program execution is stopped at the last instruction of the function



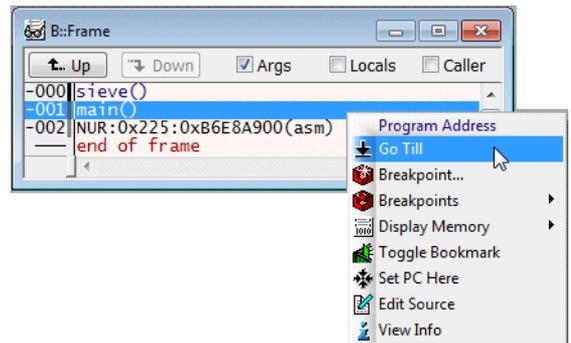
Up

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

Press **Up** to return to the function that called the current function



Display the HLL stack to see the function nesting



Performed on the currently selected core if single stepping is performed on assembler level. Otherwise all cores are executing code.

Step [<i><count></i>]	Single step
Step.Change <i><expression></i>	Step until <i><expression></i> changes
Step.Till <i><condition></i>	Step until <i><condition></i> becomes true, <i><condition></i> written in TRACE32 syntax
Var.Step.Change <i><hll_expression></i>	Step until <i><hll_expression></i> changes
Var.Step.Till <i><hll_condition></i>	Step until <i><hll_condition></i> becomes true, <i><hll_condition></i> 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
```

All core are executing code, when one of the following commands is used.

Step.Over	Step over call
------------------	----------------

Go [<i><address></i> <i><label></i>]	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 [<i><level></i> <i><address></i>]	Run program until it returns to the caller function