LAUTERBACH
DEVELOPMENT TOOLS

Training Basic SMP Debugging for Intel® x86/x64

MANUAL

Training Basic SMP Debugging for Intel® x86/x64

Version 06-Jun-2024

Debug Configurations

An Intel® x86/x64 chip can provide the following debug features:

- **Extended debugging**
- **System trace**

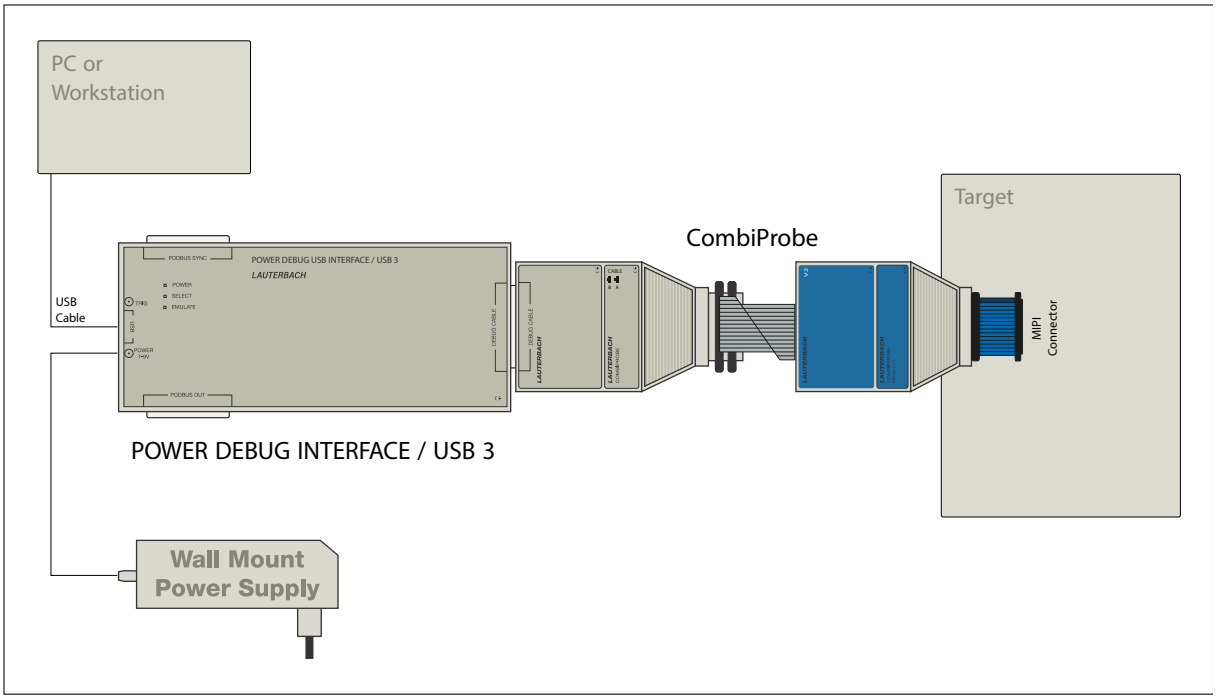
  A system trace provides visibility of various events/states inside the chip. Trace data can be generated by instrumented application code and/or by hardware modules within the chip.
- **Core traces**

  A core trace provides detailed visibility of the program execution on a core. Trace data are generated for the instruction execution sequence.

MIPI60-Cv2 Configuration

A TRACE32 configuration consists of:

- Universal debugger hardware e.g. *PowerDebug Module USB 3.0*
- *CombiProbe 2 Intel® x86/x64 MIPI60-Cv2*



Deprecated module:

- *CombiProbe Intel® x86/x64 MIPI60-C*
- *Debug Cable for Intel® x86/x64 XDP60*

The features of the CombiProbe 2 MIPI60-Cv2 can be derived from the connected pins:

Signal	Pin	Pin	Signal
VREF_DEBUG	1	2	TMS
TCK0	3	4	TDO
TDI	5	6	Reset Out
PMODE/Reset In	7	8	No Connect
TRST_N	9	10	PREQ_N
PRDY_N	11	12	VREF_TRACE
PTI_0_CLK	13	14	PTI_1_CLK
GND	15	16	GND
GND	17	18	PTI_1_DATA[0]
PTI_0_DATA[0]	19	20	PTI_1_DATA[1]
PTI_0_DATA[1]	21	22	PTI_1_DATA[2]
PTI_0_DATA[2]	23	24	PTI_1_DATA[3]
PTI_0_DATA[3]	25	26	No Connect
PTI_0_DATA[4]	27	28	No Connect
PTI_0_DATA[5]	29	30	No Connect
PTI_0_DATA[6]	31	32	No Connect
PTI_0_DATA[7]	33	34	No Connect
No Connect	35	36	Boot Stall
No Connect	37	38	CPU Boot Stall
No Connect	39	40	Power Button
No Connect	41	42	PWRGOOD
No Connect	43	44	No Connect
No Connect	45	46	No Connect
No Connect	47	48	I2C_SCL
No Connect	49	50	I2C_SDA
TCK1	51	52	reserved by TRACE32
HOOK[9]	53	54	DBG_UART_TX
HOOK[8]	55	56	DBG_UART_RX
GND	57	58	GND
No Connect	59	60	No Connect

- **Standard JTAG**
- **PRDY/PREQ**
  - PREQ: allows the debugger to stop the core(s)
  - PRDY: signals the debugger that the core(s) stopped
- **HOOK pins**
  - PWRGOOD: VTREF + PWRGOOD indicate that the JTAG power domain is powered.
  - Power Button: allows the debugger to control the target power (command **SYStem.POWER**).
  - CPU Boot Stall: allows the debugger to stop in CPU boot stall mode after power on. For details refer to the command **TrOnchip.Set CpuBootStall**.
  - Boot Stall: allows the debugger to stop in boot stall mode after power on. For details refer to the command **TrOnchip.Set BootStall**.
  - Reset In: allows the debugger to reset the platform/SOC/cores (command **SYStem.Mode Go**).
  - Reset Out: signals the debugger that the platform was reset.
- **PTI_0**

  8-bit System Trace, can be selected with command **CAnalyzer.TracePORT TracePortA**.
- **PTI_1**

  4-bit System Trace, can be selected with command **CAnalyzer.TracePORT TracePortB**.
- **UART**

  Function currently not specified.
- **I2C**

  Allows the debugger to control I2C bus (command group **I2C**).

On-Chip Core Trace

Core trace information can be generated and routed to SDRAM. Details on the features and the tool configuration for core tracing are provided by **"Intel® Processor Trace Training"** (training_ipt_trace.pdf).

Off-Chip System/Core Trace

Core trace information can be generated and routed to a System Trace Module. The STM merges the core trace information with the system trace information and exports it via 8/16 trace data pins. Details on the features and the tool configuration for off-chip core tracing are provided by **"Intel® Processor Trace Training"** (training_ipt_trace.pdf).

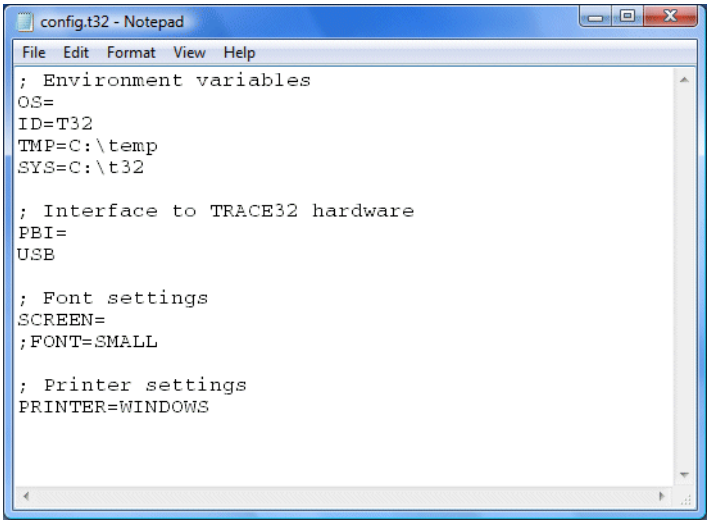
Basic TRACE32 PowerView Parameters

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

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

Configuration File

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



The following rules apply to the configuration file:

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

Parameter	Syntax	Description
Host interface	PBI= *<host_interface>*	Host interface type of TRACE32 tool hardware (USB or ethernet)
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	The 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

```
; 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                 ; standard Windows printer can be
                                ; used from TRACE32 PowerView
```

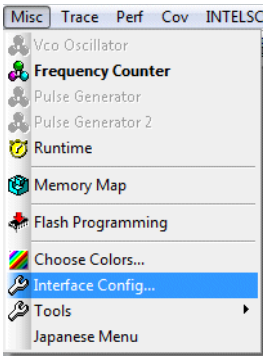
Remote Control for POWER DEBUG INTERFACE / USB

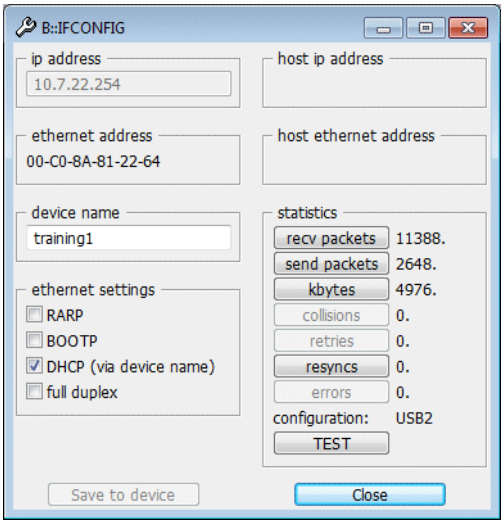
TRACE32 allows to communicate with a POWER DEBUG INTERFACE USB from a remote PC. For an example, see **"Example: Remote Control for POWER DEBUG INTERFACE / USB"** in TRACE32 Installation Guide, page 48 (installation.pdf).

Configuration File for Ethernet

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

; 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                 ; standard Windows printer can be
                                ; used from TRACE32 PowerView
```

Ethernet Configuration and Operation Profile



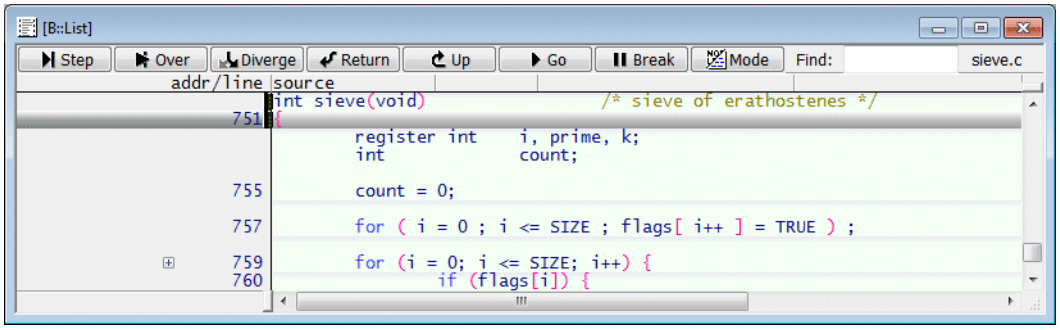


IFCONFIG Display and change information for the Ethernet interface

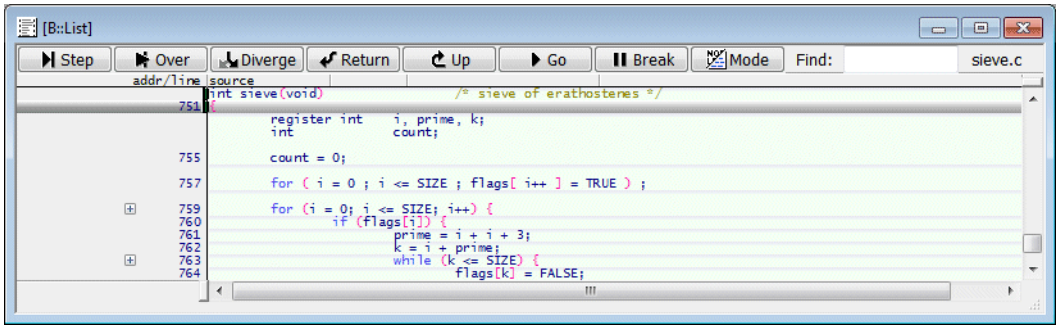
Changing the font size can be helpful for a more comfortable display of TRACE32 windows.

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

Display with normal fonts:

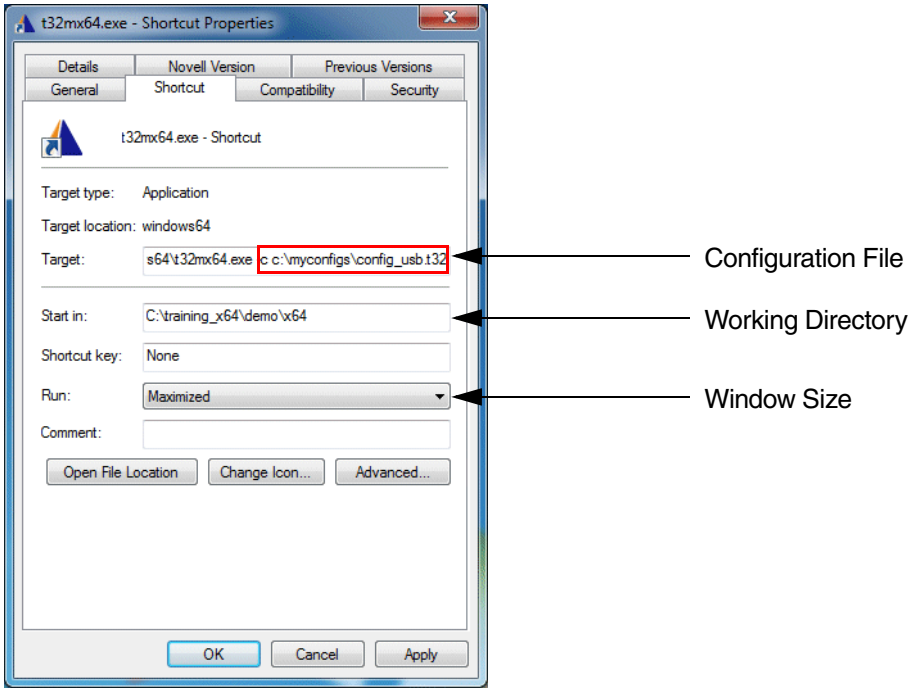


Display with small fonts:



Application Properties (Windows only)

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



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:\training_x64\bin\windows64\t32mx64.exe -c j:\and\config_debug.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

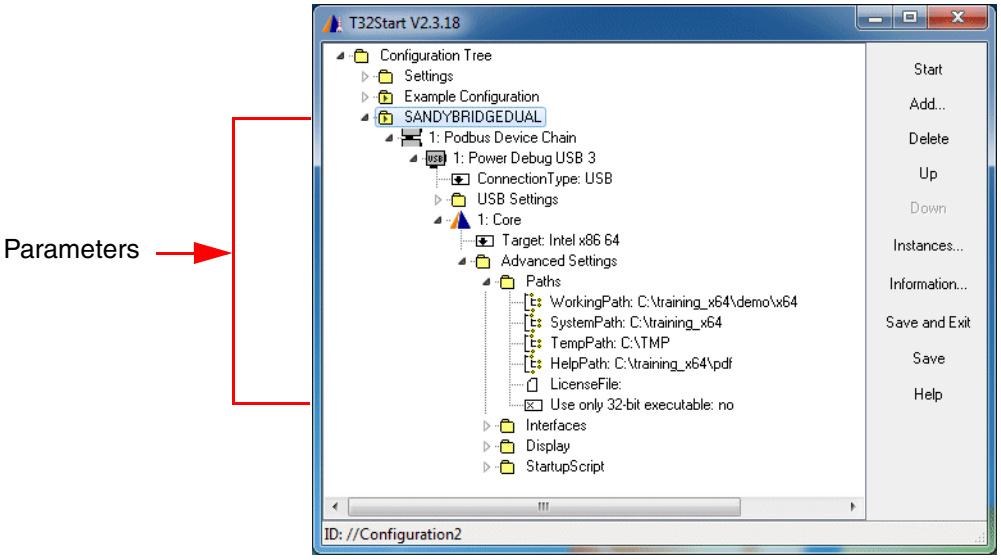
Definition of the Window Size for TRACE32 PowerView

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**.

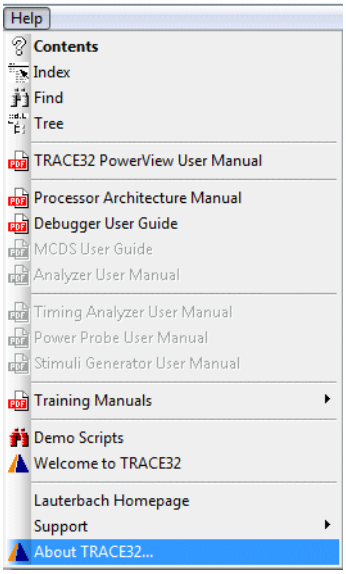
A detailed online help for **t32start.exe** is available via the **Help** button or in **"T32Start"** (app_t32start.pdf).

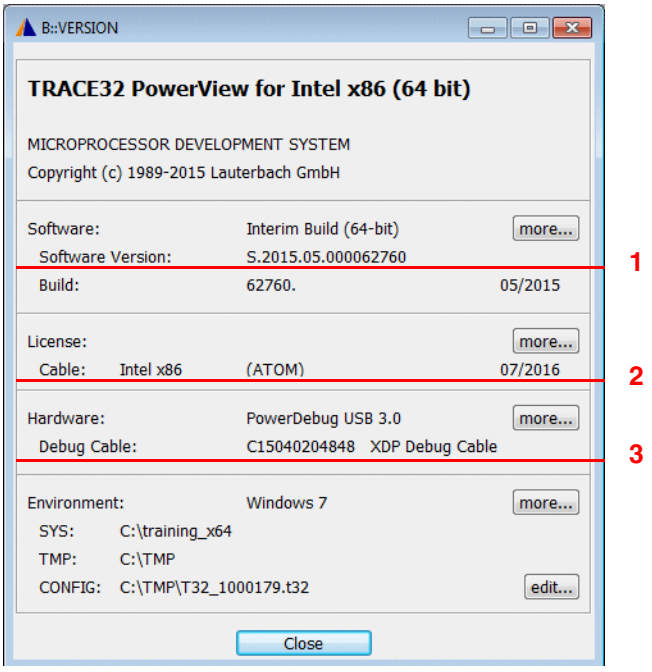


About TRACE32

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

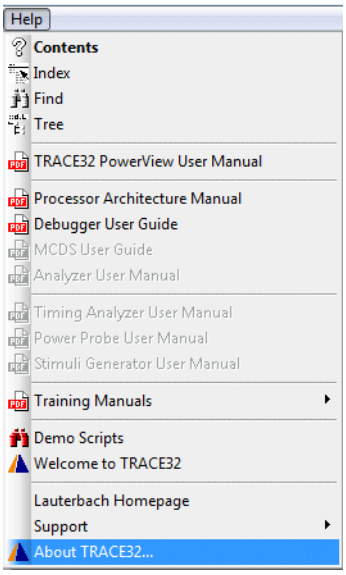
Version Information (Debug Cable)

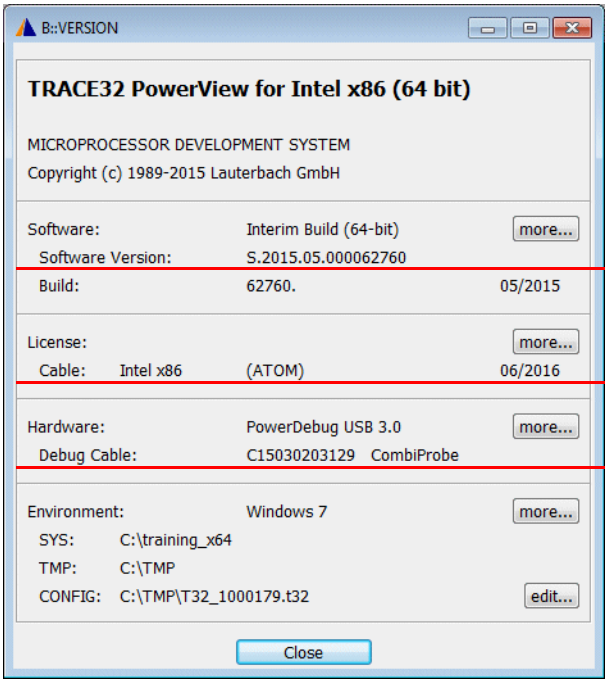




The VERSION window informs you about:

1. the version of the TRACE32 software.
2. the debug license(s) programmed into the debug cable, the expiration date of your software guarantee respectively the expiration date of your software warranty.
3. the serial number of the debug cable.





The VERSION window informs you about:

1. the version of the TRACE32 software
2. the debug license(s) programmed into the CombiProbe, the expiration date of your software guarantee respectively the expiration date of your software warranty.
3. the serial number of the CombiProbe.

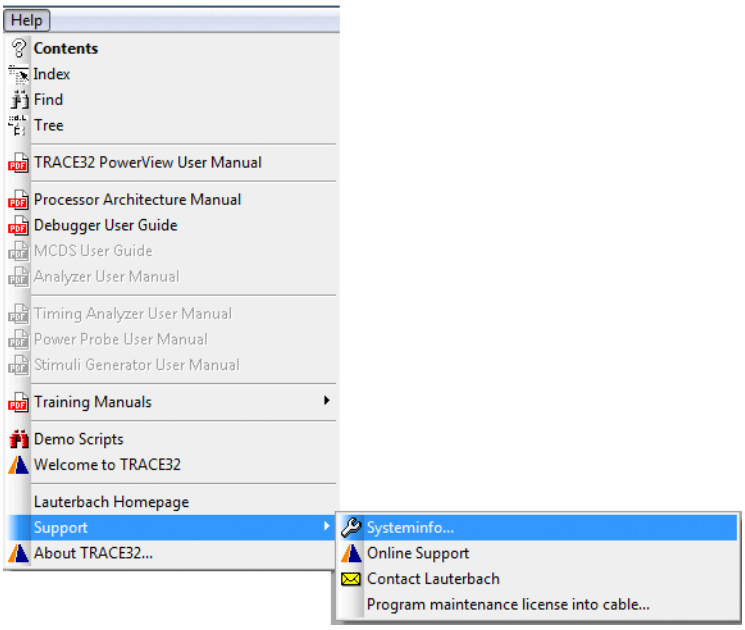
Command summary

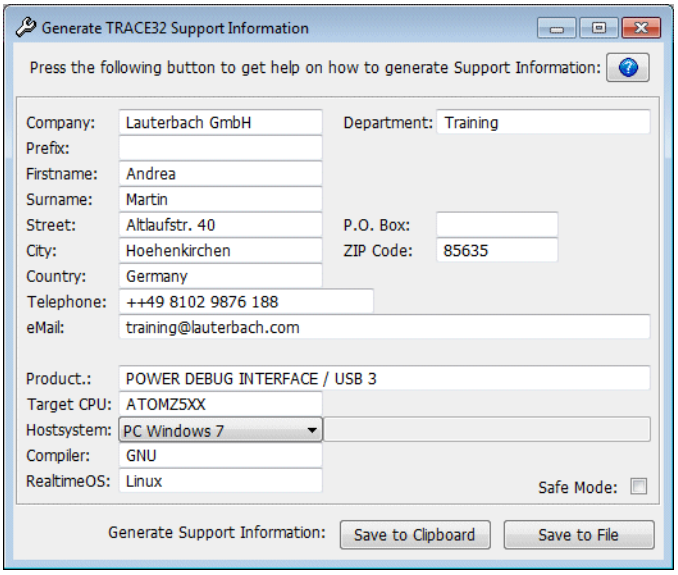
VERSION.view	Display the VERSION window.
VERSION.HARDWARE	Display more details about the TRACE32 hardware modules.
VERSION.SOFTWARE	Display more details about the TRACE32 software.

Prepare Full Information for a Support Email

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

1. To generate a TRACE32 information report, choose **Help** > **Support** > **Systeminfo ….**





2. Preferred: click **Save to File**, and send the information as an attachment to your e-mail.

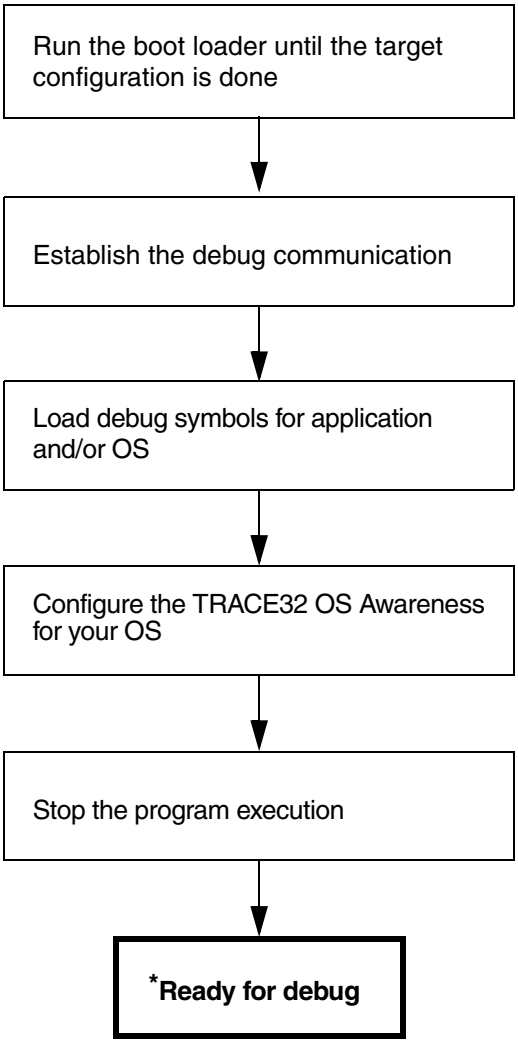
3. Click **Save to Clipboard**, and then paste the information into your e-mail.

Establish your Debug Session

Before you can start debugging, the debug environment has to be set up.

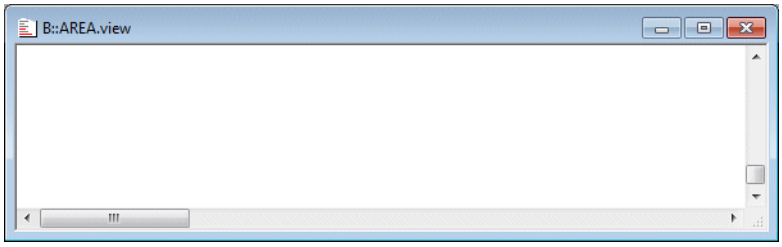
Course of Action

The setup procedure described on the following pages assumes that the application (and/or the operating system) under debug are running out of RAM and a ready-to-run boot loader configures the target system and especially the RAM for this debug scenario.



*Considering the circumstance that a process has to be started manually e.g. via a **TERMinal** window

An AREA window can be opened to monitor the start-up process.



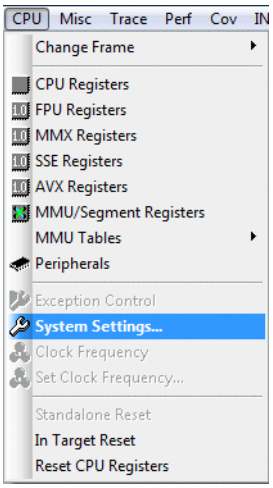
AREA.view	Open an AREA window

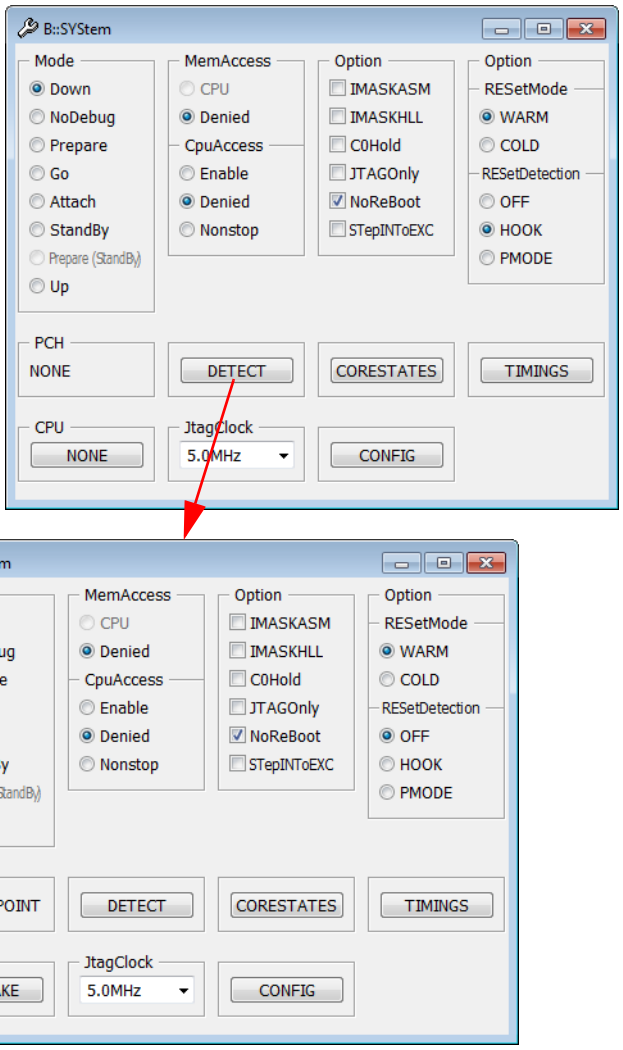
Run the Boot Loader until the Target Configuration is Done

When the target reset is released the boot loader starts to configure the target.

Establish the Debug Communication

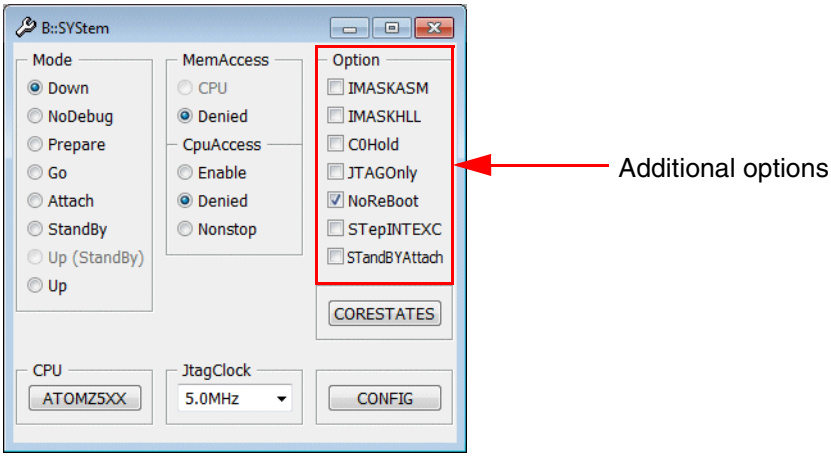
Before the debug communication can be established, the debugger needs to know the target chip. The recommended method is to use the auto detection feature:



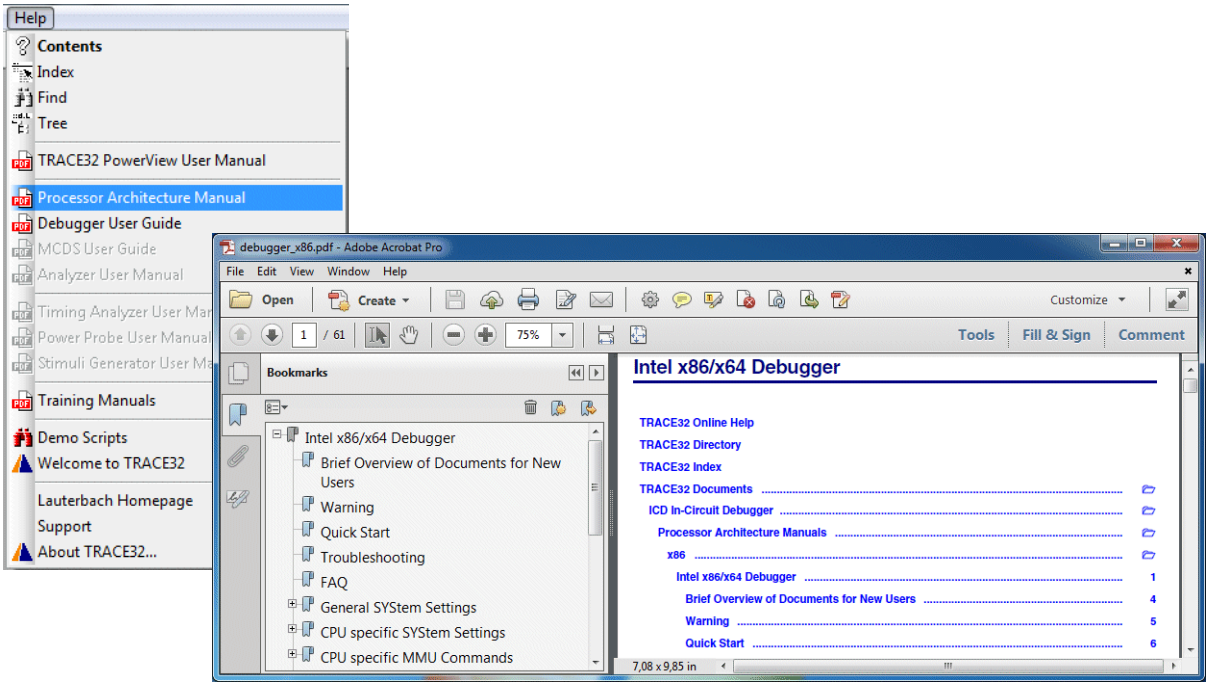



```
SYStem.DETECT TARGET
```

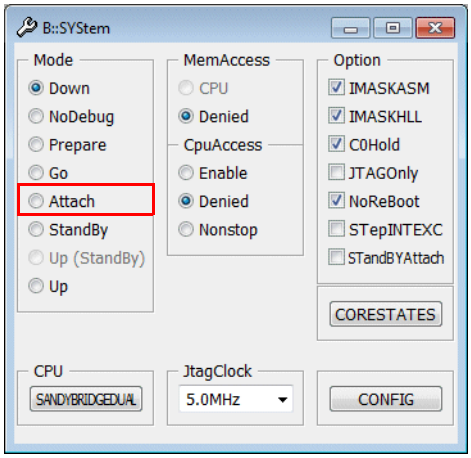
Then the options required for your chip have to be set.



For details on the available options, refer to **"Intel® x86/x64 Debugger"** (debugger_x86.pdf).

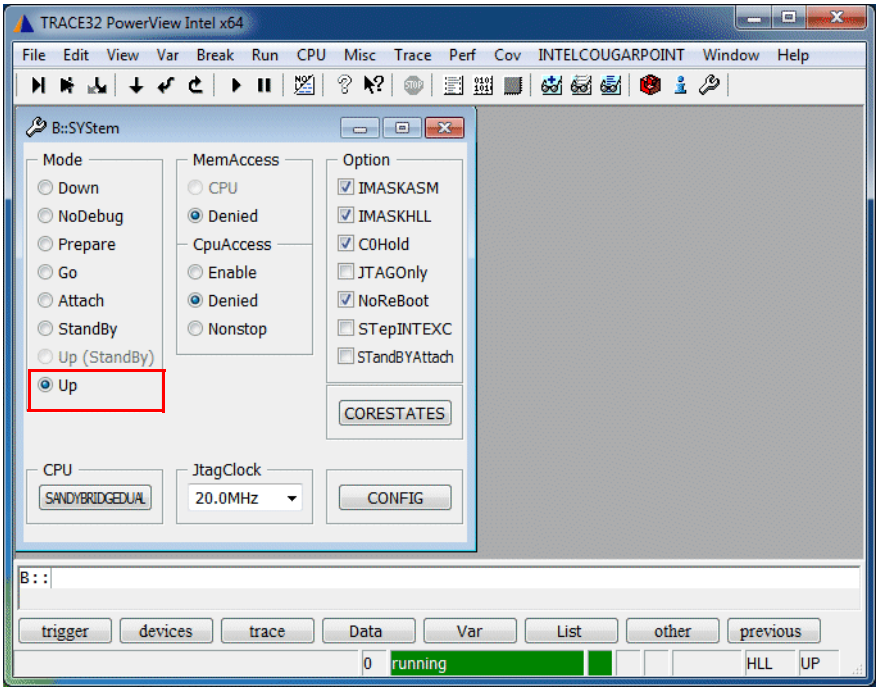


Choose the **Attach** radio button to establish the communication between the debugger and the target chip.



TRACE32 PowerView selects the radio button **Up** to indicate that the communication between the debugger and the target chip is established.

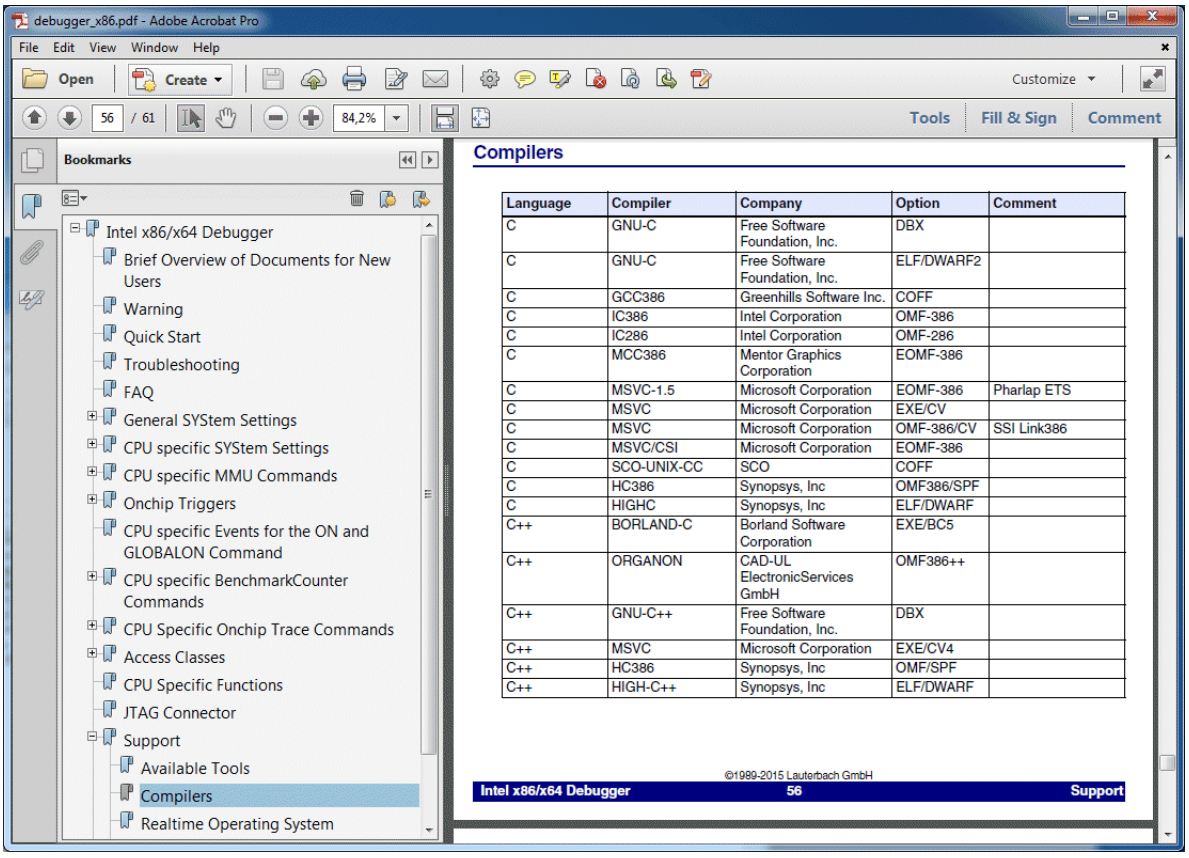
The **running** in Debug field of the TRACE32 State Line indicates that the boot loader is still running.



SYStem.Attach	Establish the communication between the debugger and the target chip.

Alternative way to establish the communication between the debugger and the target chip might be available depending on the used platform and the used TRACE32 debug tool. For details refer to **SYStem.Mode**.

TRACE32 supports a wide range of compilers and compiler output formats. Refer to the **Compilers** section of your **Processor Architecture Manual** for details.



Data.LOAD.*<sub_cmd>* *<file>* **/NoCODE** [*/<option>*]

```
; Load debug symbols from ELF file sieve_x86.elf
Data.LOAD.Elf sieve_x86.elf /NoCODE

; Load debug symbols from ELF file
; open file browser to select file
Data.LOAD.Elf * /NoCODE
```

A in-depth introduction to the **Data.LOAD** command is given in the chapter **"Load the Application Program"** in Training HLL Debugging, page 4 (training_hll.pdf).

Configure the TRACE32 OS Awareness for Your OS

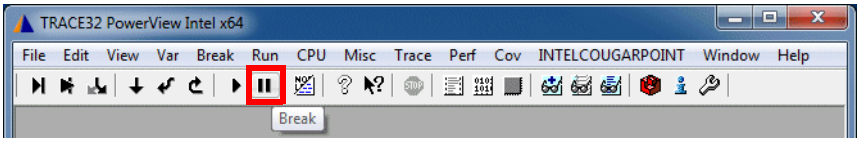
Please refer to **"Training Linux Debugging for Intel® x86/x64"** (training_rtos_linux_x86.pdf) on how to activate the TRACE32 Linux awareness on your target.

Please refer to **"OS Awareness Manual Windows Standard"** (rtos_windows.pdf) on how to activate the TRACE32 Windows awareness on your target.

If you use a different OS refer to the corresponding target **OS Awareness Manual** (rtos_*<os>*.pdf).

Stop the Program Execution

The program execution can be stopped by pushing the **Break** button.



Break	Stop the program execution

Start-Up Script

It is strongly recommended to summarize the commands, that are used to set up the debug environment, in a start-up script. The script language PRACTICE is provided for this purpose.

The standard extension for a script file is `.cmm`.

Write a Start-Up Script

The debugger provides an ASCII editor, that allows to write, to run and to debug a start-up script.

PEDIT *<file>*	Create *<file>* and open it with the script editor

```
PEDIT my_startup
```

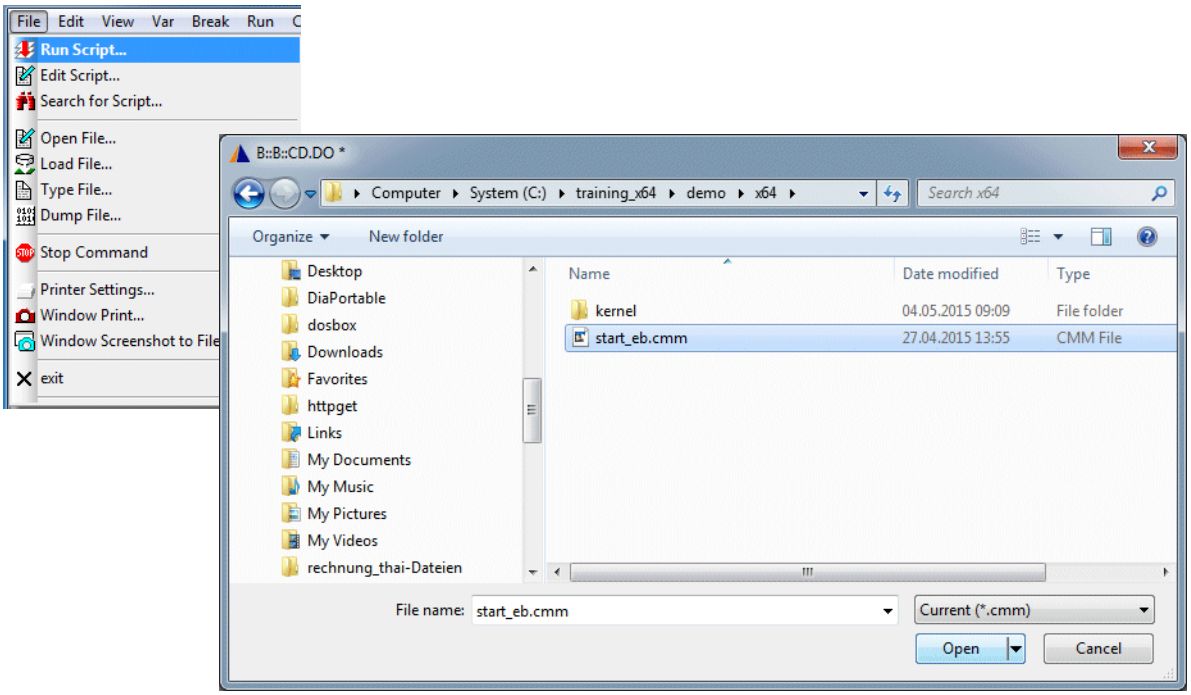
The debugger provides two commands, that allow you to convert debugger configuration information to a script.

STOre *<file> {<item>}*	Generate a script that allows to reproduce the current settings
ClipSTOre *{<item>}*	Generate a command list in the clip-text that allows to reproduce the current settings

```
STOre system_settings SYStem          ; Generate a script that allows you
                                      ; to reproduce the settings of the
                                      ; SYStem window at any time

PEDIT system_settings                 ; Open the file system_settings
```

```
ClipSTOre SYStem                      ; Generate a command list that
                                      ; allows you to reproduce the
                                      ; settings of the SYStem window
                                      ; at any time
                                      ; The generated command list can be
                                      ; pasted in any editor
```

Run a Start-up Script



DO *<filename>* Run a start-up script

```
DO start_eb
```

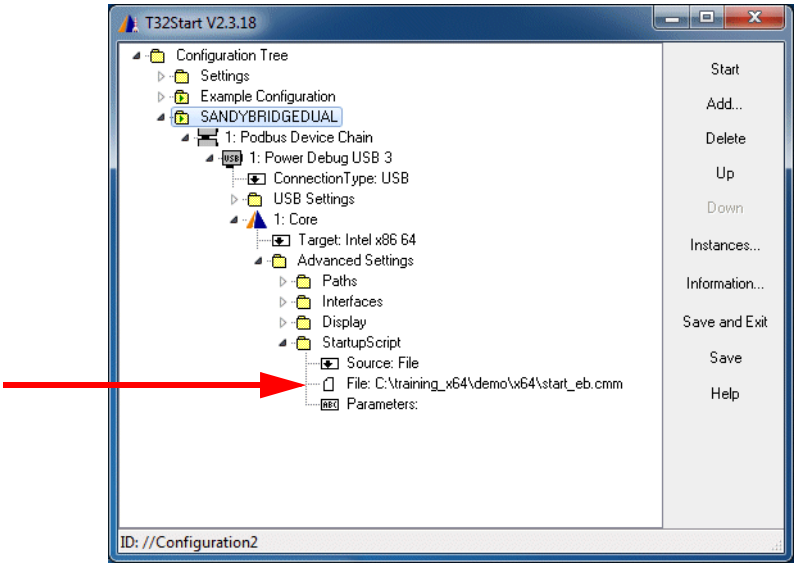
There a two ways to define a start-up script, that is automatically started, when the debugger is started.

1. Define start-up script in conjunction with the executable.

The debugger-executable can be started with the start-up script as parameters.

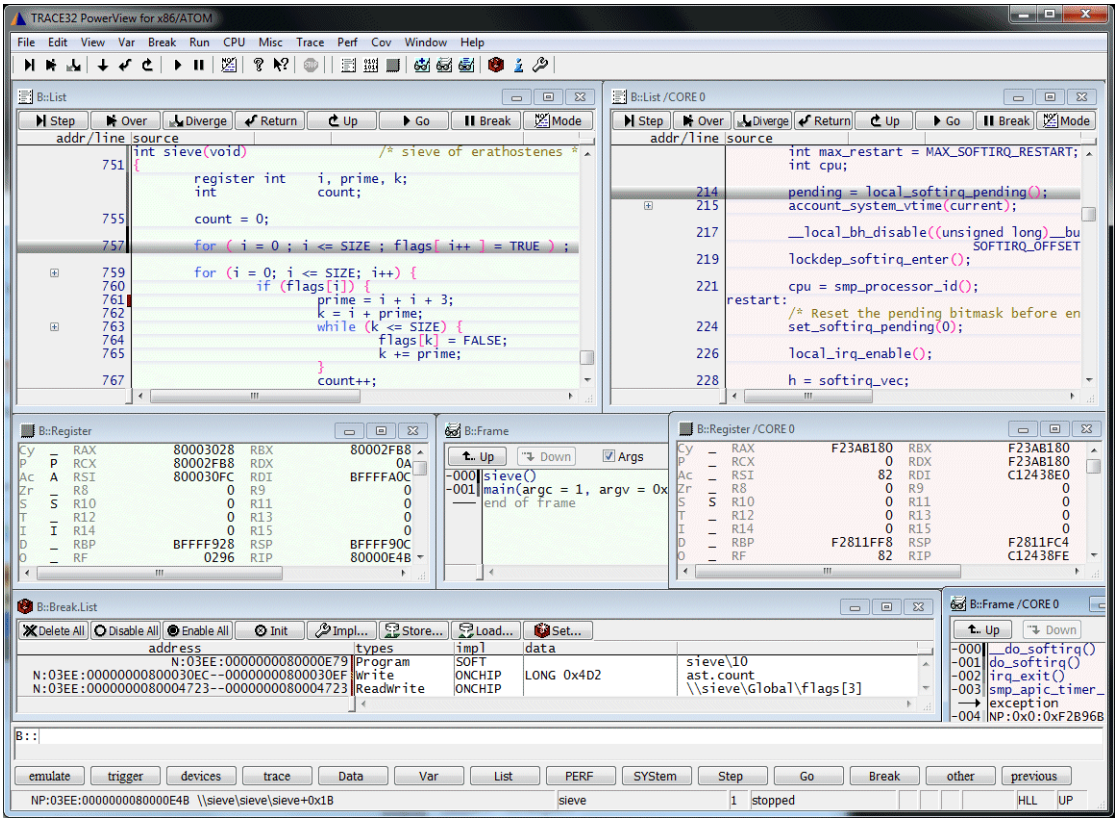
```
c:\t32\t32mx64.exe -s g:\and\training\start.cmm
```

2. Use *T32Start* to define an automated start-up script.

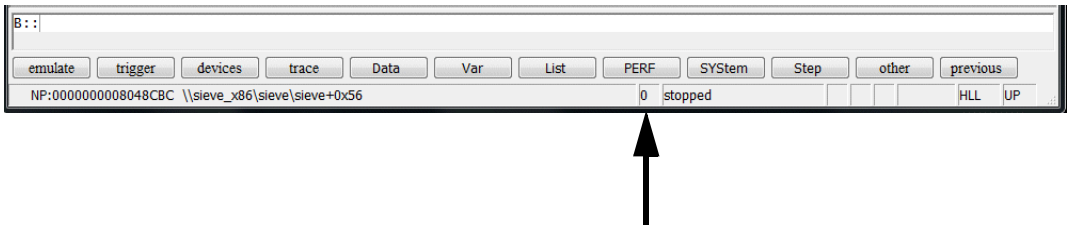


SMP Concept

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



In TRACE32 PowerView one core is the selected one.



The **Cores** field in the TRACE32 PowerView State Line displays the number of the currently selected core

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

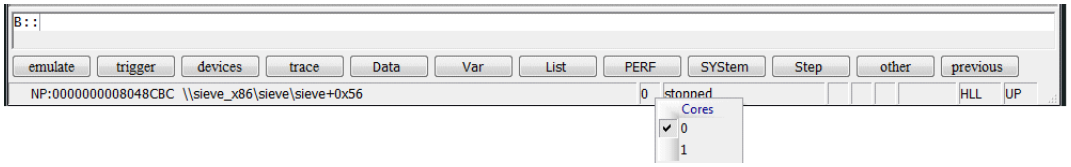
- By default system information is visualized from the perspective of the selected core.

```
; core 0 is the selected core

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

Register.view                       ; display the core registers 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 core registers of
                                    ; core 1
```

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



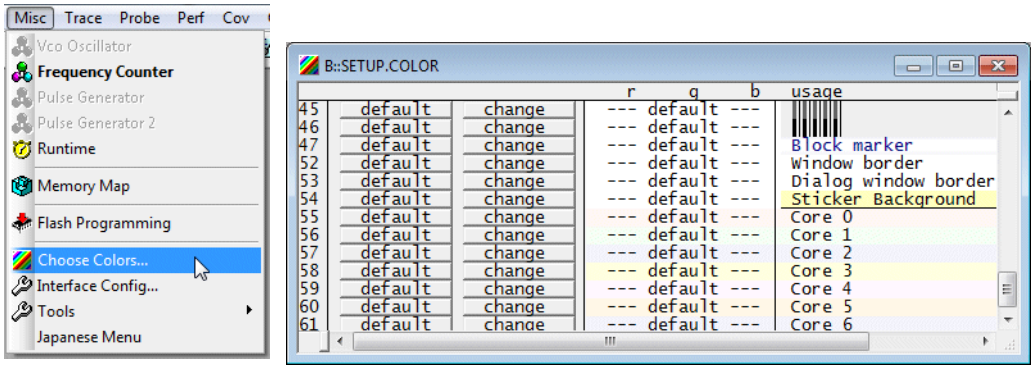
CORE.select *<number>* Select a different core

TRACE32 PowerView distinguishes two types of information:

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

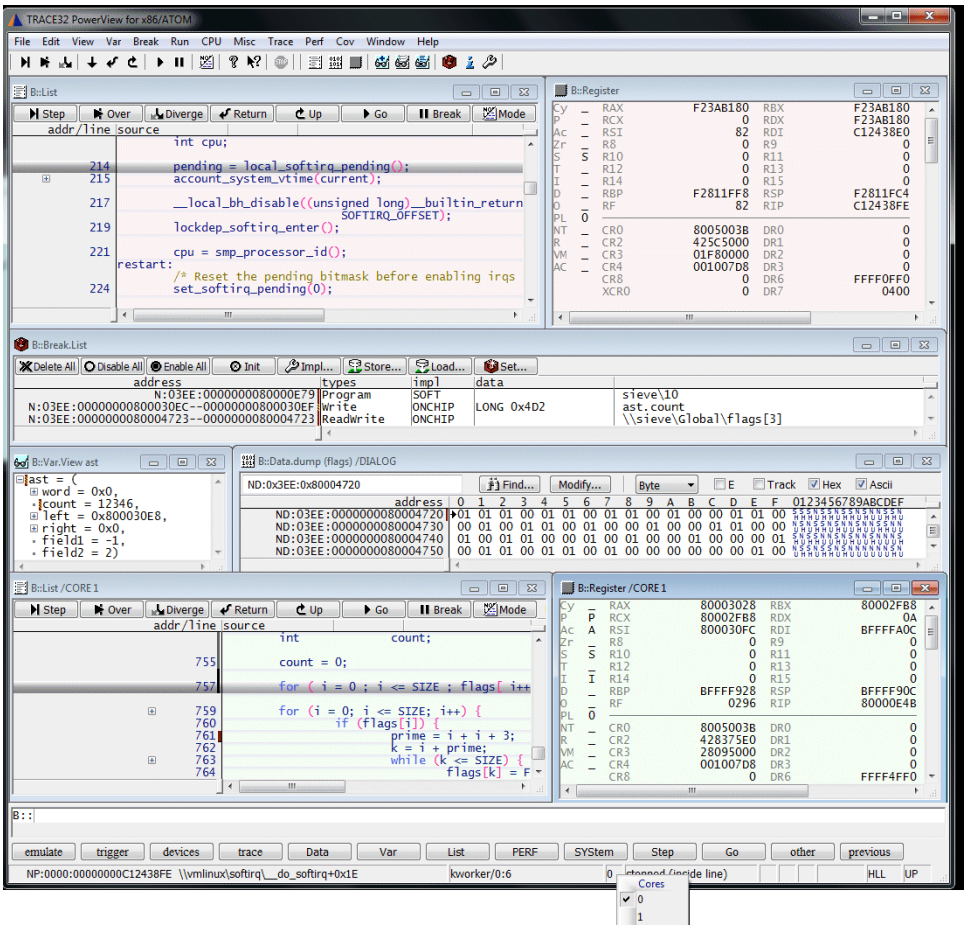
  Typical core-specific information are: register contents, source listing of the code currently executed by the core, the stack frame.

  TRACE32 PowerView uses predefined color settings for the cores.



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

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

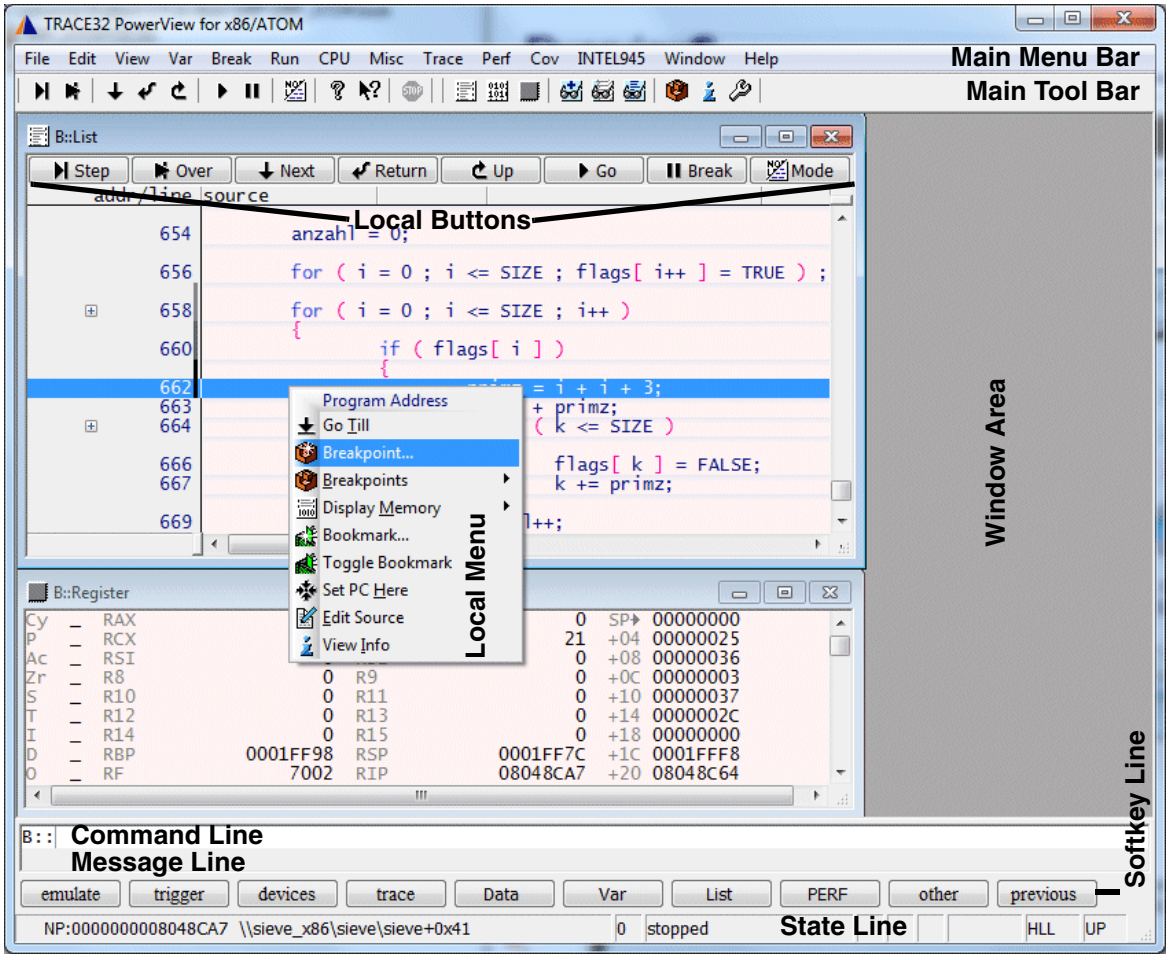


Core-specific information (here for the currently selected one)

Information common for all cores

Core-specific information (here for the core 1)

TRACE32 PowerView Components



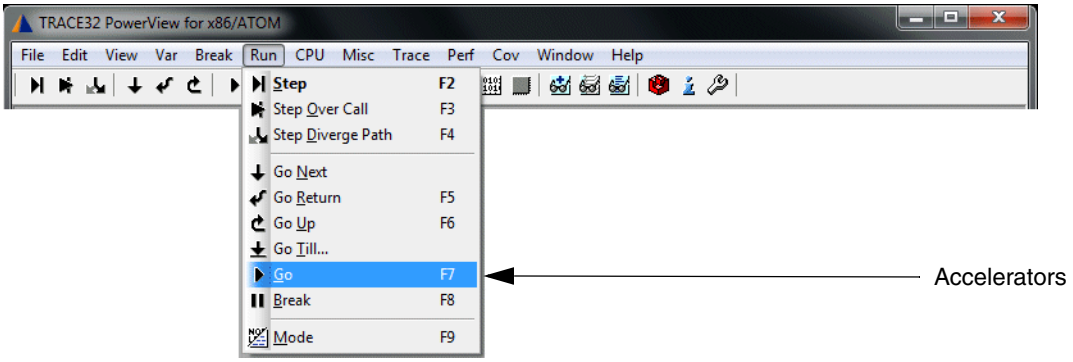
The structure and functionality of TRACE32 PowerView is largely defined by the file **`t32.men`** which is located in the TRACE32 system directory.

TRACE32 allows you to modify the GUI so it will better fit to 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 commands sorted by functional groups.

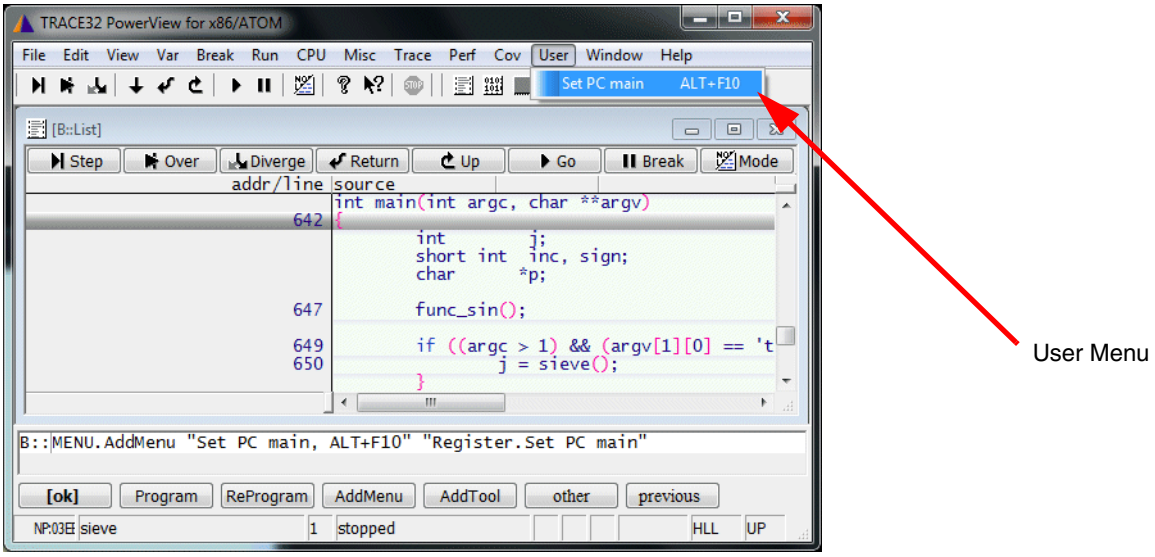
For often used commands accelerators are defined.



A user specific menu can be defined very easily:

MENU.AddMenu *<name> <command>*	Add a user menu
MENU.RESet	Reset menu to default

```
MENU.AddMenu "Set PC to main" "Register.Set PC main"

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



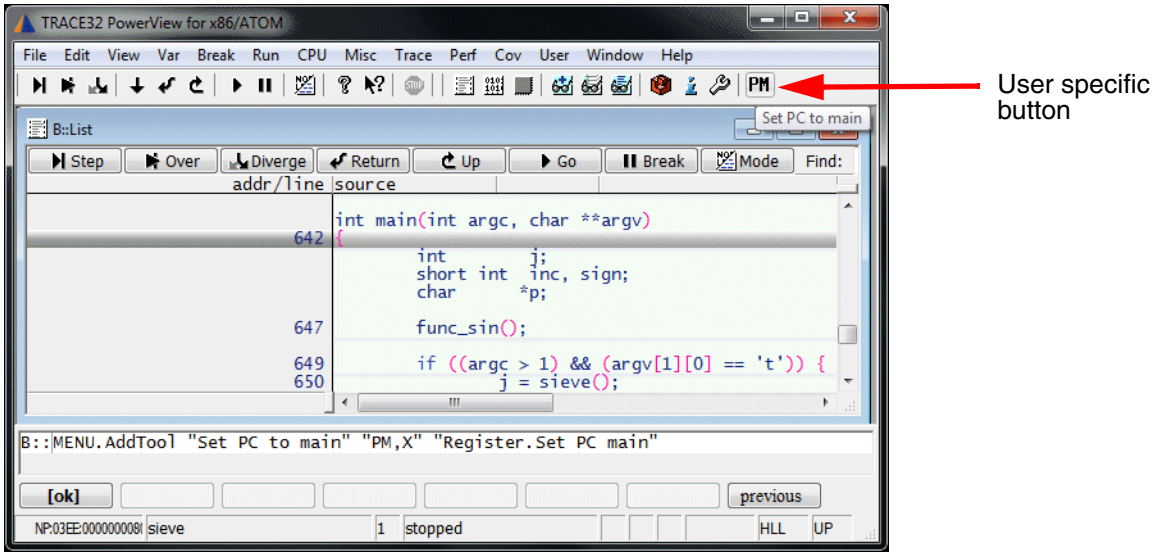
For more complex changes to the main menu bar refer to **"Training Menu Programming"** (training_menu.pdf).

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

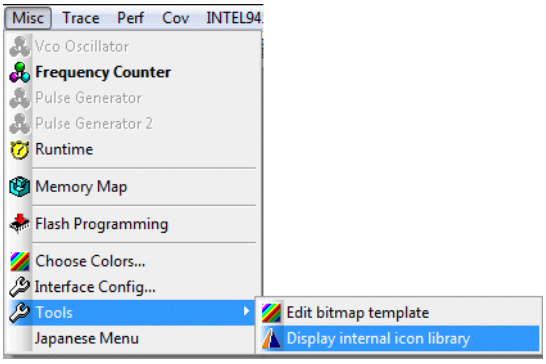
```
MENU.AddTool "Set PC to main" "PM,X" "Register.Set PC main"
```

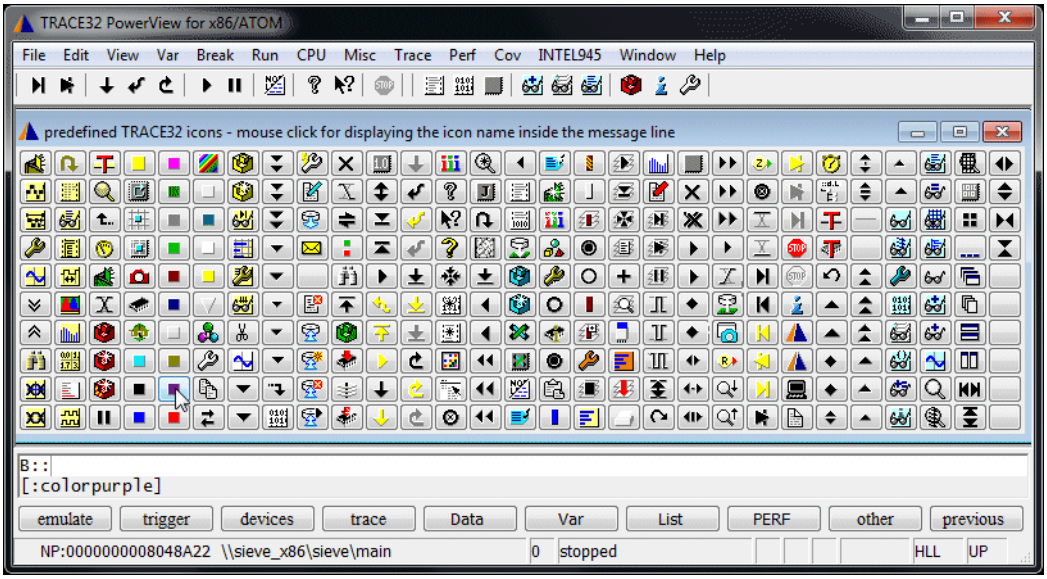


Information on the *<tool image>* can be found in ***Help -> Contents***

TRACE32 Documents -> PowerView User Interface -> PowerView User's Guide -> MENU -> Programming Commands -> **TOOLITEM*.***

All predefined TRACE32 icons can be inspected as follows:






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

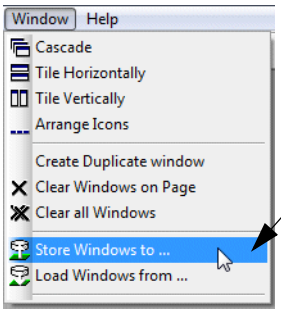
The predefined icons can easily be used to create new icons.

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

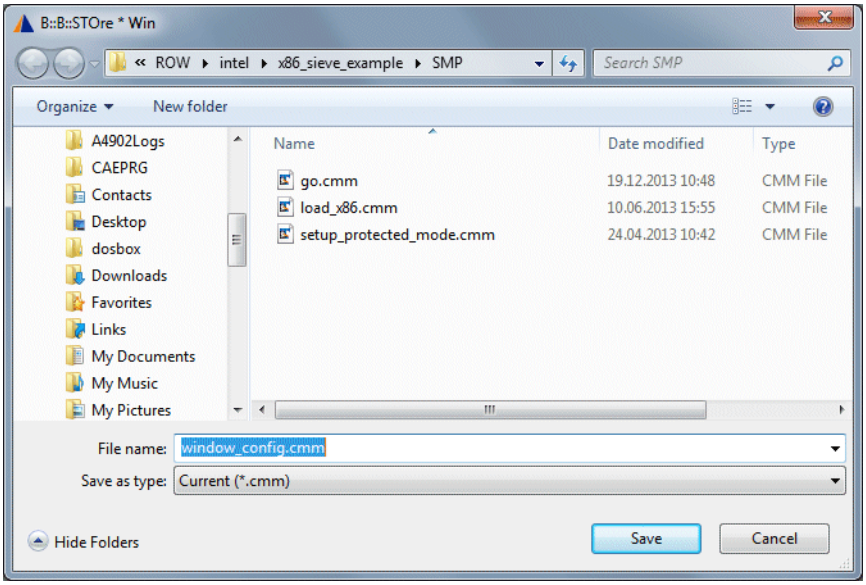
For more complex changes to the main tool bar refer to **“Training Menu Programming”** (training_menu.pdf).

Save Page Layout

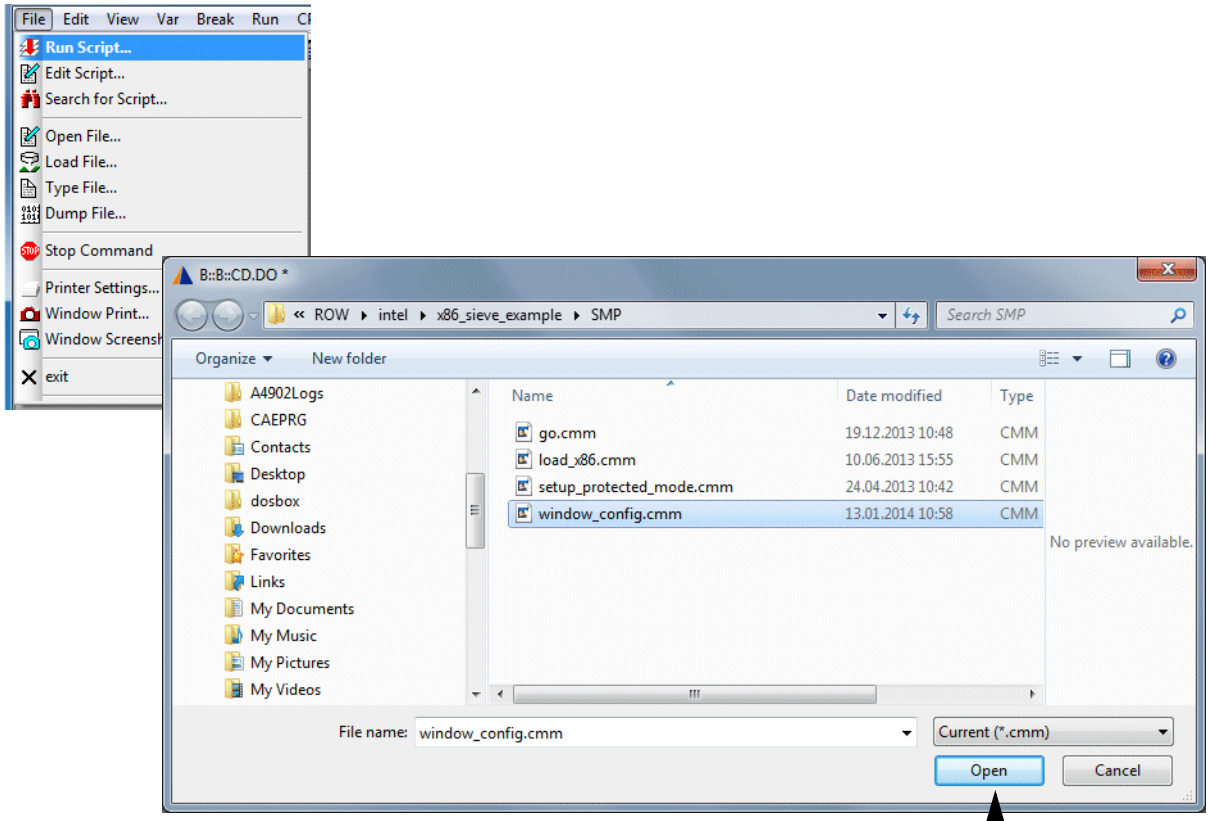
No information about the page layout is saved when you exit TRACE32 PowerView. To save the window layout use the **Store Window to …** command in the **Window** menu.



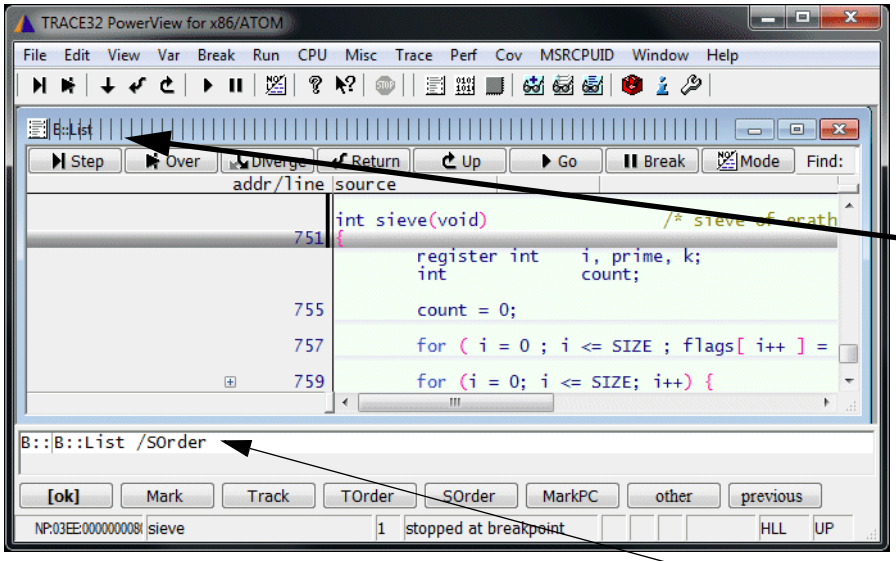
Store Windows to … generates a script, that allows you to reactivate the window-configuration at any time.




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

 B::

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

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

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

 WinPAGE.select P000

 ENDDO
```

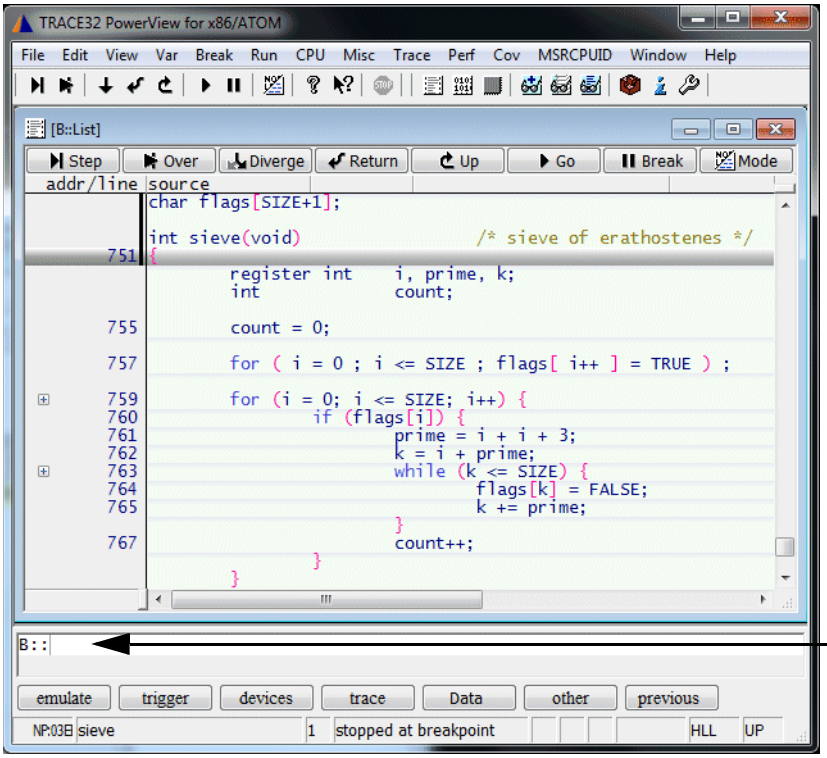


Run the script to reactivate the stored window-configuration



The Window Header displays the command which was executed to open the window

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



Command Structure

Device Prompt

Selects the command set used by the TRACE32:

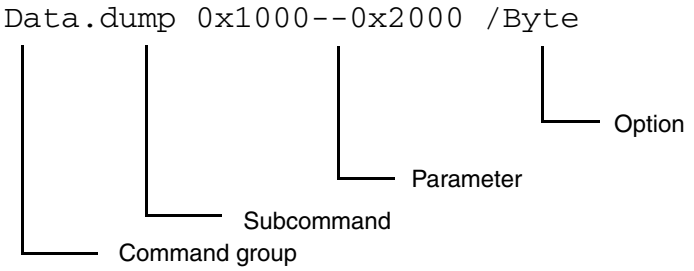
no device prompt	TRACE32 PowerView commands
B::	**command set for Debugger** (B stands for BDM which was the first on-chip debug interface supported by Lauterbach)

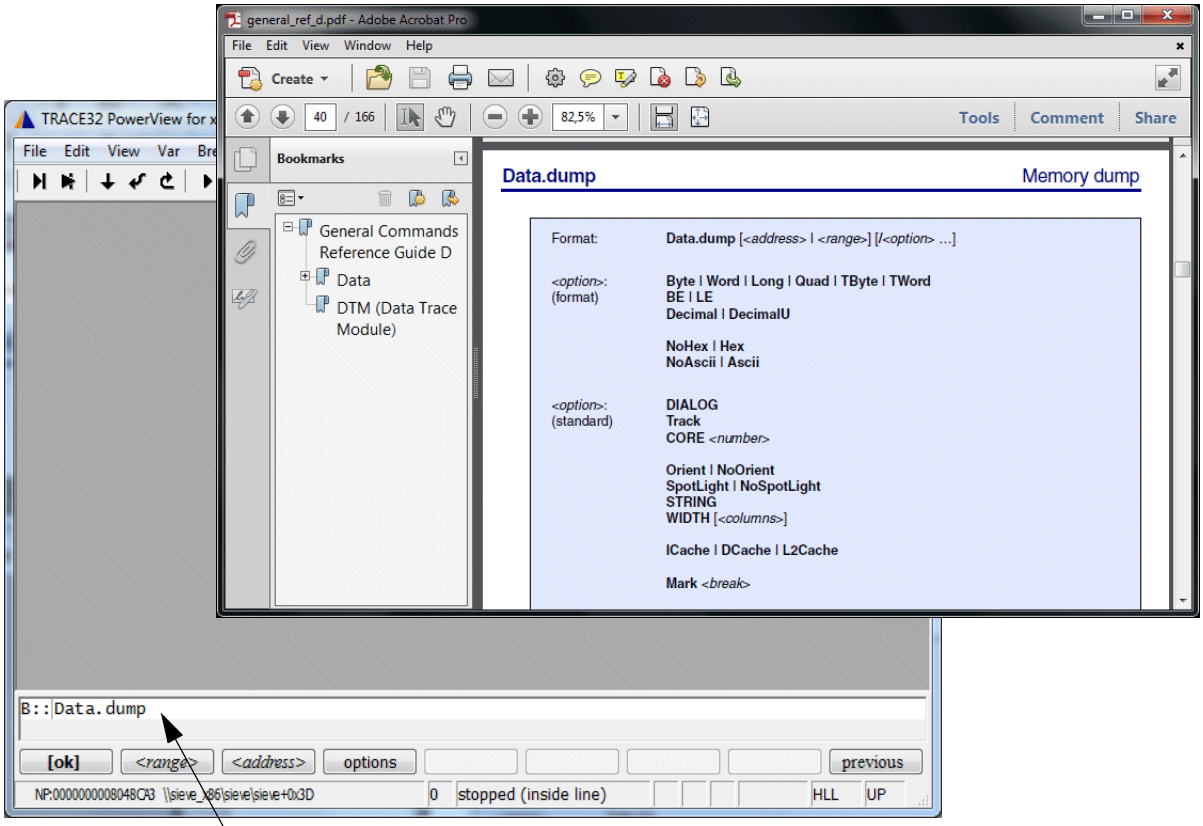
Command Examples

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

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

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





Enter the command to the command line.

Add one blank.

Push F1 to get the online help for the specified command.

RADIX.<*mode*> Define parameter syntax

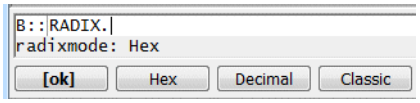
The RADIX defines the input format for numeric values.

RADIX.Hex	Number base is hex and C-like operators are used (default).
RADIX.Decimal	Number base is decimal and C-like operators are used.

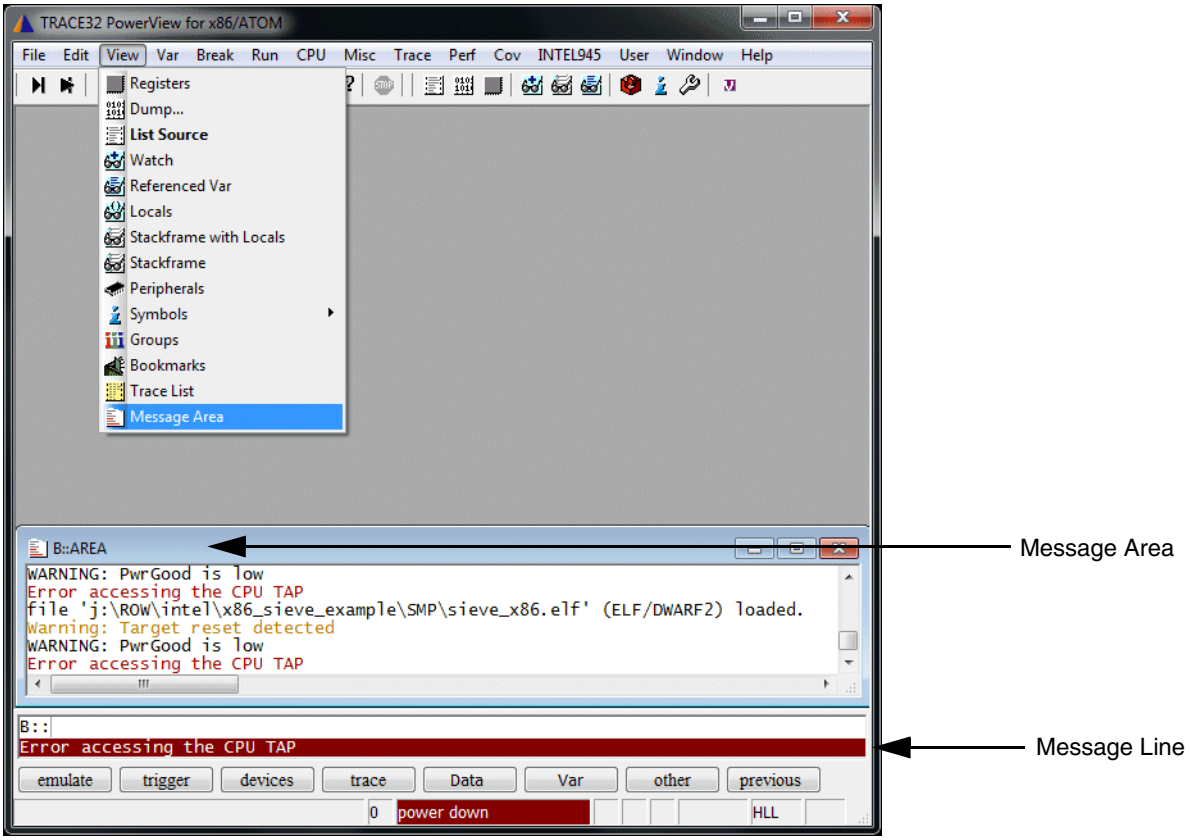
Examples:

	Decimal	**Hex**
`Data.dump 100`	`100d`	`100h`
`Data.dump 100.`	`100d`	`100d`
`Data.dump 0x100`	`100h`	`100h`

To check the currently used parameter syntax, type **`RADIX.`** to the command line.



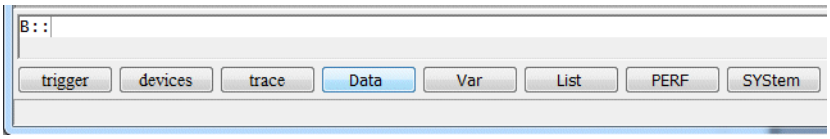
Message Line



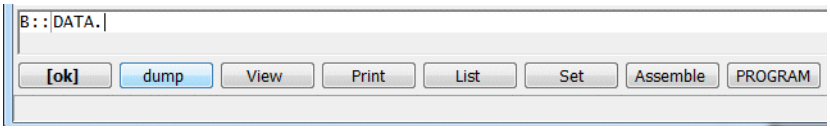
- **Message line** for system and error messages
- **Message Area window** for the display of the last system and error messages

The softkey line allows to enter a specific command step by step.

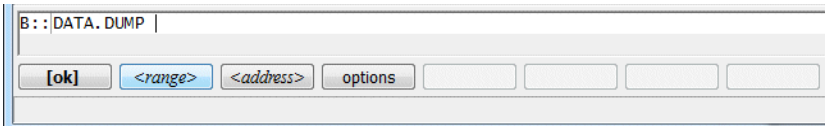
Select the command group, here **Data.**



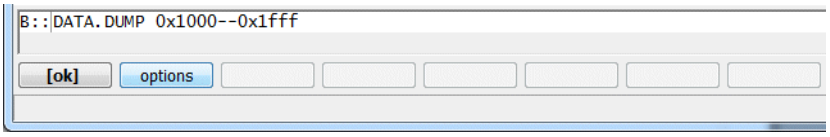
Select the subcommand, here **dump.**



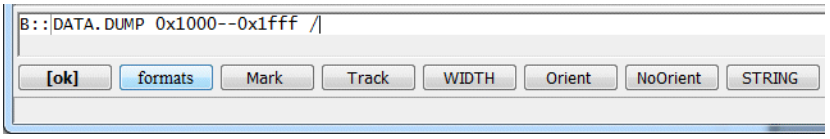
Angle brackets request an entry from the user,
here e.g. the entry of a *<range>* or an *<address>*.



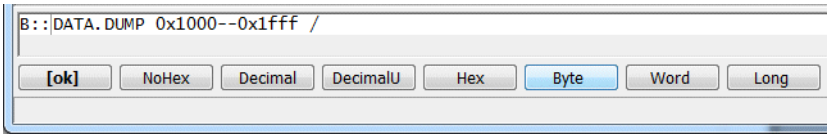
The display of the hex. dump can be adjusted to your needs by an option.



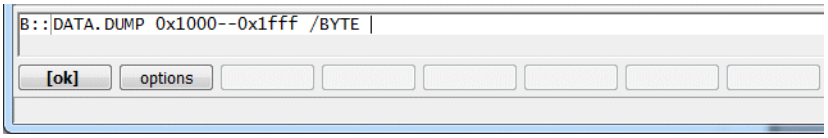
Select the option **formats** to get a list of all format options.

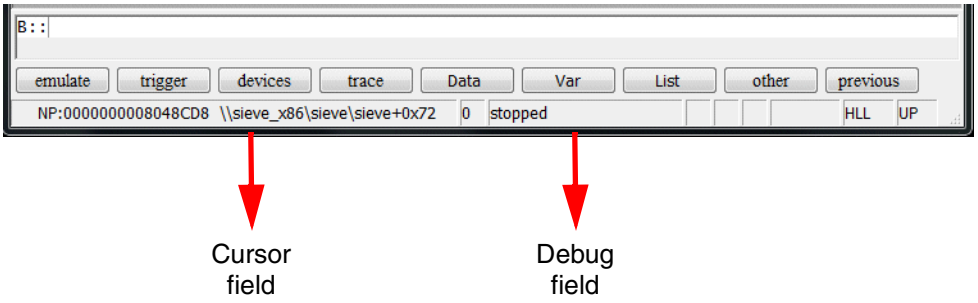


Select a format option, here **Byte**.



The command is complete now.



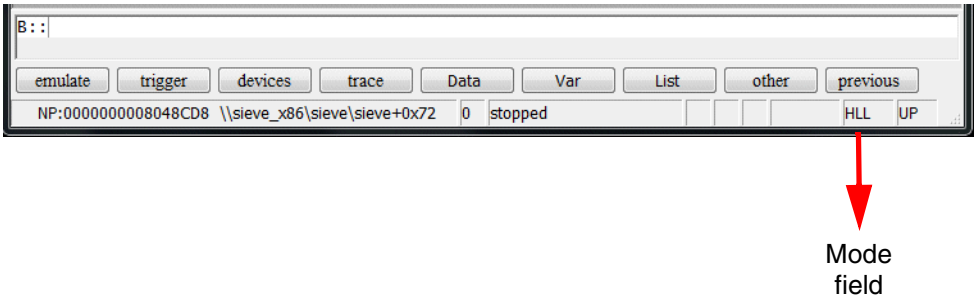


The **Cursor** field provides:

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

The **Debug** field provides:

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

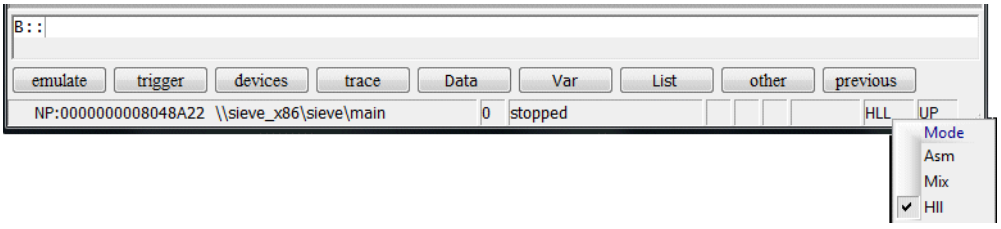


The **Mode** field indicates the debug mode. The debug mode defines how source code information is displayed.

- 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.



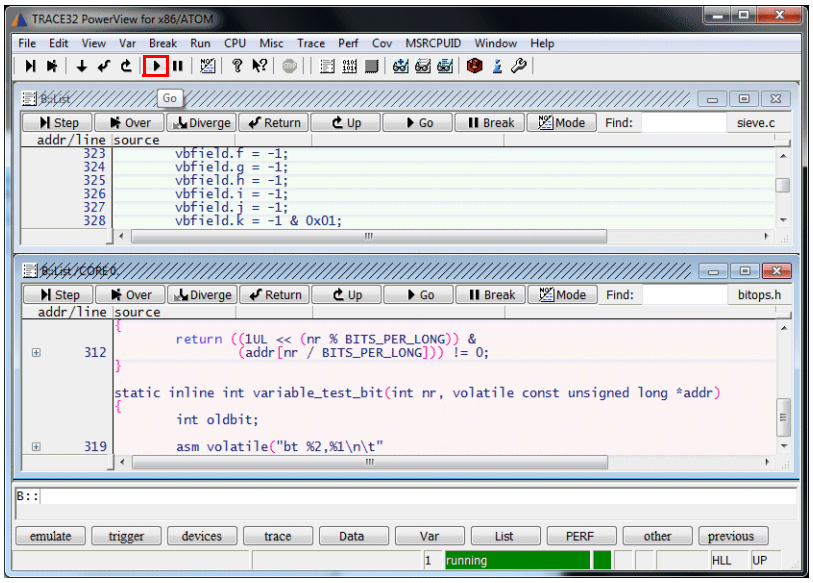
Further Documentation

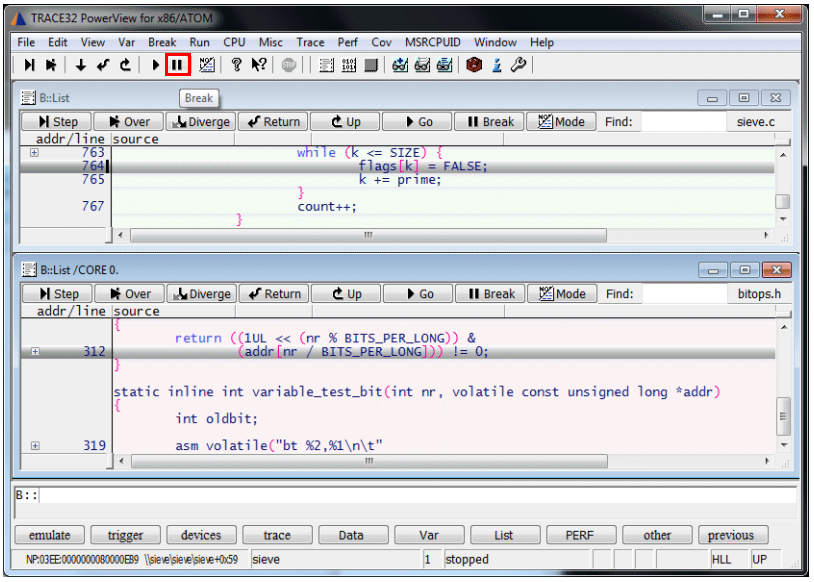
The following PDFs provide detailed information on TRACE32 PowerView:

- **"PowerView User's Guide"** (ide_user.pdf)
- **"PowerView Command Reference"** (ide_ref.pdf)

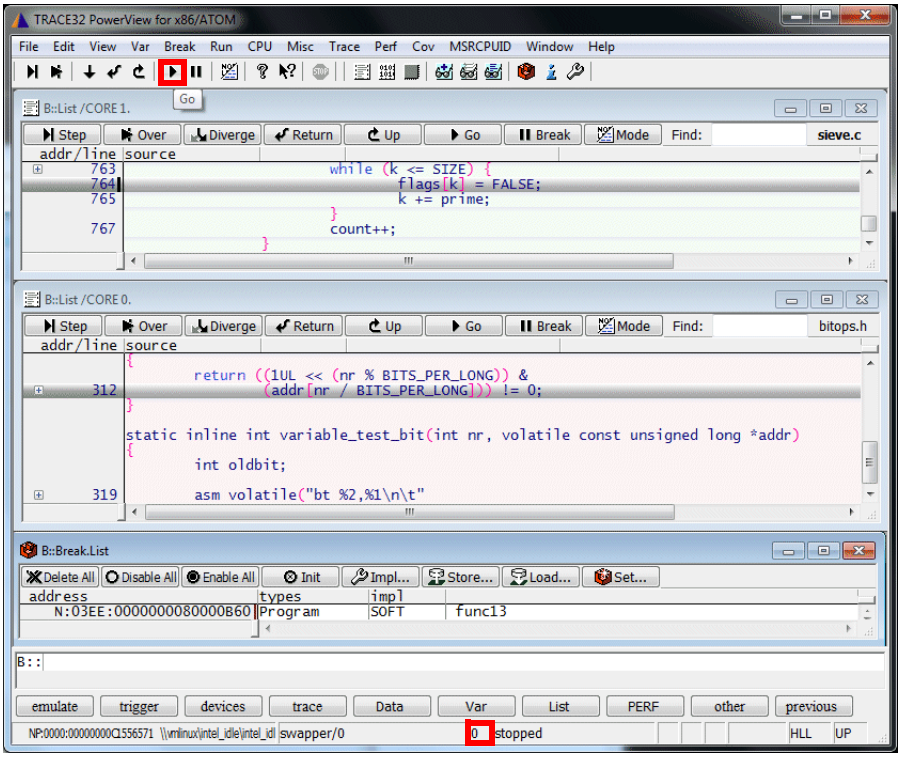
Go/Break

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

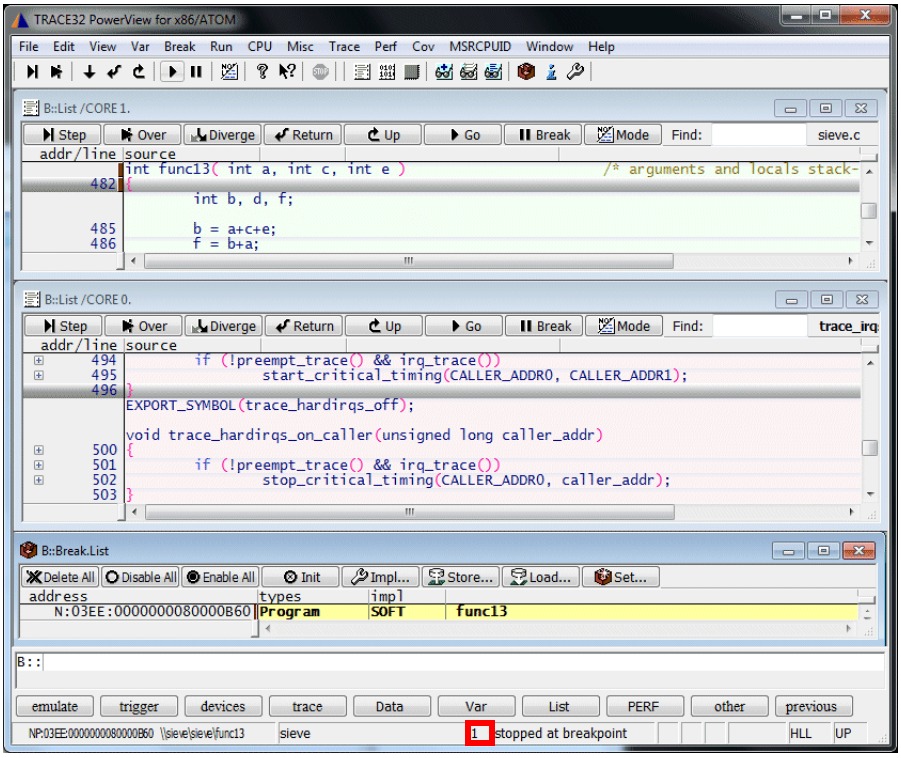




If a breakpoint is hit, TRACE32 makes the core the selected one on which the breakpoint occurred.



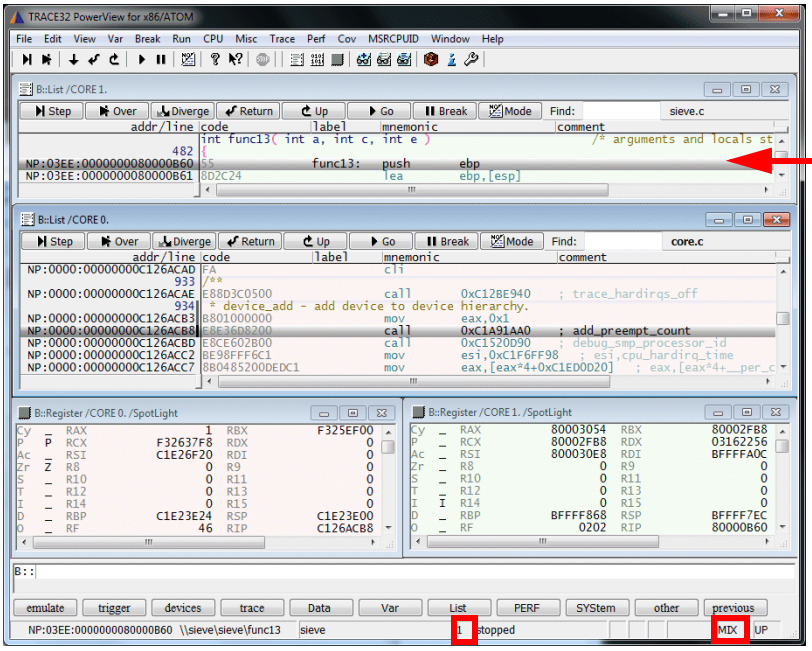
CORE 0 is the selected one when the program execution is started.

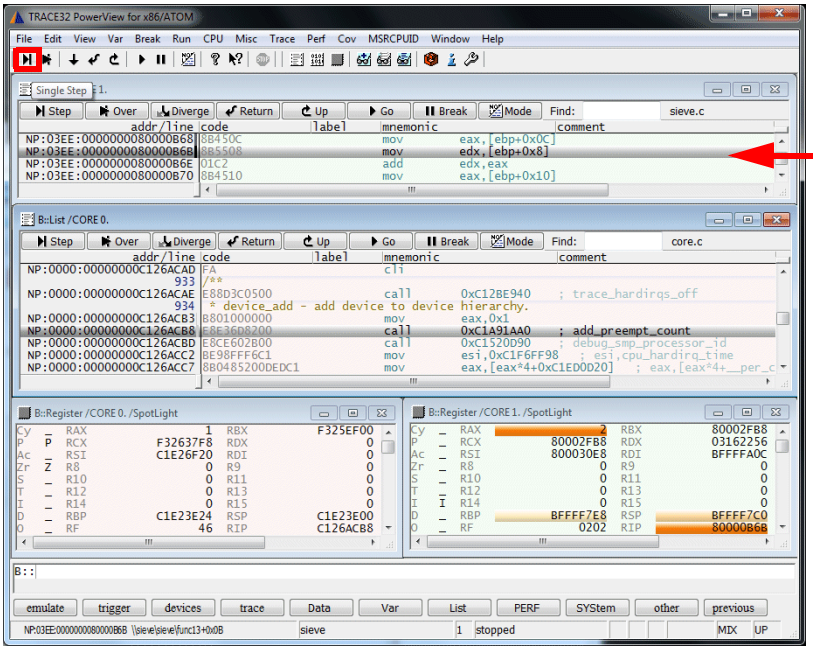


The breakpoint was hit on **CORE 1**. So CORE 1 is the selected one after the program execution stopped.

Single Stepping on Assembler Level

Assembler single steps are only performed on the selected core.



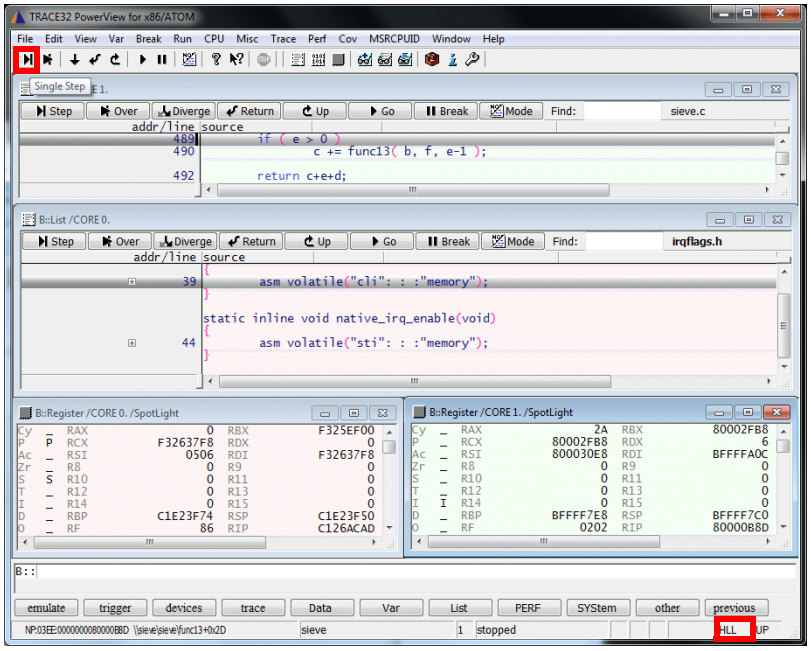


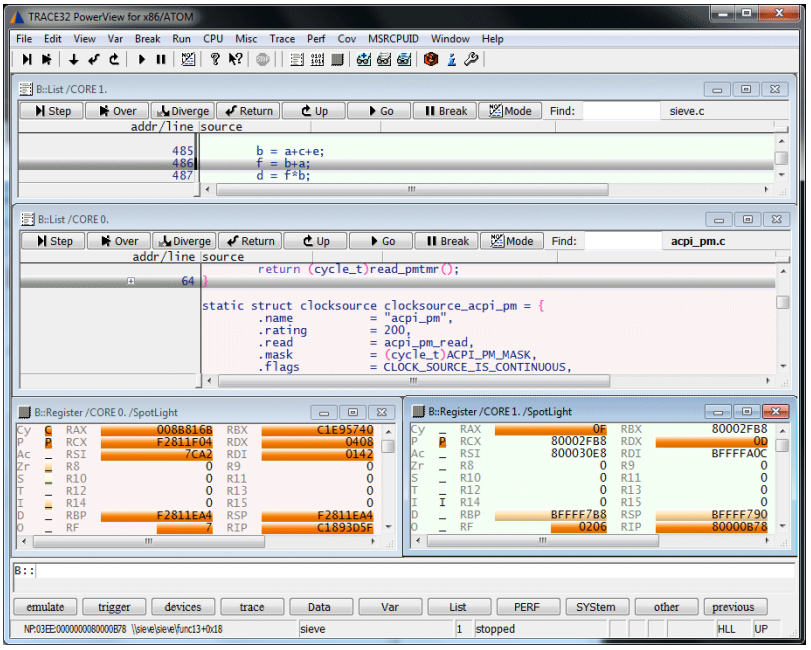
Only the program counter of core 1 has changed

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.

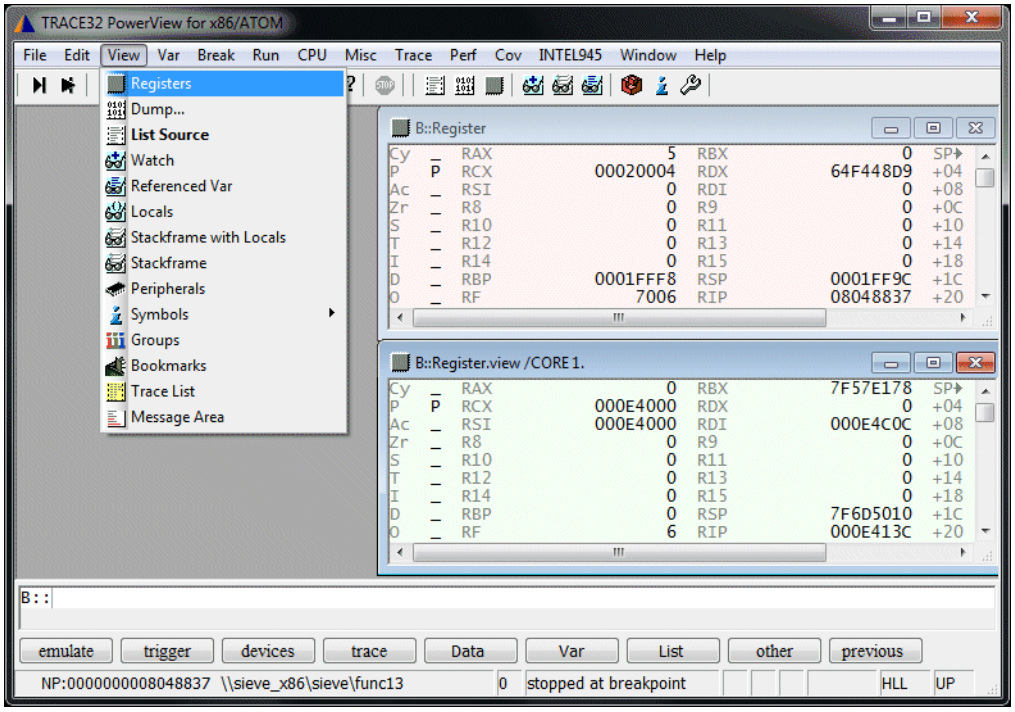




Mode.HII **Step**	Select High-level language mode for debugging and perform a single step.
Step.HII	Perform an HLL single step.
SETUP.StepWithinTask ON	When ON all HLL stepping is performed only in the currently active task.

Core Registers

Display the Core Registers



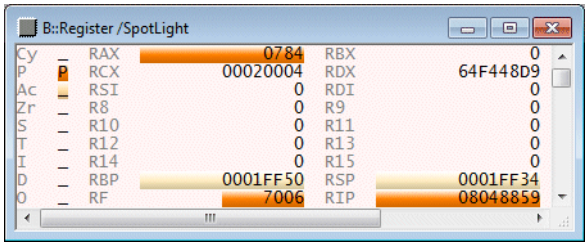
The core register contents is core-specific information. It is printed on a colored background.

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

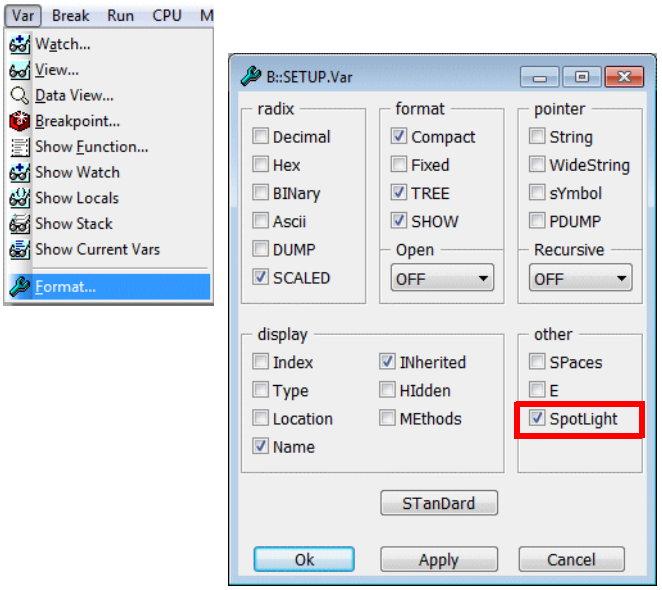
```
Register.view                          ; display core register contents of
                                       ; currently selected core
                                       ; (here core 0)

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

Colored Display of Changed Registers

```
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.
```

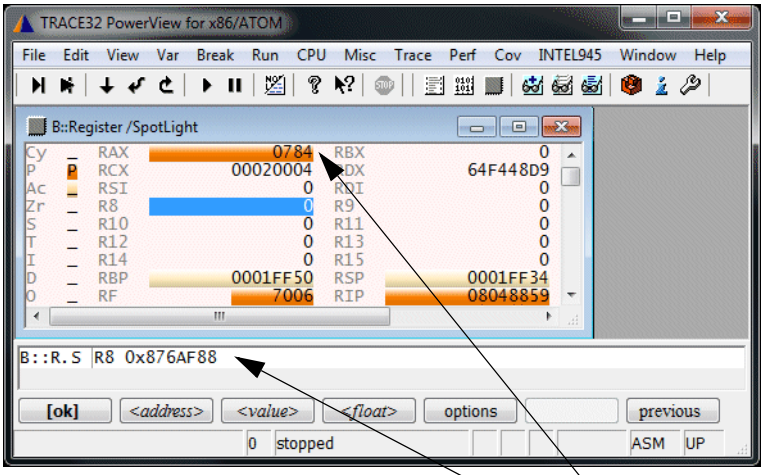


Establish /SpotLight as default option



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

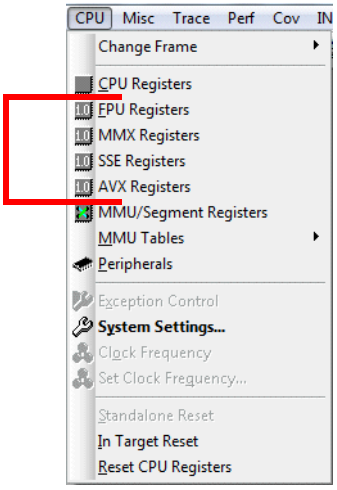


By double clicking to the register contents
a **`Register.Set`** command is automatically displayed
in the command line.
Enter the new value and press Return to modify the
register contents.

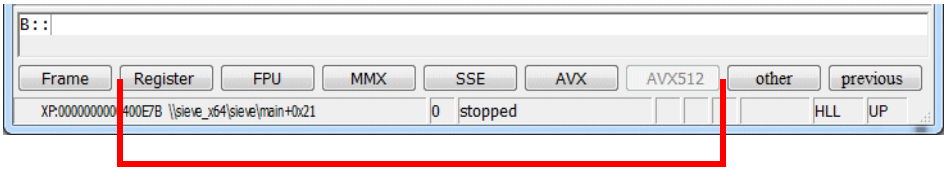
Register.Set *<register> <value>*	Modify core register of selected core
Register.Set *<register> <value>* **/CORE** *<n>*	Modify core register of specified core

Further Register Sets

TRACE32 PowerView supports also the display and modification of all other register sets available for the core under debug.



A list of all other supported register sets is also provided by the softkey line.



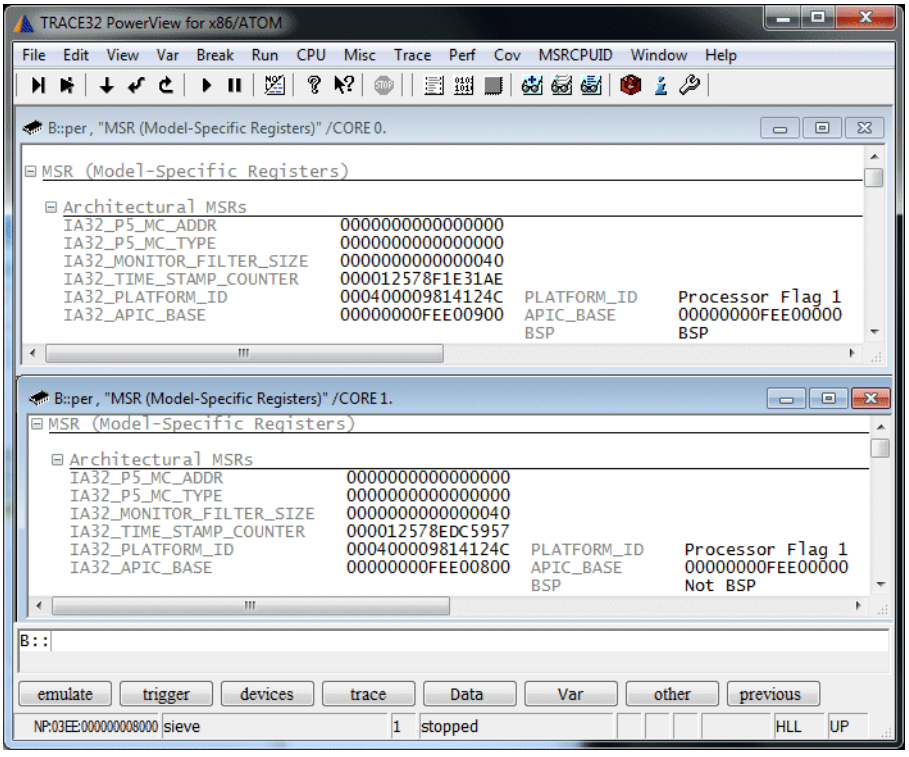
Display the Special Function Registers

TRACE32 supports a free configurable window to display/manipulate configuration registers and on-chip peripheral registers at a logical level.

The so-called PER file describes these registers. The PER file is either provided by Lauterbach or by the chip manufacturer.

In an SMP system all cores can usually access the commonly used external interfaces. So TRACE32 PowerView regards all these registers as common resources and thus displayed them on a white background.

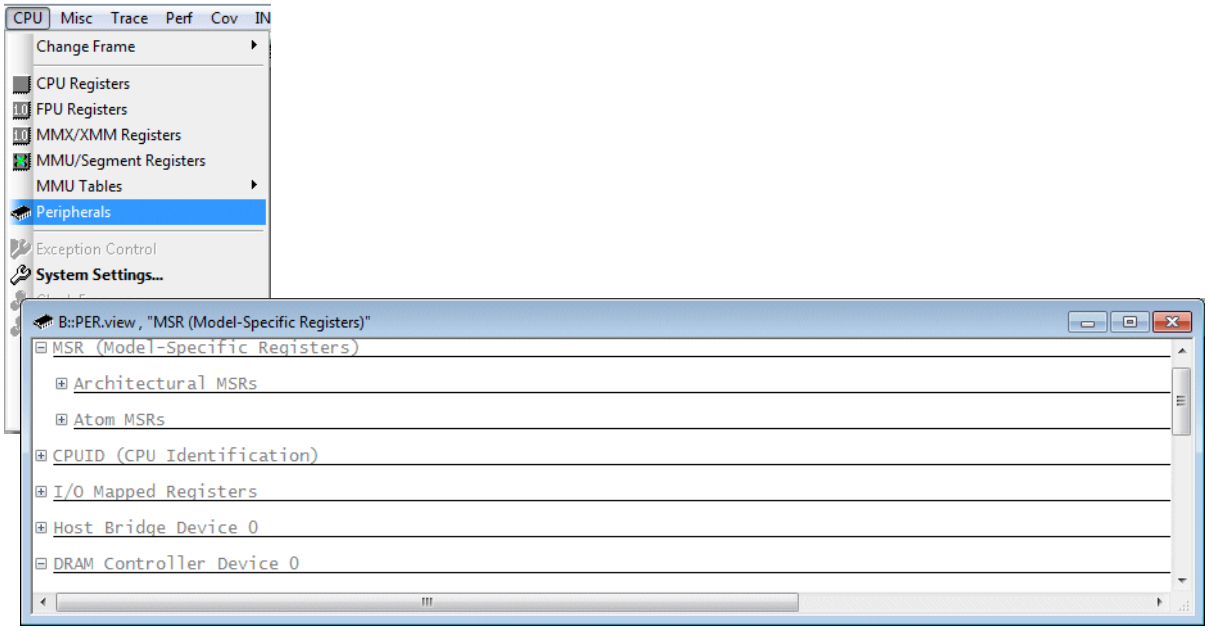
But not all configuration registers are common resources. Exceptions are the core-related registers e.g. CPUID registers, MSR registers …



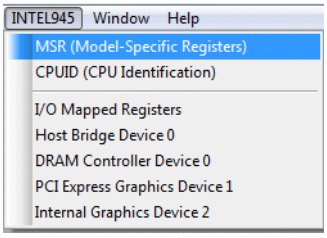
TRACE32 PowerView provides the /**CORE** *<n>* option in order to display details on core-related configuration registers:

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

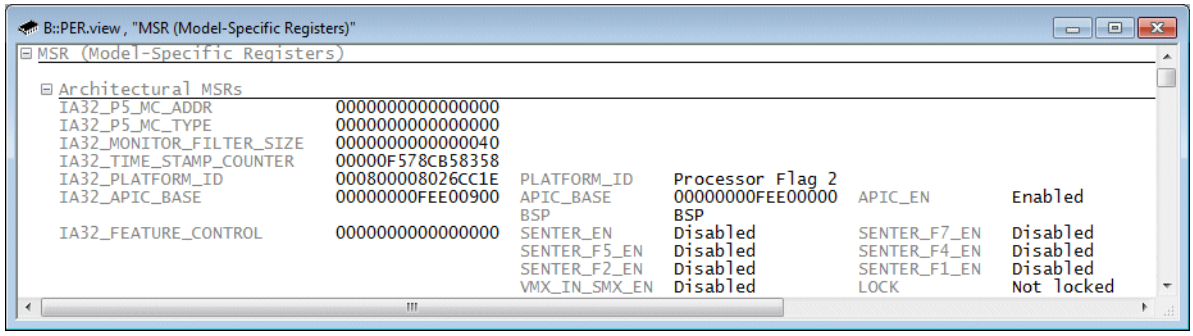
The individual configuration registers/on-chip peripherals are organized by TRACE32 PowerView in a tree structure. On demand, details about a selected register can be displayed.

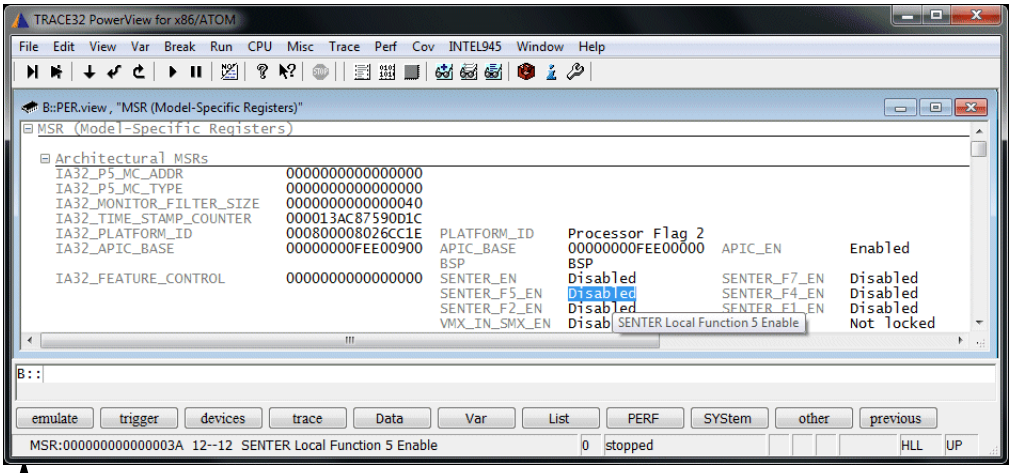


Platform menu



Platform menu provides direct access to specific registers





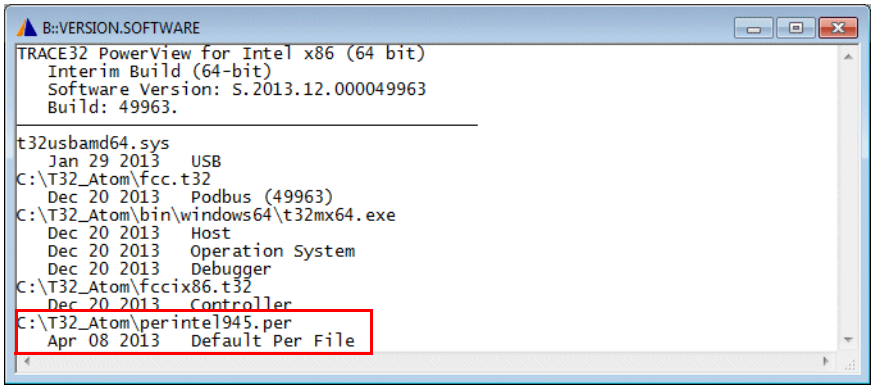
Select the content of a configuration register:
The access class, address, bit position and the full name of the selected item are displayed in the state line; the full name of the selected item is taken from the processor/chip manual.

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 currently used PER definition file can be displayed by using:

VERSION.SOFTWARE



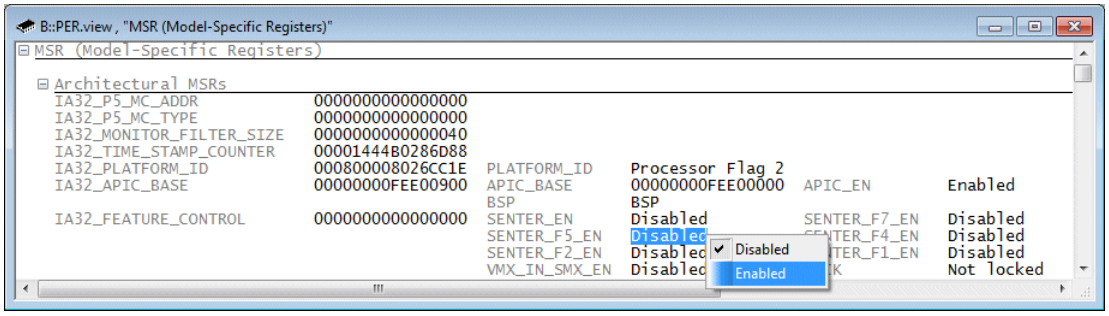
PER.view [*<filename>*] [*<tree-search-item>*] Display the configuration registers/on-chip peripherals

```
PER.view C:\T32\perintel948.per     ; Use the peripheral file
                                    ; perintel948 instead of the default
                                    ; PER definition file
```

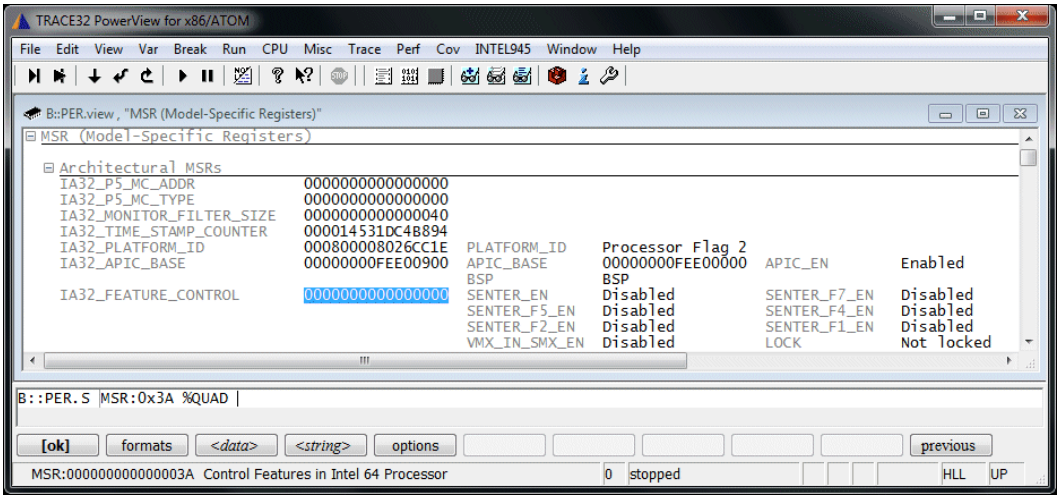
Modify a Special Function Register

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>*] *<string>*	Modify configuration register/on-chip peripheral
Data.Set *<address>|<range>* [**%***<format>*] *<string>*	Modify memory

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

Memory Display and Modification

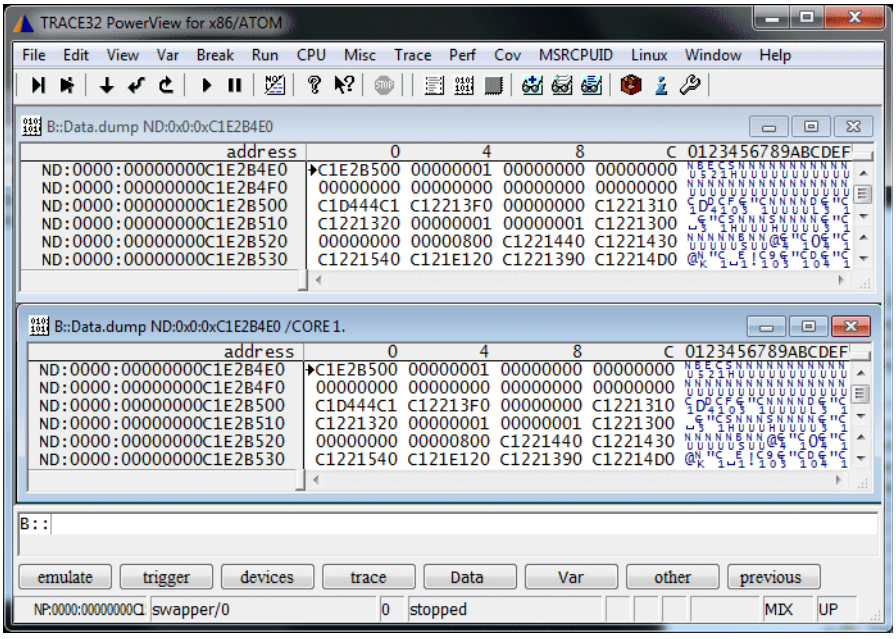
This training section introduces the most often used methods to display and modify memory:

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

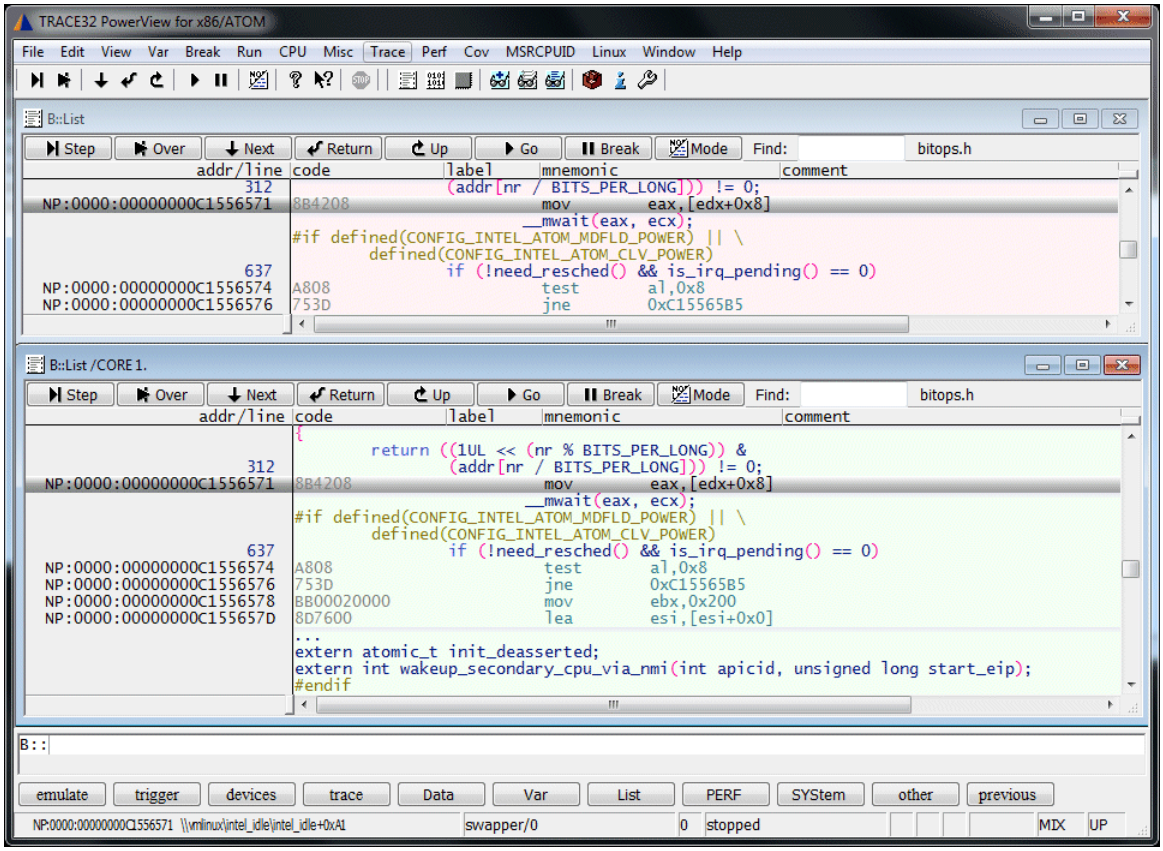
Shared memory is a characteristics of an SMP system. This is the reason why the **Data.dump** window is regarded as common information and is displayed therefore on a white background. TRACE32 PowerView assumes that cache coherency is maintained in an SMP system.

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.

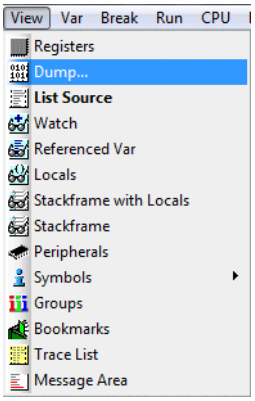


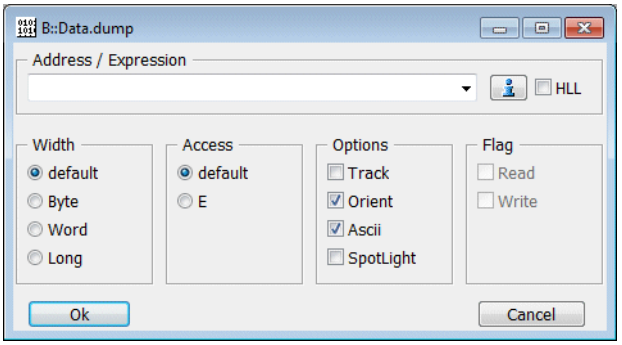
A so-called **access class** is always displayed together with a memory address. Examples:

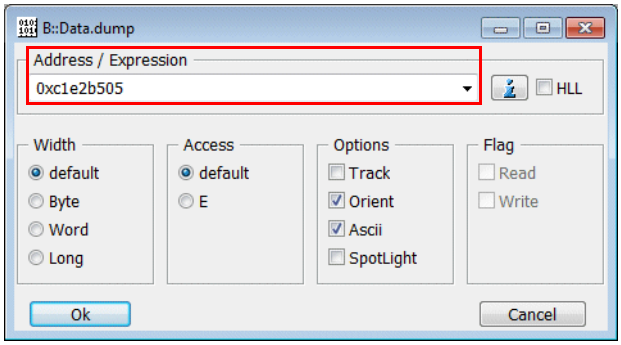
NP:1000	Protected Mode Program (32-bit) address 0x1000
ND:6814	Protected Mode Data (32-bit) address 0x6814

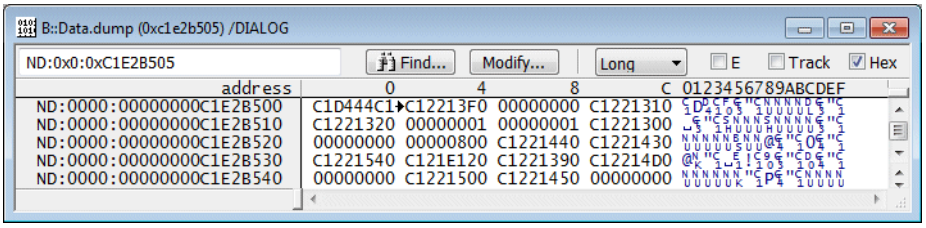
For a list of all access classes provided for the Intel® x86/x64 architecture refer to **"Intel® x86/x64 Debugger"** (debugger_x86.pdf).

Basics


View
Var
Break
Run
CPU
Registers
Dump...
List Source
Watch
Referenced Var
Locals
Stackframe with Locals
Stackframe
Peripherals
Symbols
Groups
Bookmarks
Trace List
Message Area


B::Data.dump
Address / Expression
HLL
Width
default
Byte
Word
Long
Access
default
E
Options
Track
Orient
Ascii
SpotLight
Flag
Read
Write
Ok
Cancel



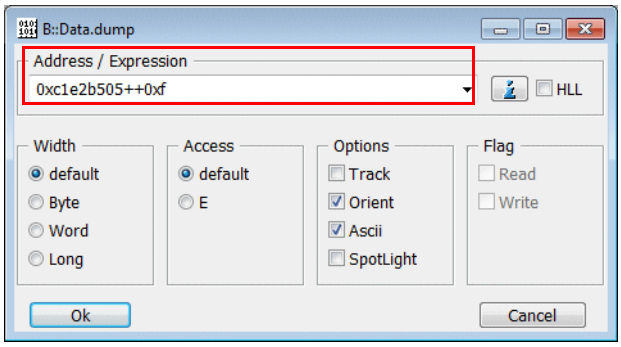


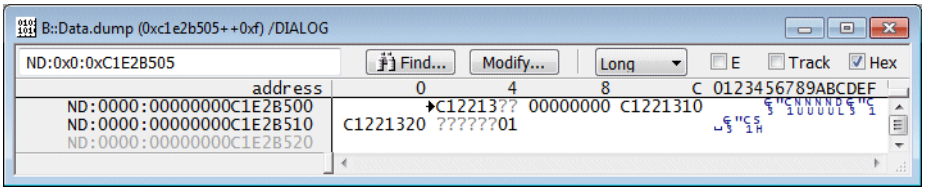
Please be aware, that TRACE32 permanently updates all windows. The default update rate is 10 times per second.

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.

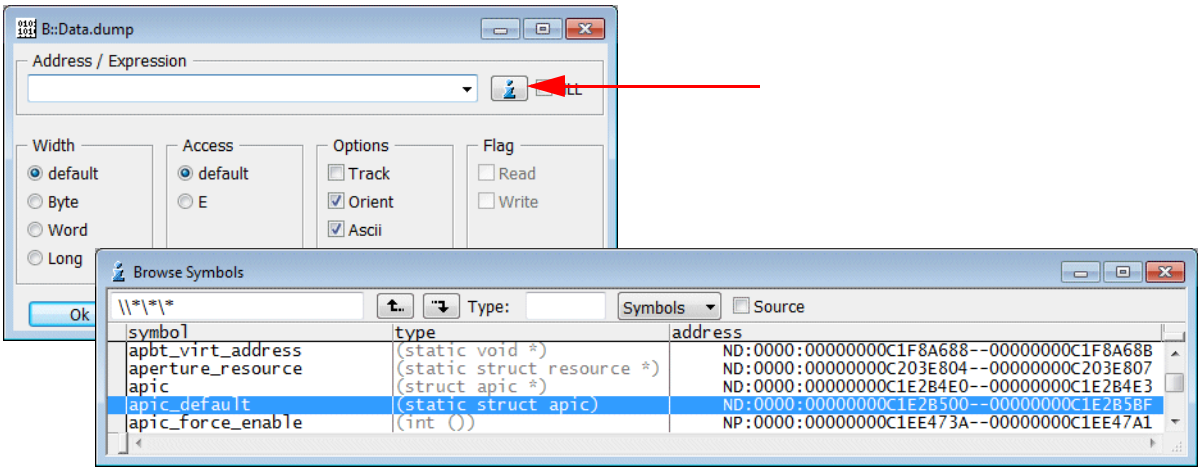
Conventions for address ranges:

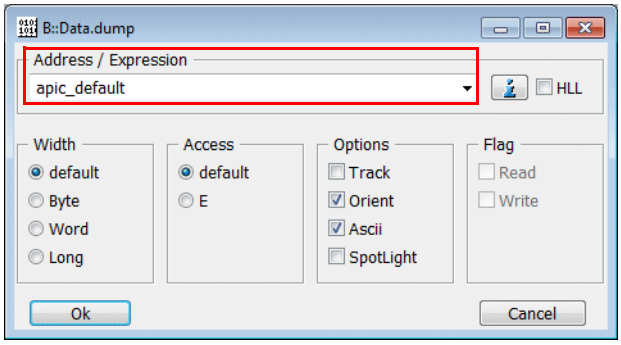
- <start address>--<end_address>
- <start address>..<end_address>
- <start address>++<offset_in_byte>





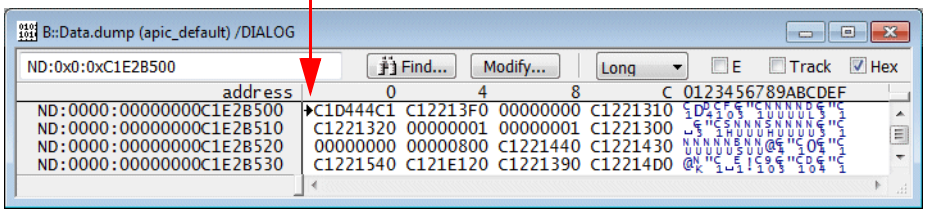
Use **i** to select any symbol name or label known to TRACE32 PowerView.





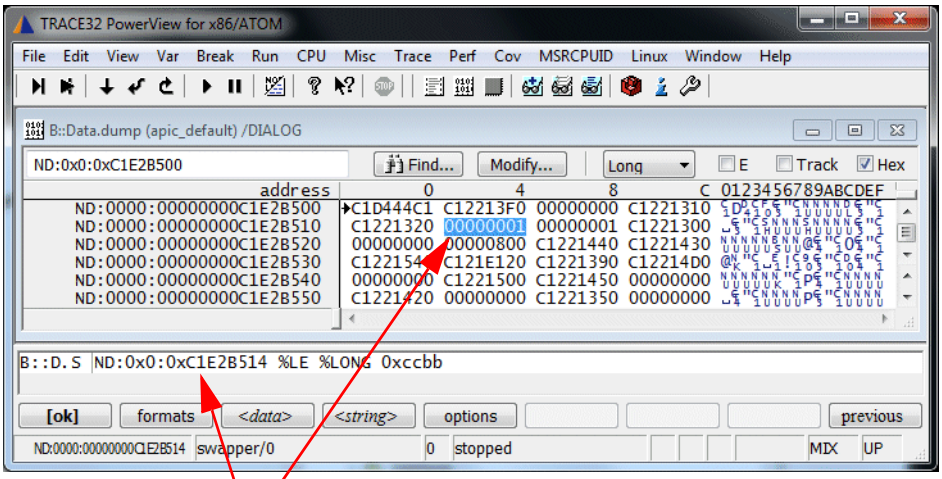
By default an oriented display is used (line break at 2^x).

A small arrow indicates the specified dump address.



Data.dump *<address>* | *<range>* [{**/***<option>*}] Display a hex dump of the memory

```
Data.dump 0xC1E2B4E0                    ; Display a hex dump starting at
                                        ; logical address 0xC1E2B4E0 in the
                                        ; current address space

Data.dump 0xC1E2B4E0--0xC1E2B4EF        ; Display a hex dump of the
                                        ; specified address range

Data.dump 0xC1E2B4E0..0xC1E2B4EF        ; Display a hex dump of the
                                        ; specified address range

Data.dump 0xC1E2B4E0++0x7               ; Display a hex dump of the
                                        ; specified address range

Data.dump ND:0x88:0xC1E2B4E0            ; Display a hex dump starting at
                                        ; logical address 0xC1E2B4E0 in the
                                        ; address space of the process with
                                        ; the space ID 0x88

Data.dump apic_default                  ; Display a hex dump starting at
                                        ; the address of the label
                                        ; apic_default

Data.dump apic_default /Byte            ; Display a hex dump starting at
                                        ; the address of the label
                                        ; apic_default in byte format
```

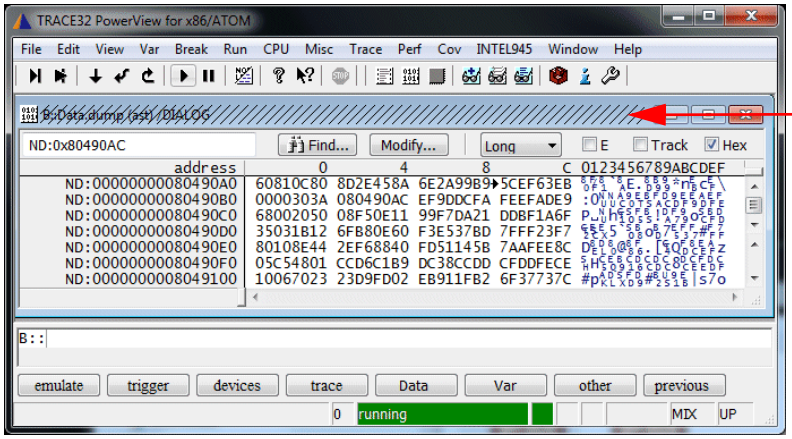


By a left mouse double-click to the memory contents a **Data.Set** command is automatically displayed in the command line, you can enter the new value and confirm it with Return.

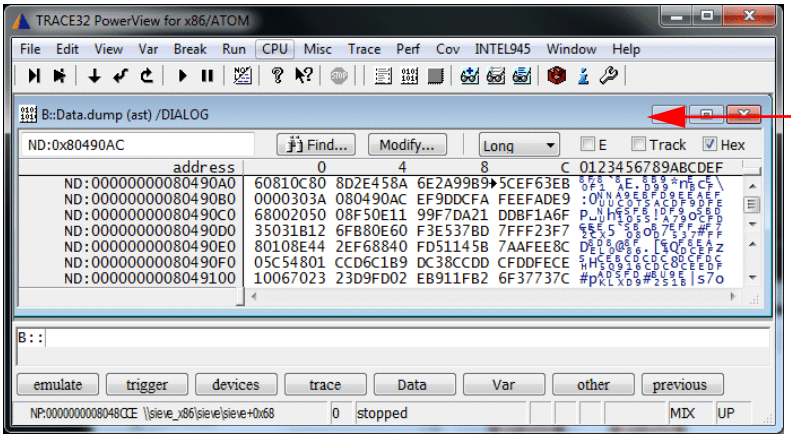
Data.Set *<address>*|*<range>* [**%***<format>*] [**/***<option>*] Modify the memory contents

```
Data.Set 0xC1E2B4E0 0x0000aaaa          ; Write 0x0000aaaa to the logical
                                        ; address 0xC1E2B4E0 in the current
                                        ; address space

Data.Set 0xC1E2B4E0 %Long 0xaaaa        ; Write 0xaaaa as a 32 bit value to
                                        ; the address 0xC1E2B4E0, add the
                                        ; leading zeros automatically

Data.Set 0x6814 %LE %Long 0xaaaa        ; Write 0xaaaa as a 32 bit value to
                                        ; the address 0xC1E2B4E0, add the
                                        ; leading zeros automatically

                                        ; Use Little Endian mode

Data.Set ND:0x88:0xC1E2B4E0 0xaaaa      ; Write 0xaaaa to the logical
                                        ; address 0xC1E2B4E0 of the
                                        ; process with the space ID 0x88
```

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

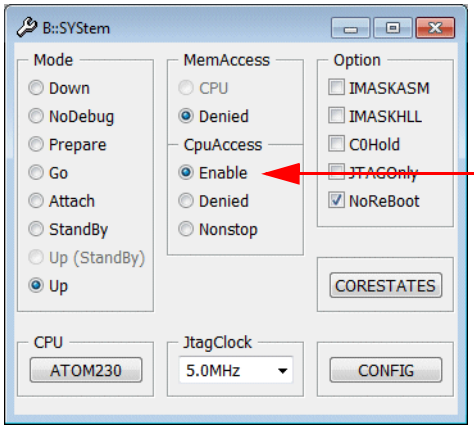


A hatched window frame indicates that the information display is frozen because the cores are executing the program.



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

The Intel® x86/x64 architecture doesn't allow a debugger to read or write memory while the cores are executing the program, but you can activate an intrusive run-time memory access if required.



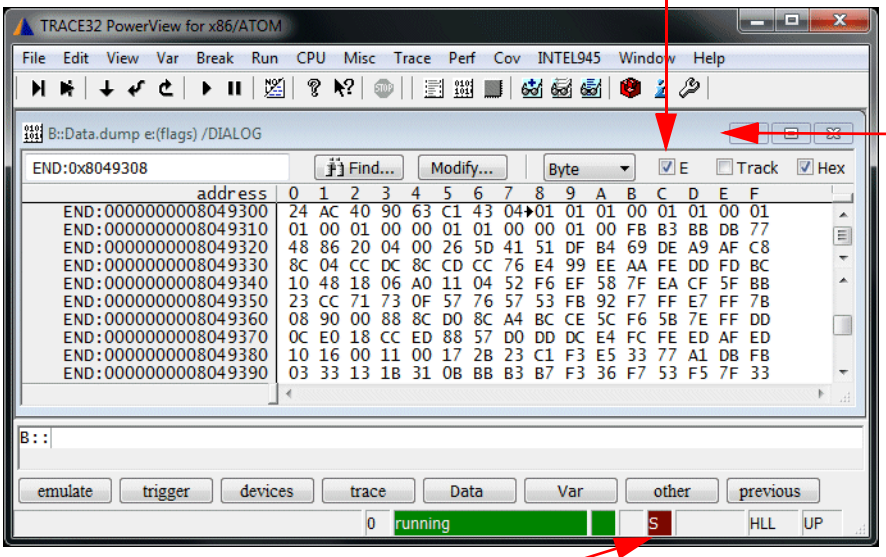
CpuAccess Enable allows an intrusive run-time memory access

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** per core.

cores are executing the program

cores are stopped to allow TRACE32 PowerView to read/write the specified memory

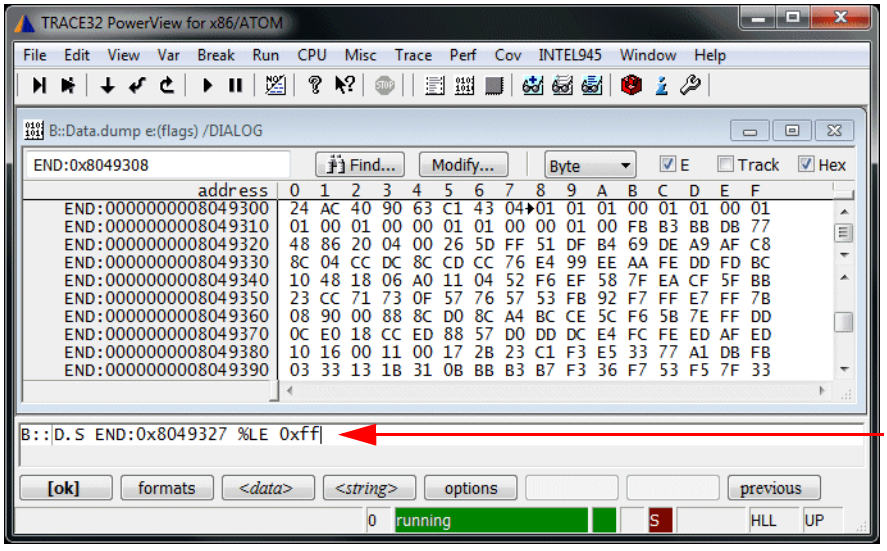
Enable the **E** check box to switch the run-time memory access to ON



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

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

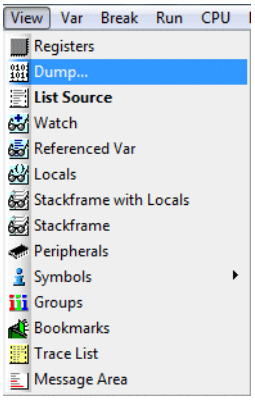
Write accesses to the memory work correspondingly:

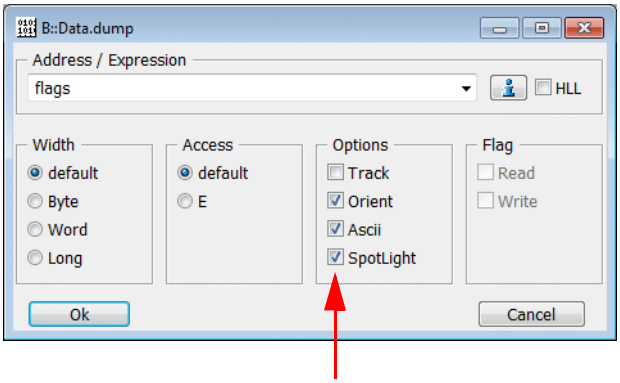


`Data.Set` via run-time memory access with short stop of the program execution

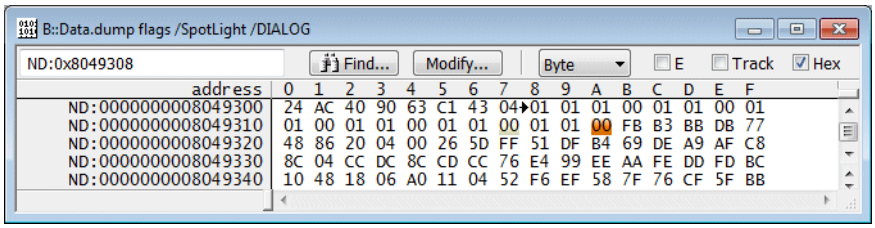
```
SYStem.CpuAccess Enable                 ; Enable the intrusive
                                        ; run-time memory access

;…

Go                                      ; Start program execution

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

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

Colored Display of Changed Memory Contents



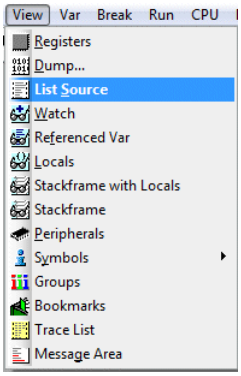


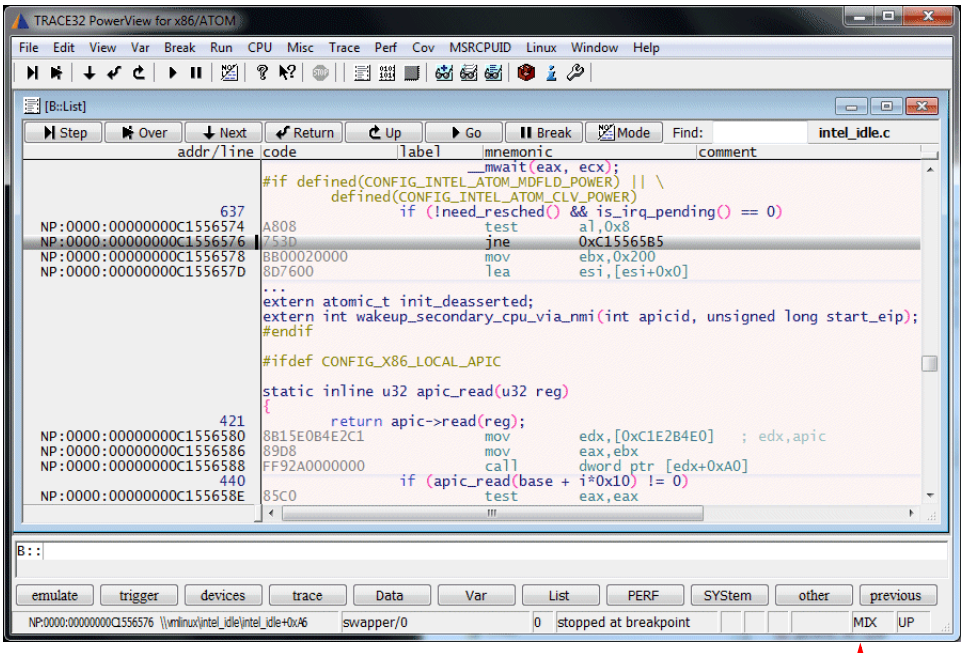
Enable the option **SpotLight** to mark the memory contents changed by the last 4 single steps in orange, older changes being lighter.



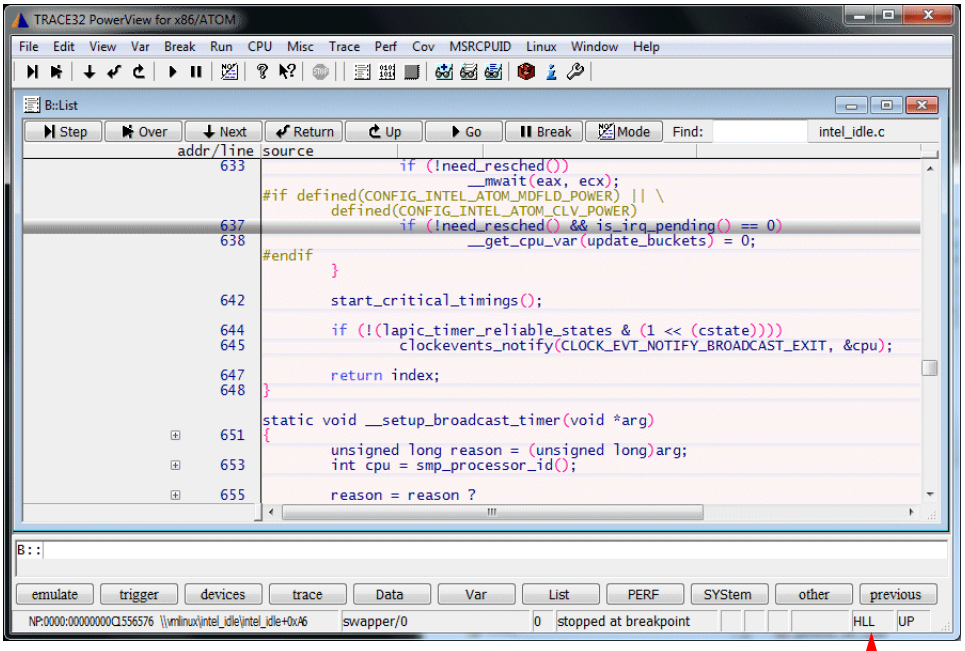

```
Data.dump flags /SpotLight            ; Display a hex dump starting at
                                      ; the address of the label flags

                                      ; Mark changes
```

Displays the Source Listing Around the PC



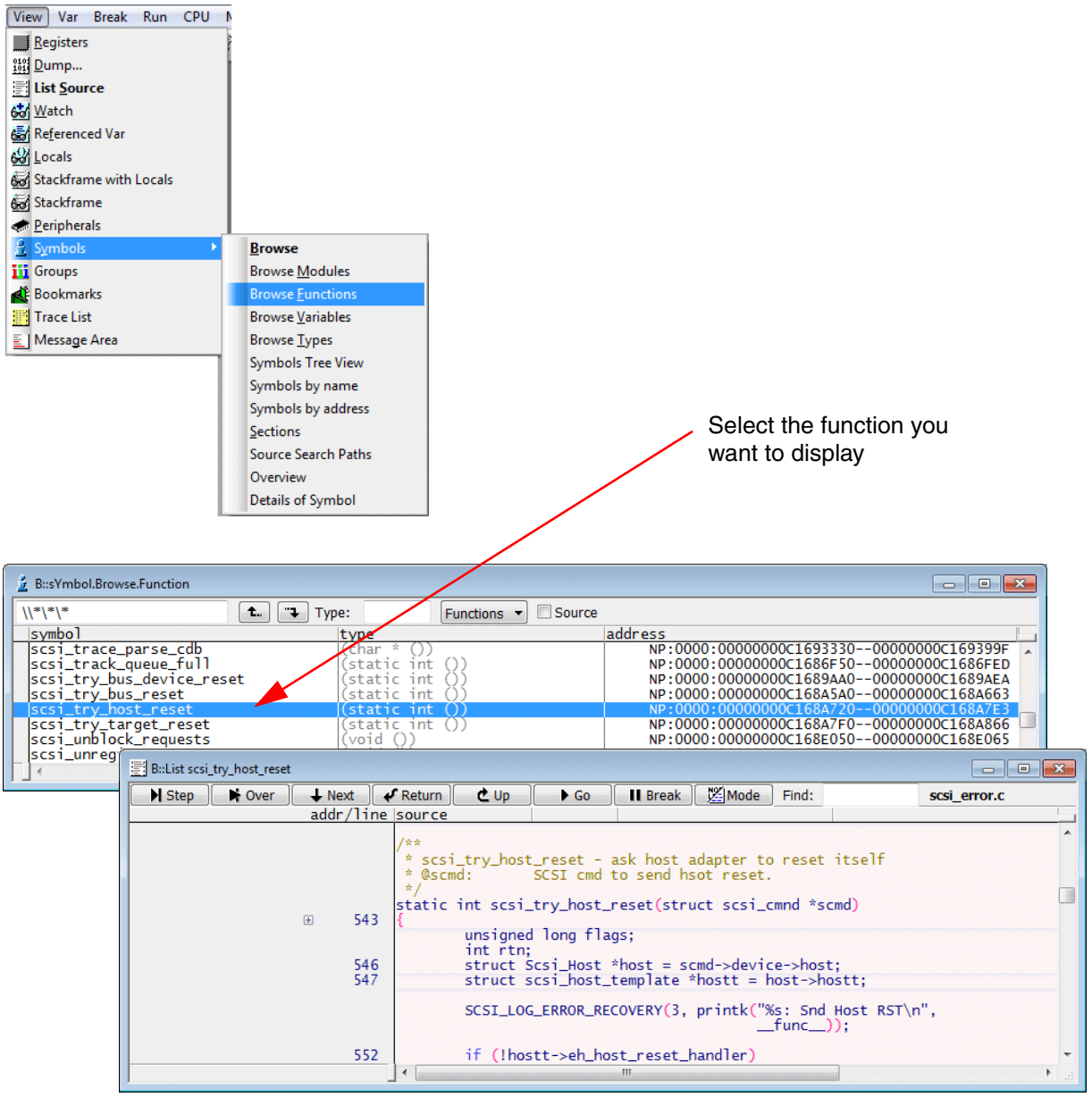


If **MIX** mode is selected for debugging, assembler and HLL information is displayed



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

Displays the Source Listing of a Selected Function



List [*<address>*] [*/<option>*]	Display source listing from the perspective of the selected core
List [*<address>*] **/CORE** *<n>* [*/<option>*]	Display source listing from the perspective of the specified core

```
List                                    ; Display a source listing
                                        ; around the PC of the selected
                                        ; core

List *                                  ; Open the symbol browser to
                                        ; select a function for display

List scsi_try_host_reset                ; Display a source listing of
                                        ; the function scsi_try_host_reset
```

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

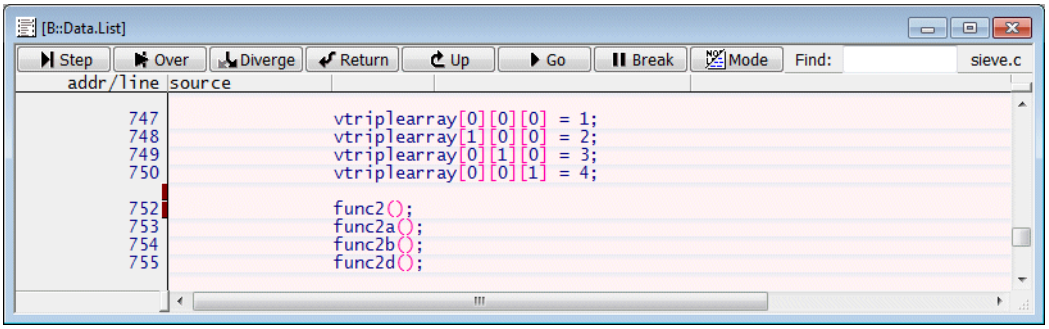
Breakpoint Implementations

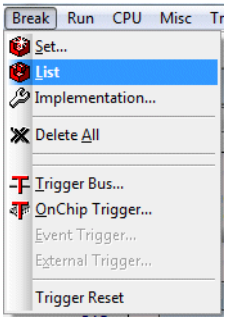
A debugger has two methods to realize breakpoints: Software breakpoints and Onchip Breakpoints.

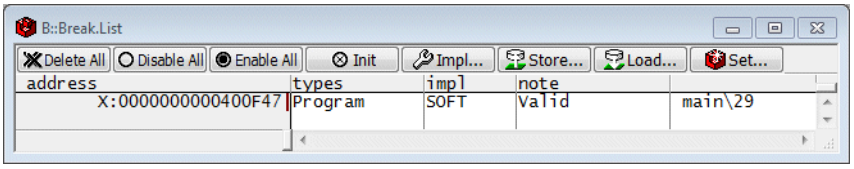
Software Breakpoints in RAM (Program)

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 to stop the program and return the control to the debugger.

The number of software breakpoints is unlimited.





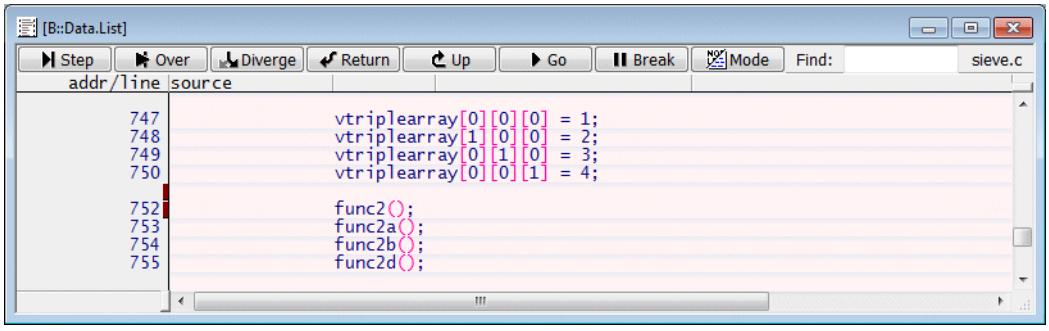


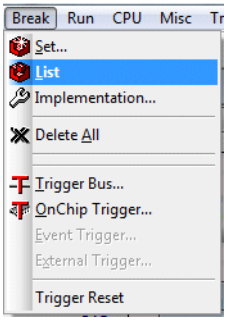
	Please be aware that TRACE32 PowerView always tries to set an Onchip breakpoint, when the setting of a Software Breakpoint fails.

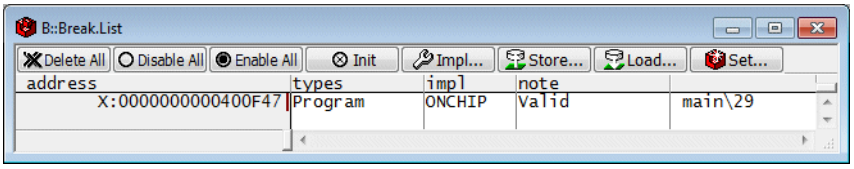
Intel® x86/x64 cores provide a small number of onchip breakpoints in form of breakpoint registers. These onchip breakpoints can be used to set breakpoints to instructions in read-only memory like NOR FLASH.

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 onchip 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.







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

- The access class
- The memory-space ID of the process
- The logical address

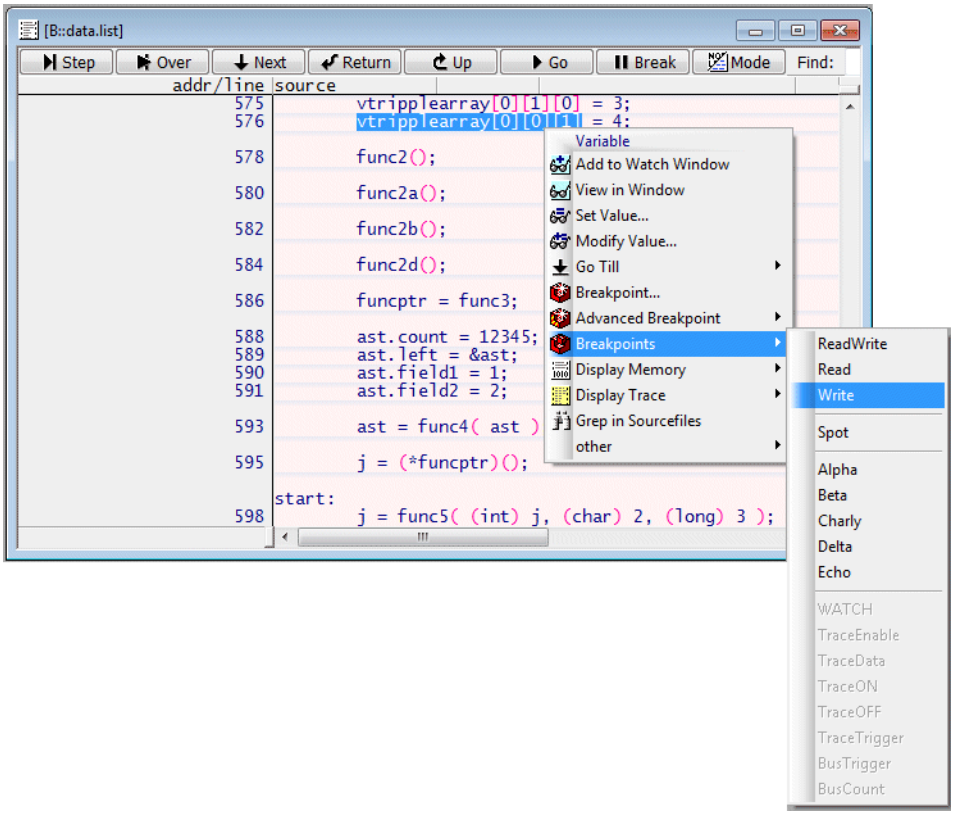
```
<access_class>:<space_id>:<logical_address>
 NP:0088:00000000C168A720
```

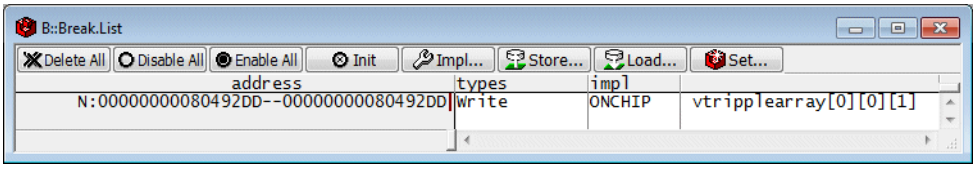
The onchip breakpoints of Intel® x86/x64 cores store only the logical address, but not the memory space ID. As a result an identical logical address within another process can also result in a breakpoint hit.

For details on the TRACE32 PowerView address scheme of operating systems that uses dynamic memory management to handle processes/tasks refer to your **OS manual** (rtos_<os>.pdf).

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

Onchip breakpoints can be used to stop the core(s) at a write access or 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 logical address result in a breakpoint hit.

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

Onchip Breakpoints for Intel® x86/x64

The list on this page gives an overview of the availability and the usage of the **onchip breakpoints**. The following notations are used:

- **Onchip breakpoints:** Total amount of available onchip breakpoints.
- **Program breakpoints:** Number of onchip breakpoints that can be used to set Program breakpoints into NOR FLASH.
- **Read/Write breakpoints:** Number of onchip breakpoints that stop the program when a write or read/write to a certain address happens.
- **Data value breakpoint:** Number of onchip data breakpoints that stop the program when a specific data value is written to an address or when a specific data value is read from an address.

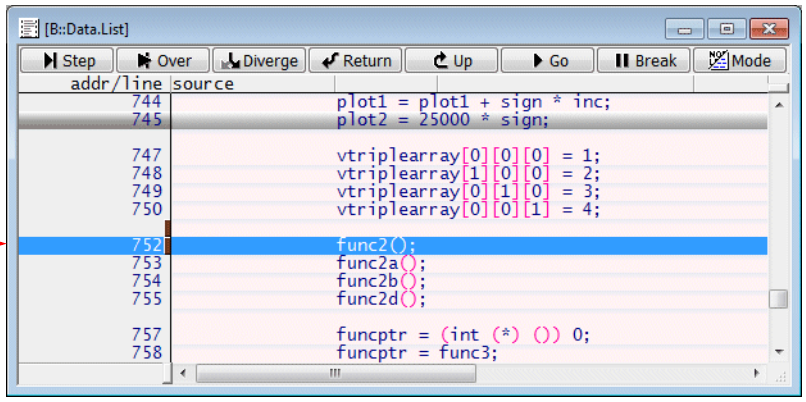
Family	Onchip Breakpoints	Instruction Breakpoints	Read/Write Breakpoint	Data Value Breakpoints
Intel® x86/x64	4	4 single address	4 Write or Read/Write single address or ranges up to 8 bytes (aligned)	—

Breakpoint Types

TRACE32 PowerView provides the following breakpoint types for standard debugging.

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

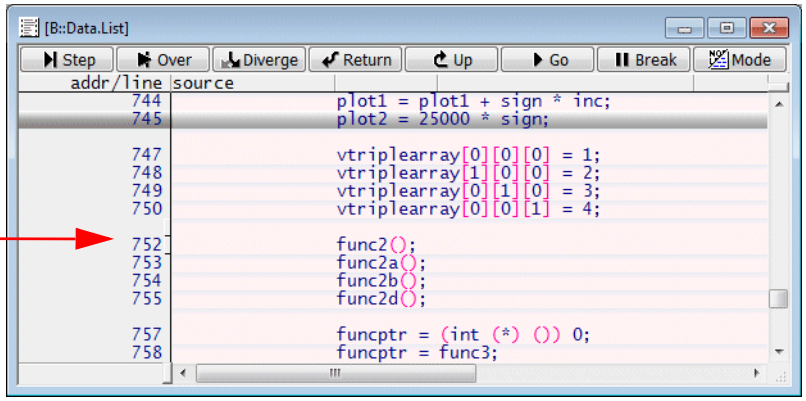
Set a Program breakpoint by a left mouse **double-click** to the instruction



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

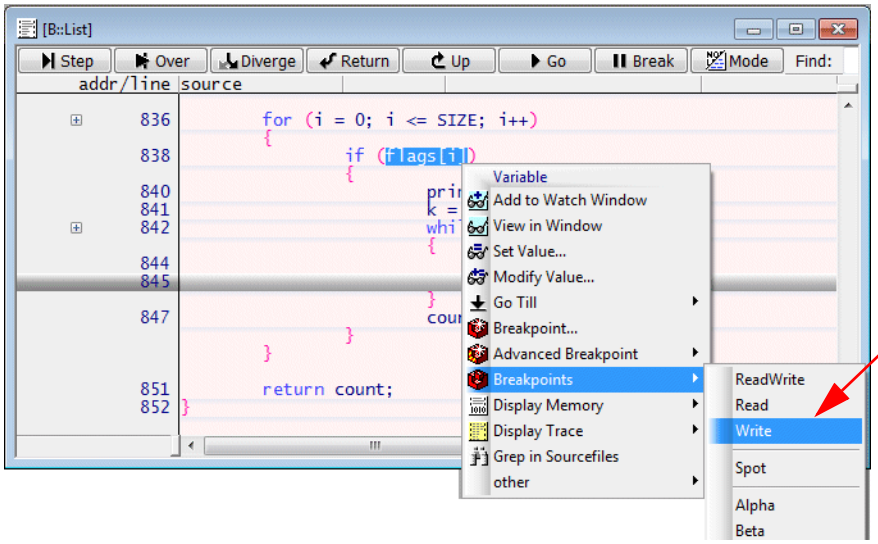
The program stops before the instruction marked by the breakpoint is executed (break-before-make).

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



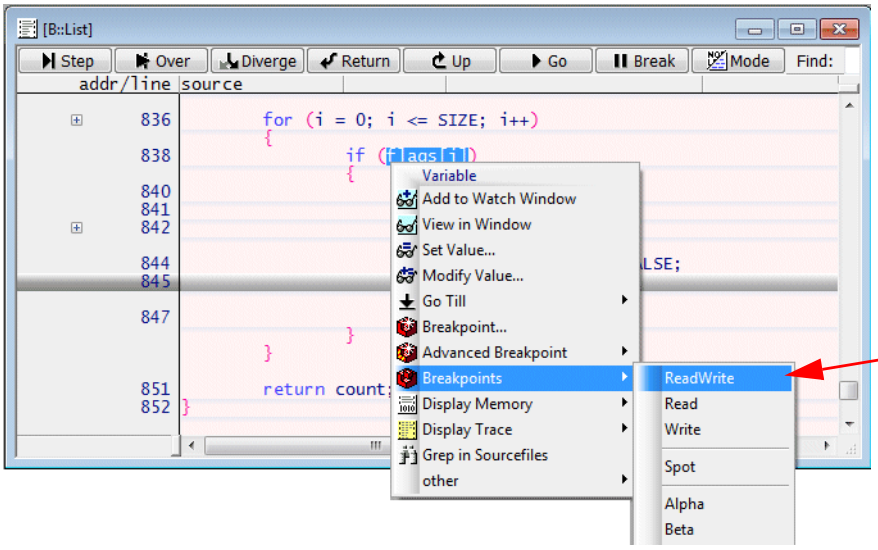
Break.Set *<address>* **/Program** [**/DISable**]	Set a Program breakpoint to the specified address. The Program breakpoint can be disabled if required.

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

Break.Set func1 /Program              ; set a Program breakpoint to the
                                      ; entry of function func1
                                      ; (first address of function func1)

Break.Set func1+0x1c /Program         ; set a Program breakpoint to the
                                      ; instruction at address
                                      ; func1 plus 28 bytes


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 function func17
                                      ; diable Program breakpoint

Break.List                            ; list all breakpoints
```

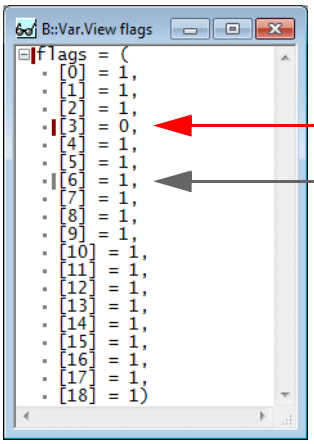


All cores are stopped after a write access to the variable (break-after-make)

Please be aware that you have to use a ReadWrite breakpoint if you want to stop the program execution on a read access if you use the Intel® x86/x64 architecture. Pure Read breakpoints are not provided.



All cores are stopped after a read or write access to the variable (break-after-make)



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

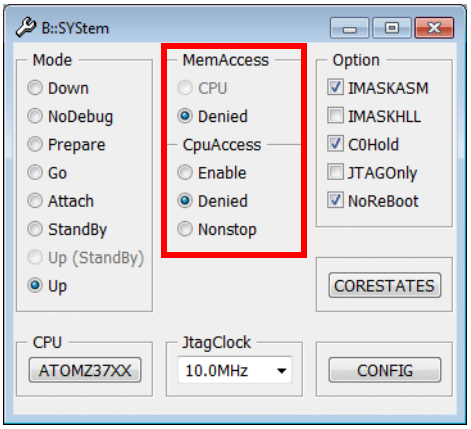
A small **grey breakpoint indicator**
marks a disabled Read/Write breakpoint.

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

; allow HLL expression to specify breakpoint
Var.Break.Set *<hll_expression>* **/Write | /ReadWrite** [**/DISable**]

```
Break.Set 0x0B56 /ReadWrite

Break.Set ast /Write

Break.Set vpchar+5 /ReadWrite /DISable

Var.Break.Set flags[3..5] /ReadWrite

Var.Break.Set ast->count /ReadWrite /DISable

Break.List
```

Breakpoint Setting at Run-time



If **MemAccess** and **CPUAccess** is Denied breakpoints can only be set when the program execution is stopped.

Breakpoints after Reset/Power Cycle

To understand this topic you have to be aware of the following:

- TRACE32 can only set breakpoints when the program execution is stopped.
- The onchip breakpoint logic is reset on a chip reset/power cycle. As a result the currently set onchip breakpoints are lost.
- If the target under debug is reset/re-powered all software breakpoints are lost.

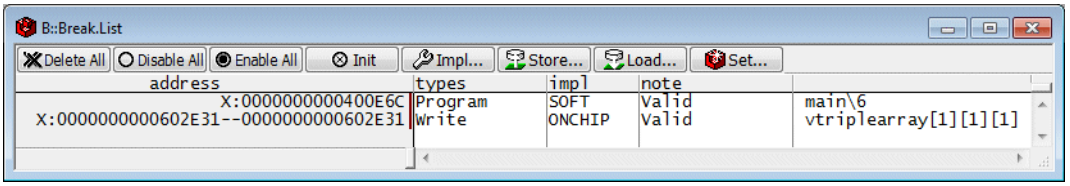
TRACE32 has the following standard behavior: The breakpoint list in TRACE32 PowerView is not deleted, when TRACE32 detects a reset/power cycle. Thus TRACE32 can set all listed breakpoints again when the program execution is stopped and restarted.

If TRACE32 detects a reset/power cycle and the core(s) immediately starts the program execution, it is highly likely that all listed breakpoints are lost.

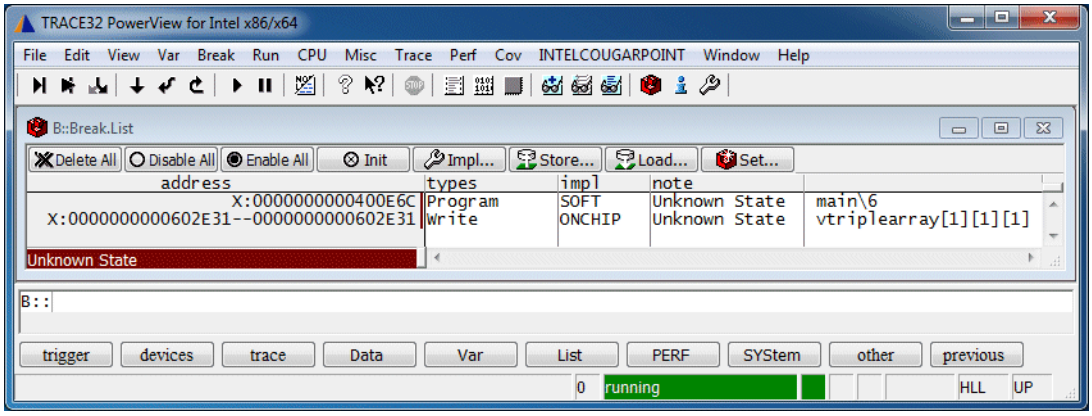
To indicate this loss, TRACE32 changes the state of the breakpoints from **Valid** to **Unknown State**. The state of the breakpoints is displayed in the **note** field of the breakpoint listing.

All listed breakpoints become **Valid** again, when the program execution is stopped the next time.

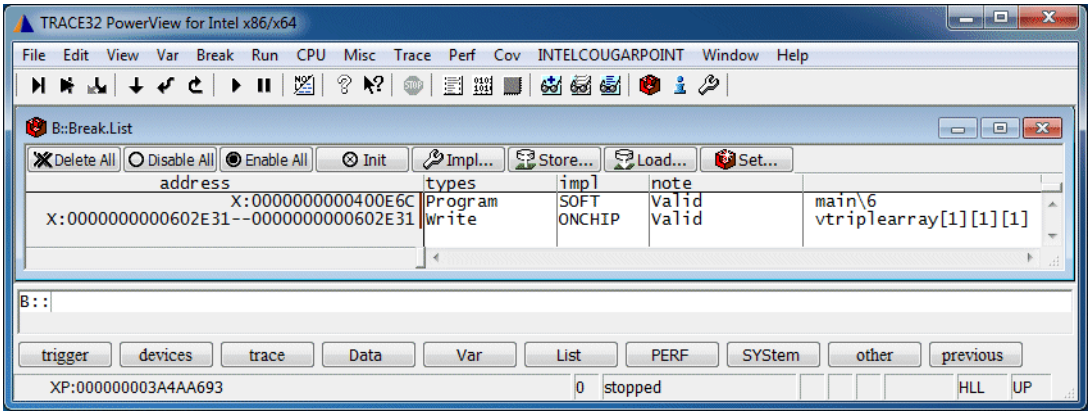
Example:



If breakpoints are listed within TRACE32 PowerView they change to the **Unknown State** when TRACE32 detects that the target is reset/re-powered and the core(s) immediately starts the program execution.

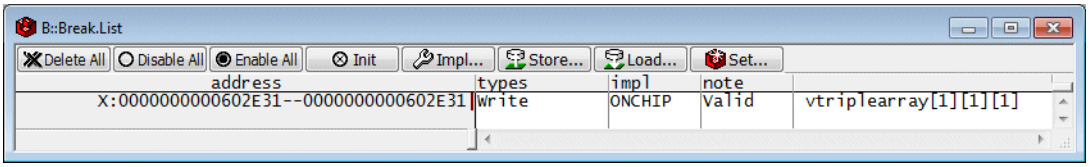


As soon as the program execution is stopped, all listed breakpoints become **Valid** again, because they will be set again when the program execution is restarted.



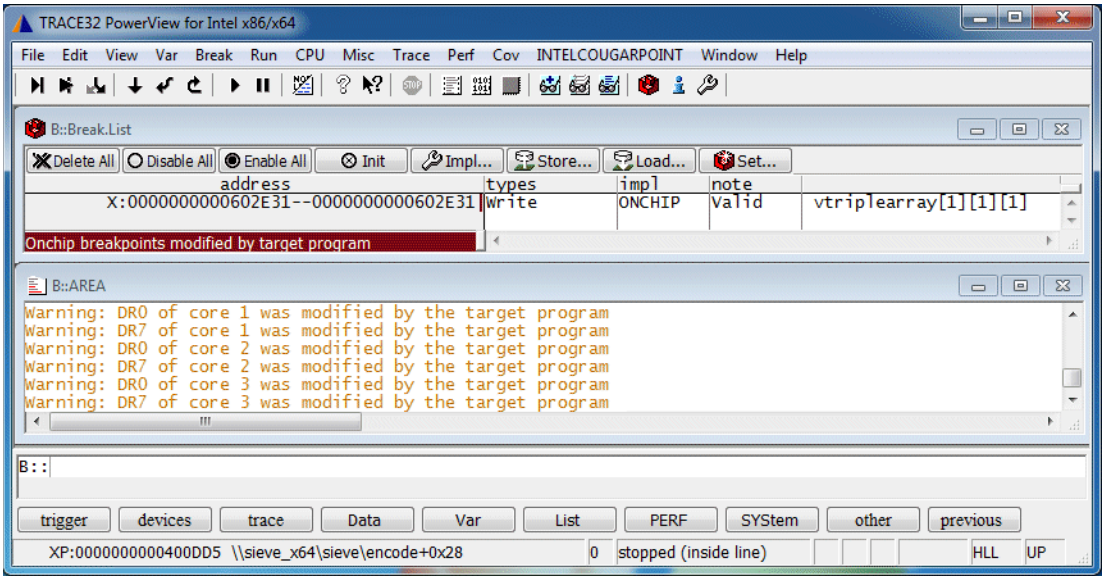
Onchip Breakpoints Changed by Target Program

Both, the debugger and the program running on the target can set/delete the onchip breakpoints.



If the program execution stops and TRACE32 detects an inconsistent programming of an onchip breakpoint, the error message **Onchip breakpoints modified by target program is displayed** in the **Break.List** window. The TRACE32 Message area provides details.

If TRACE32 stops due to an onchip breakpoint set by the target program, ***stopped by DRx breakpoint*** is displayed in the Debug field of the TRACE32 state line



Please be aware that TRACE32 programs all listed breakpoints every time the program execution is stopped and restarted. If you want to keep the onchip breakpoints set by the target program you have to delete/disable all onchip breakpoints within TRACE32.

If you want to use the onchip breakpoints by the target program for another purpose than debugging, use **SYStem.Option.IGnoreDEbugReDirections ON** to advise TRACE32 to ignore the onchip breakpoints.

Real-time Breakpoints vs. Intrusive Breakpoints

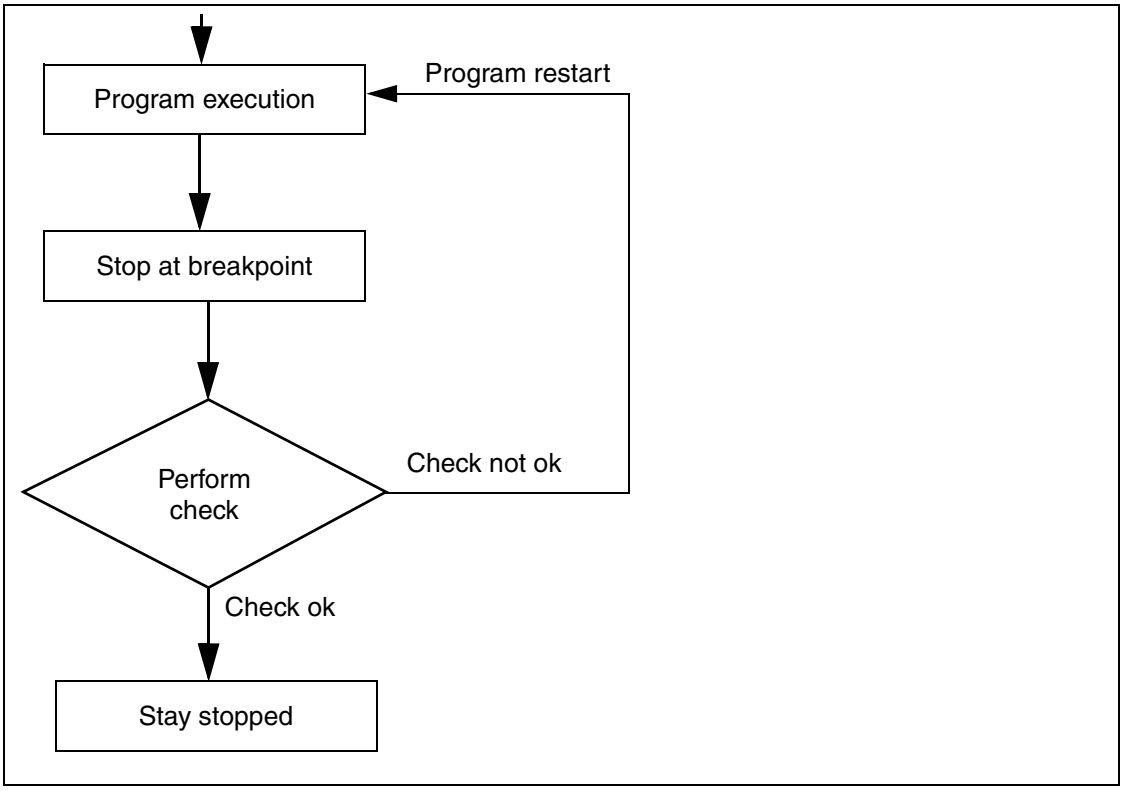
TRACE32 PowerView offers in addition to the basic breakpoints (Program/Read/Write) also complex breakpoints. Whenever possible these breakpoints are implemented as real-time breakpoints.

Real-time breakpoints do not disturb the real-time program execution on the cores, but they require a complex on-chip breakpoint logic.

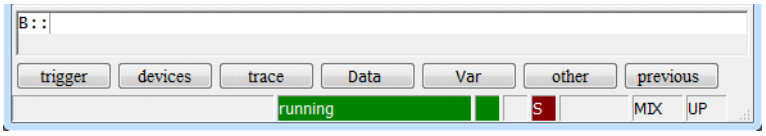
If the onchip breakpoint logic of a core does not provide the required features or if Software breakpoints are used, TRACE32 has to implement an intrusive breakpoint.

Intrusive breakpoints

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



Each *Stop at breakpoint* suspends the program execution for at least 1 ms.

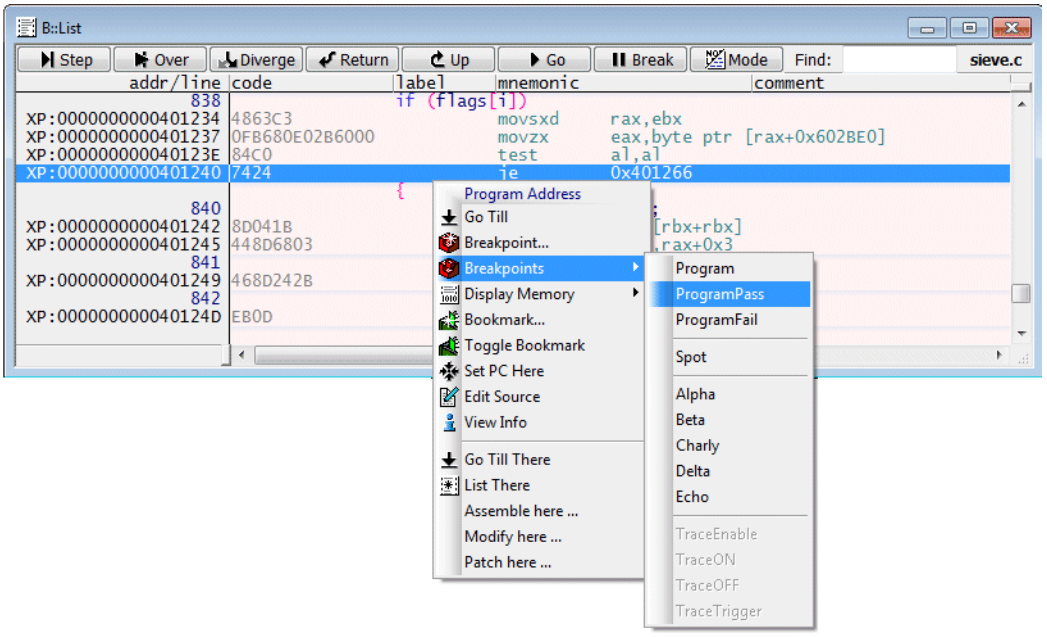


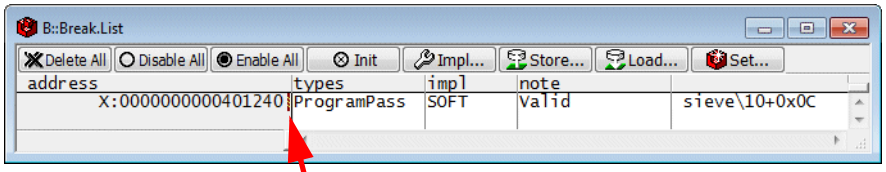
The (short-time) display of a red S in the **Debug Activity** field of the TRACE32 state line indicates that an intrusive breakpoint was hit.

Intrusive breakpoints are marked by a special breakpoint indicator: 

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

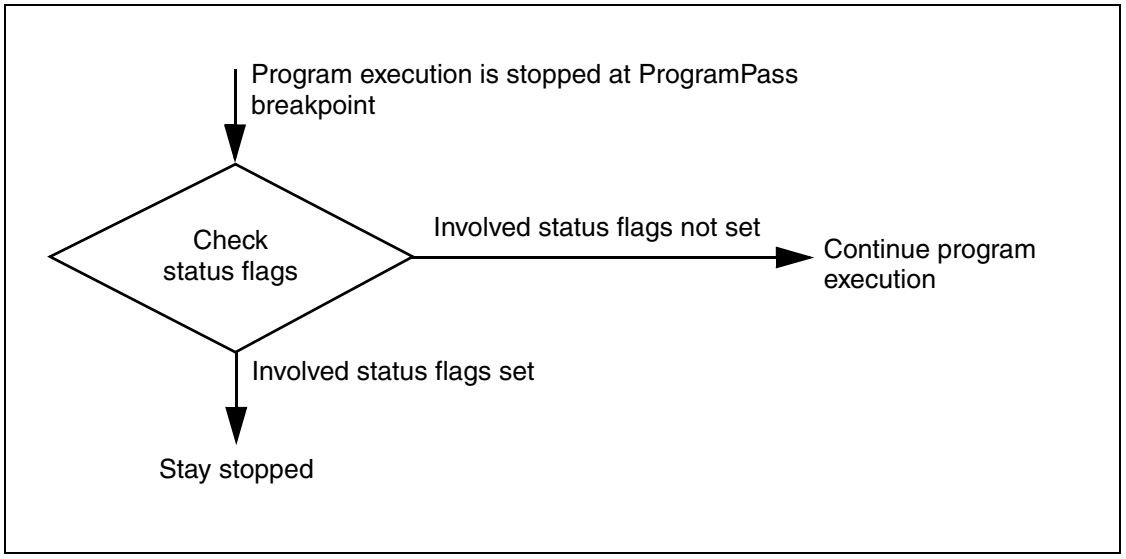
Example: Stop the program execution, when the instruction **je 0x401266** instruction passes (indicated by Zero Flag).



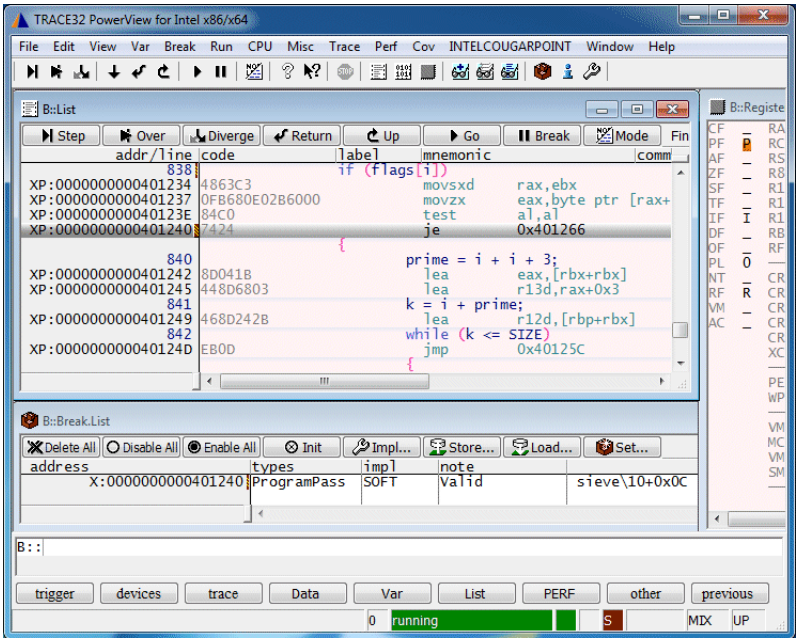


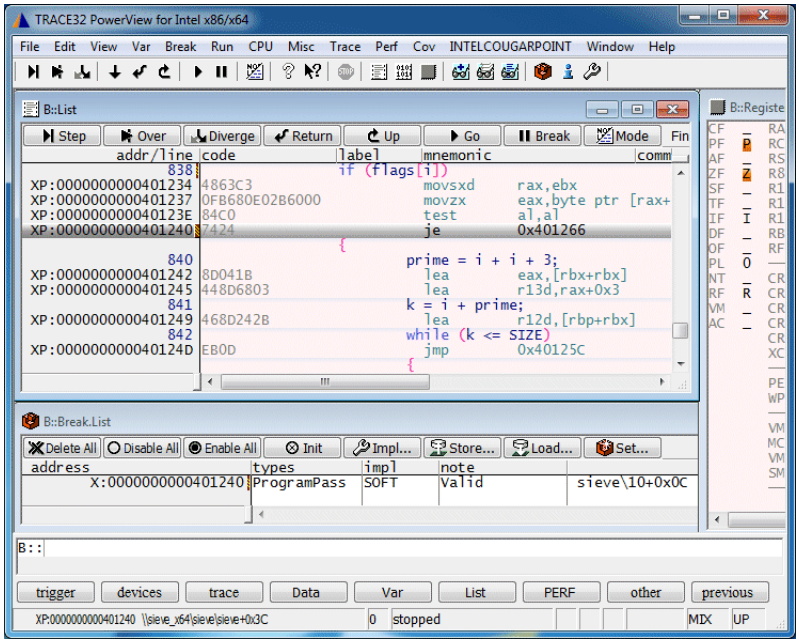
Intrusive breakpoints are marked with an intrusive breakpoint indicator

The **ProgramPass** breakpoint behaves as follows:



Each suspend to check the status flags takes at least 1. ms. This is why the red S is displayed in the Debugger Activity field of the TRACE32 PowerView state line.





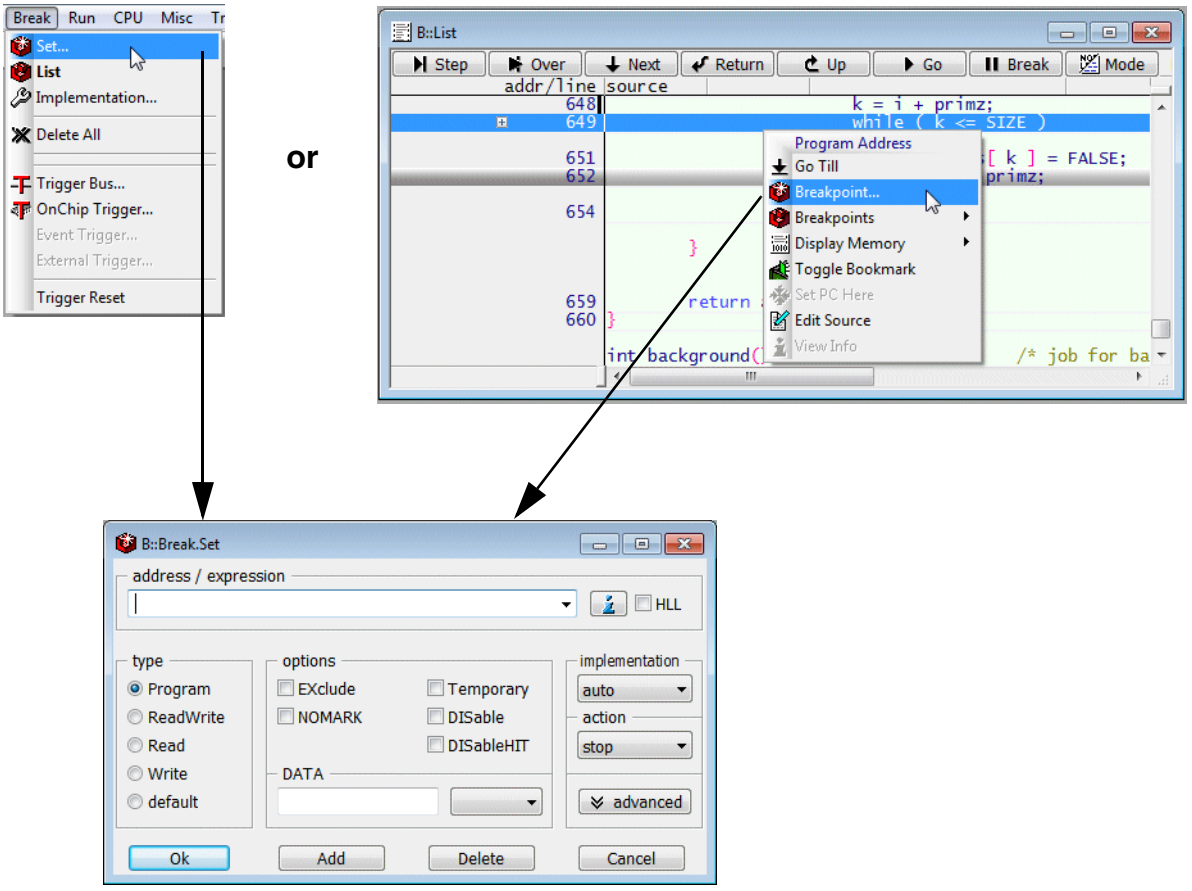
Break.Set *<address>|<range>* **/ProgramPass**

Break.Set *<address>|<range>* **/ProgramFail**

```
Break.Set 0x401240 /ProgramPass
```

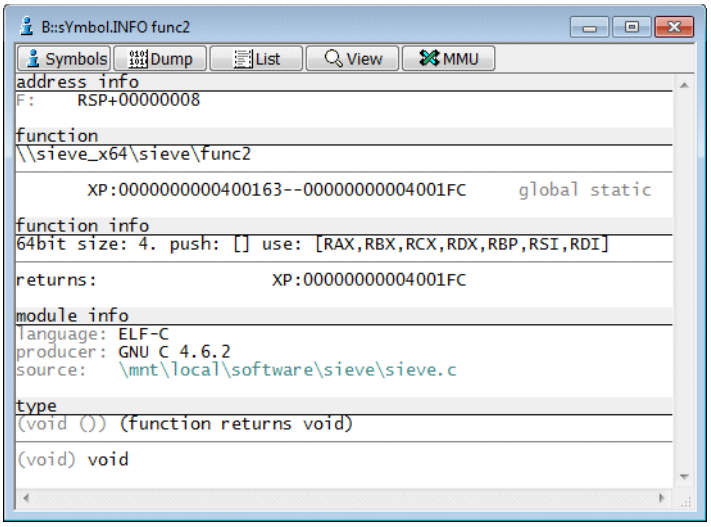
Break.Set Dialog Box

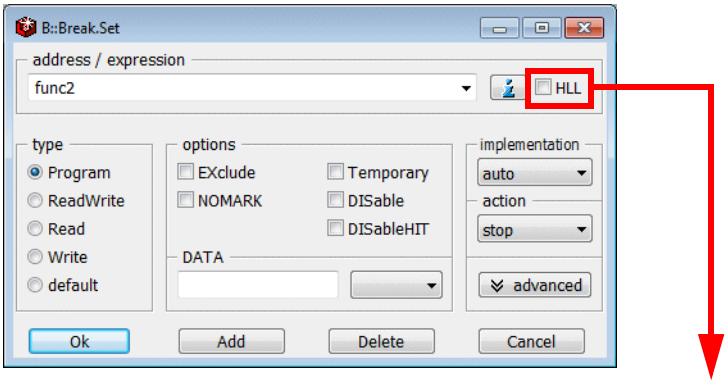
There are two standard ways to open a **Break.Set** dialog.



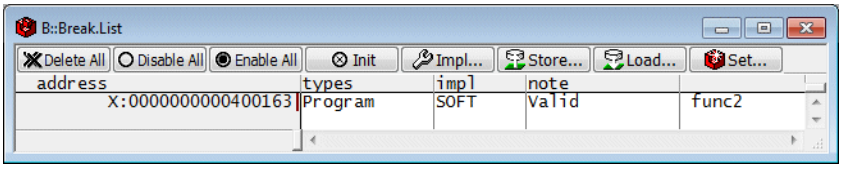
Function Name/HLL Check Box OFF

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



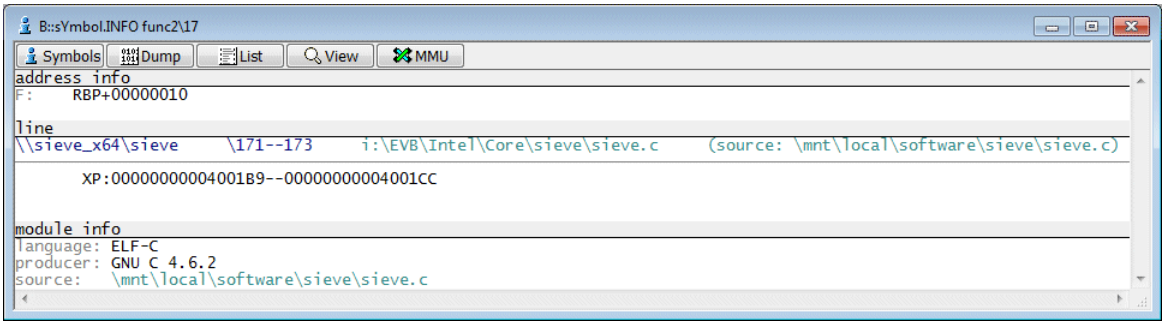


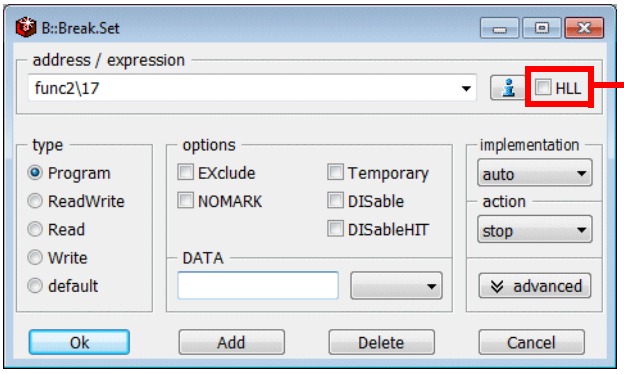
Program breakpoint is set to the function entry (first address of the function)



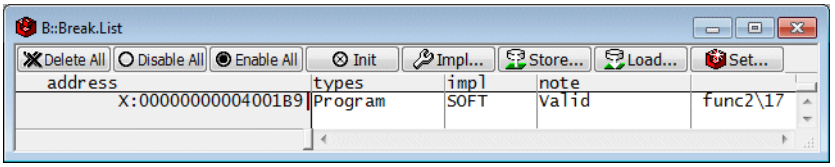

```
Break.Set func2
```

```
sYmbol.INFO func2\17                    ; display symbol information
                                        ; for 17th program line in
                                        ; function func2
```



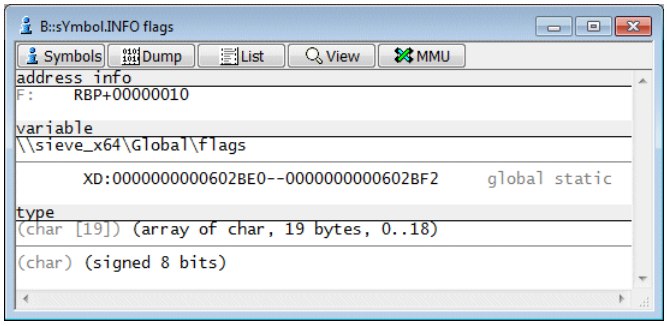


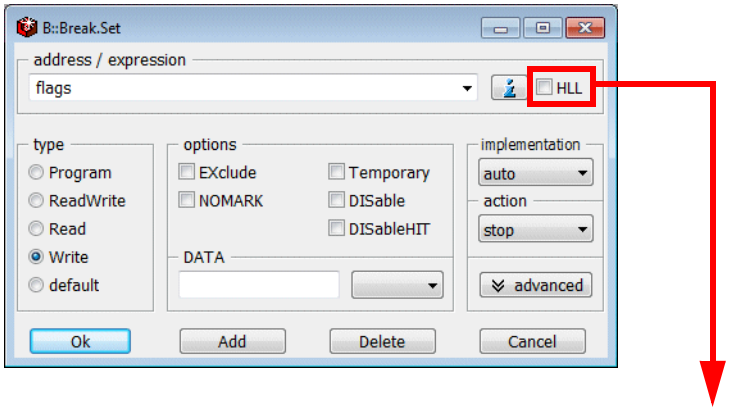
Program breakpoint is set to the first assembler instruction generated for the program line number



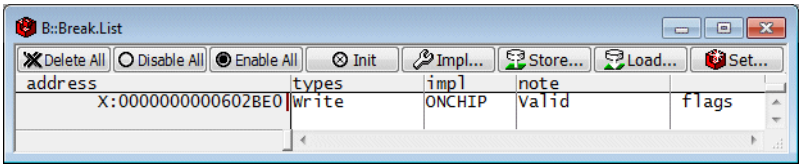

```
Break.Set func2\17
```

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



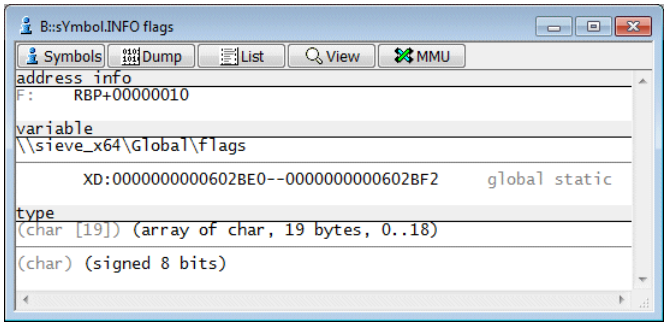


Selected breakpoint is set to the start address of the variable

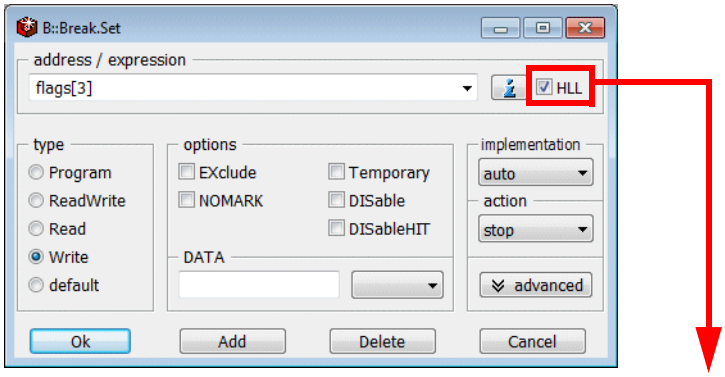



```
Break.Set flags
```

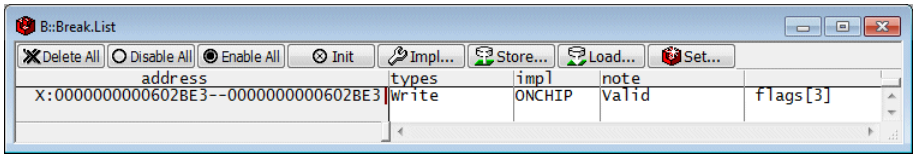
```
sYmbol.INFO flags                       ; display symbol information
                                        ; for variable flags
```



- Intel® x86/x64: The on-chip breakpoint logic supports ranges of the following sizes for Write and ReadWrite breakpoints: 1-, 2-, 4-, 8-bytes (aligned). If the specified breakpoint fulfills this requirement, the breakpoint is accepted.

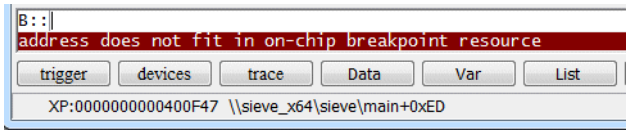


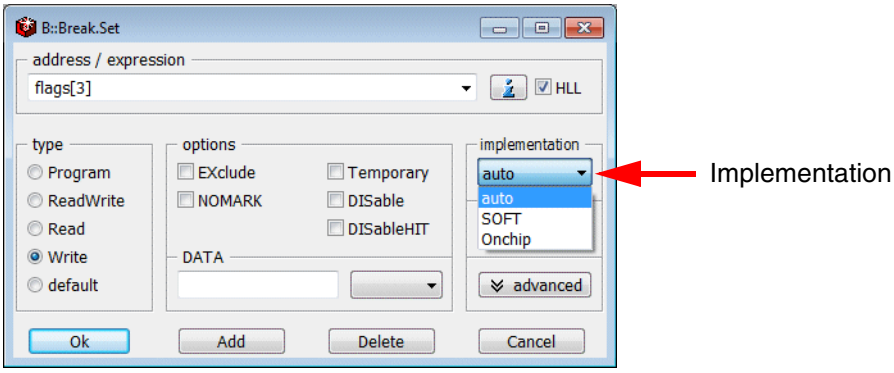
Selected breakpoint is set to the address range used by the HLL-expression



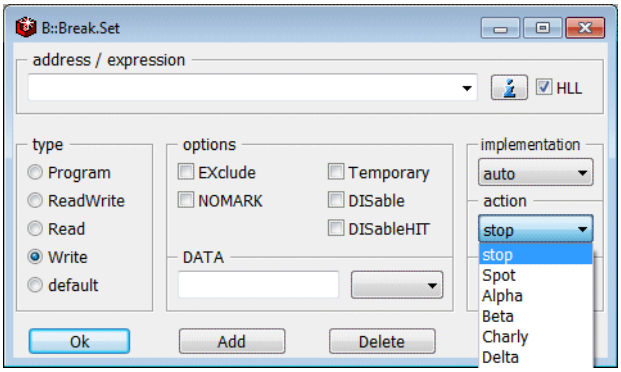

```
Var.Break.Set flags[3]
```

- otherwise the breakpoint is rejected with an error message.





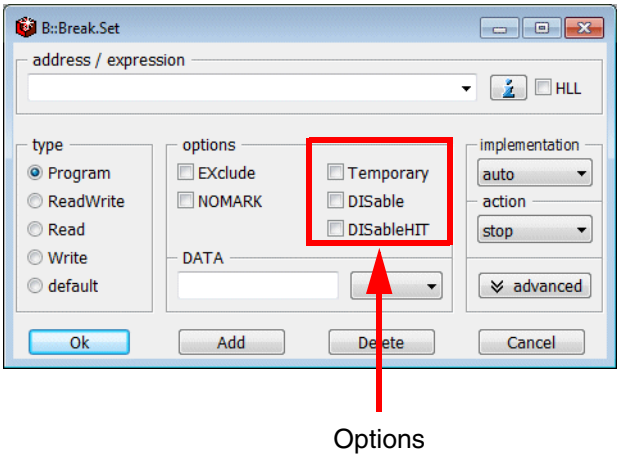
Implementation	
auto	Use breakpoint implementation predefined in TRACE32 PowerView.
SOFT	Implement breakpoint as Software breakpoint.
Onchip	Implement breakpoint as Onchip breakpoint.



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 (50..100ms) 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.

Alpha, Beta, Charly, Delta and Echo breakpoints are not used for Intel® x86/x64.



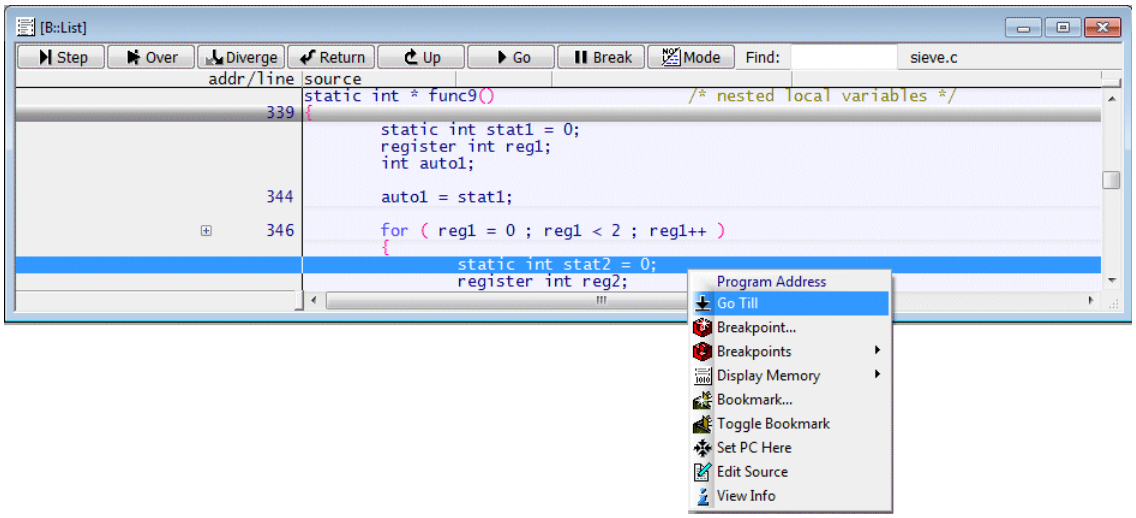
Temporary	**OFF:** Set a permanent breakpoint (default). **ON:** Set a temporary breakpoint. All temporary breakpoints are deleted the next time the core(s) stops the program execution.
DISable	**OFF:** Breakpoint is enabled (default). **ON:** Set breakpoint, but disabled.
DISableHIT	**ON:** Disable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

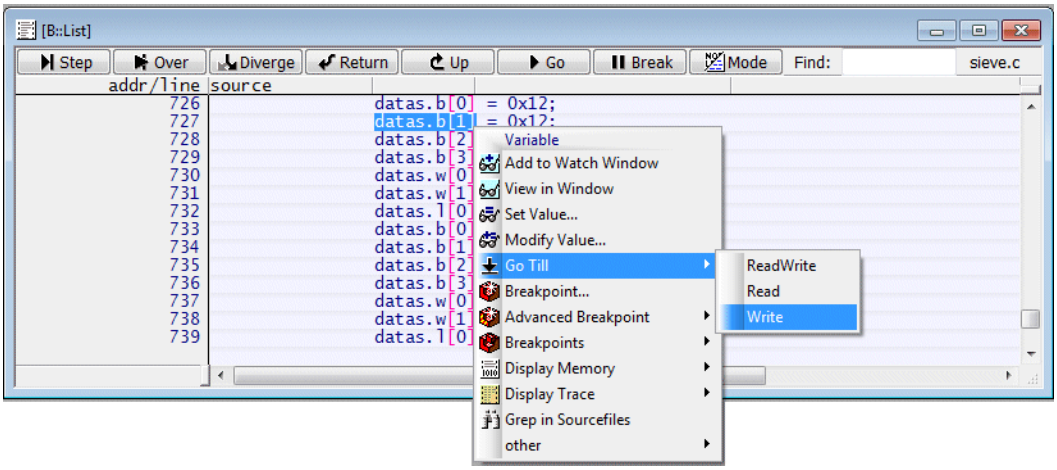
- **Go Till**



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
```

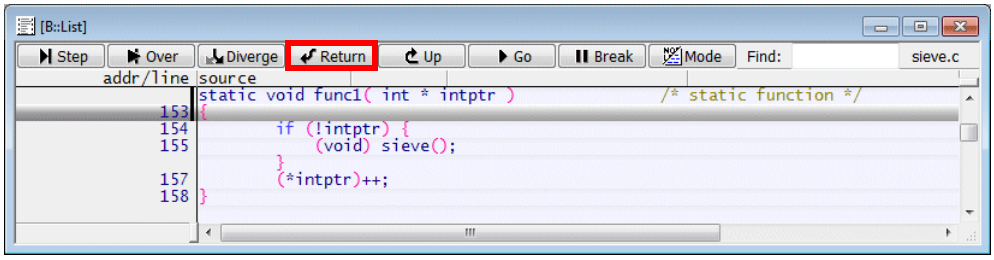
- **Go Till -> Write**



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

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

- **Go.Return and similar commands**

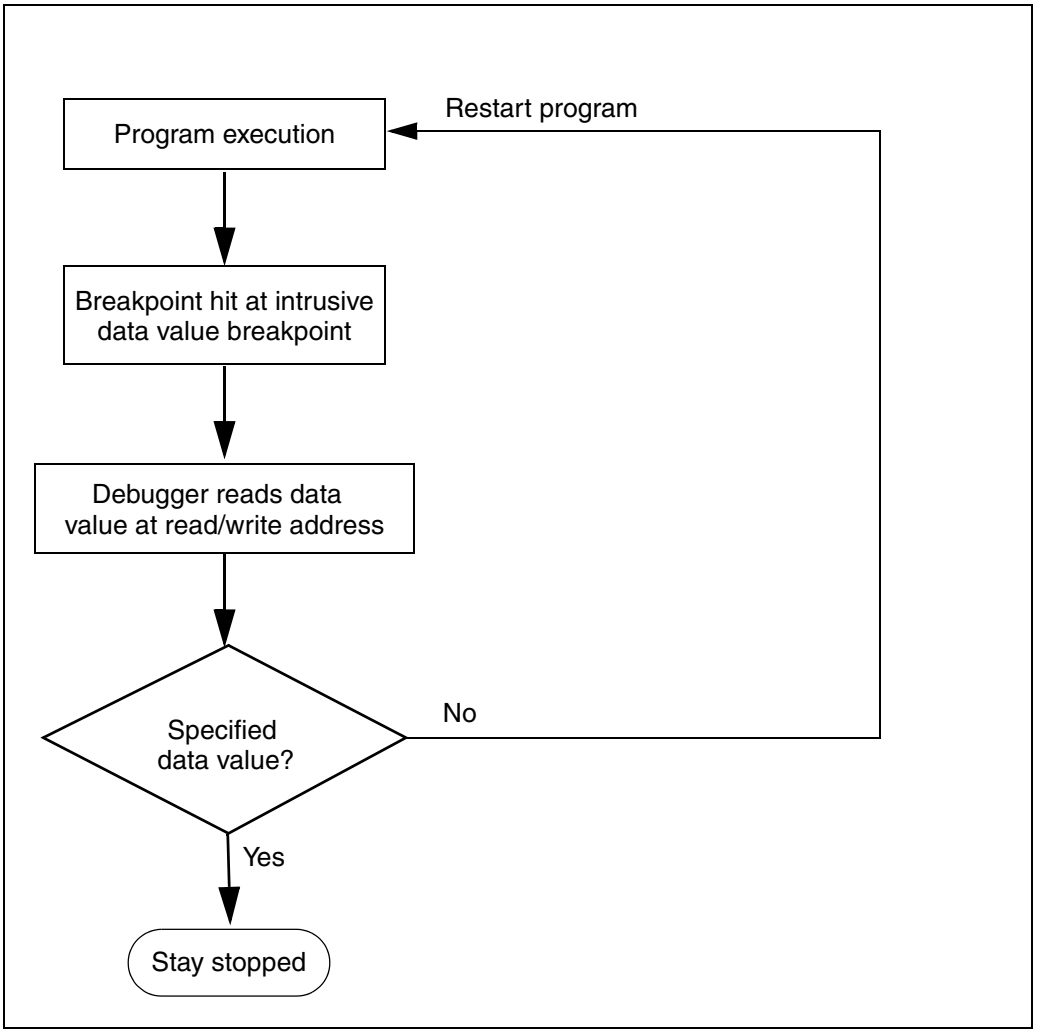


Go.Return

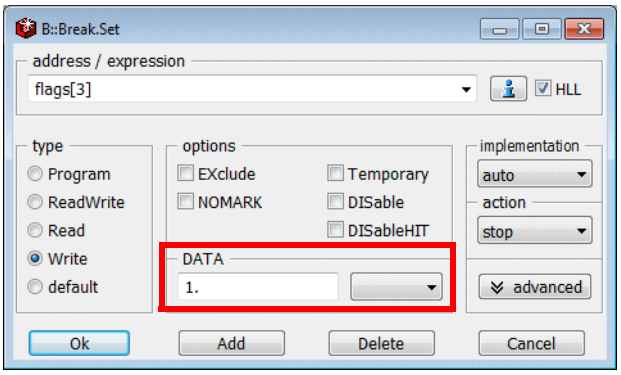
```
; set a temporary breakpoint to the last instruction of the current
; function and start the program execution
Go.Return
```

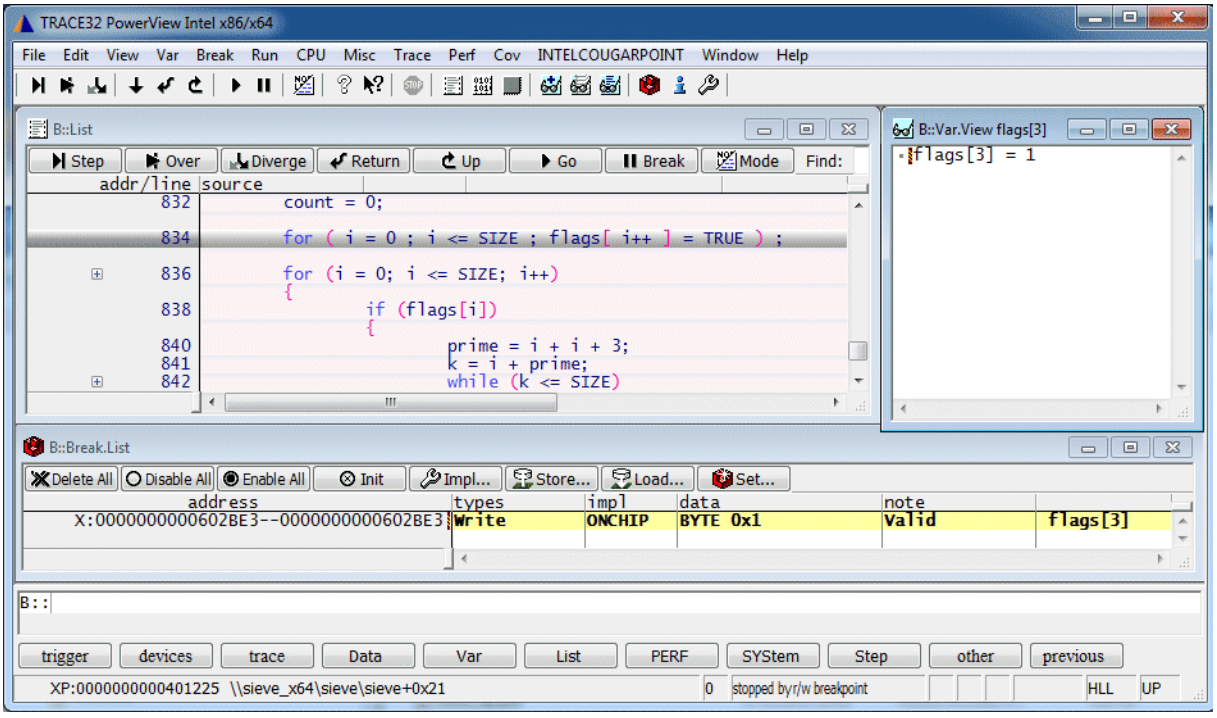
The DATA field offers the possibility to combine a Read/Write breakpoint with a specific data value. Data breakpoints are implemented as intrusive breakpoints for Intel® x86/x64. TRACE32 PowerView allows inverted data values for intrusive data value breakpoints.

An intrusive DATA breakpoint behaves as follows:



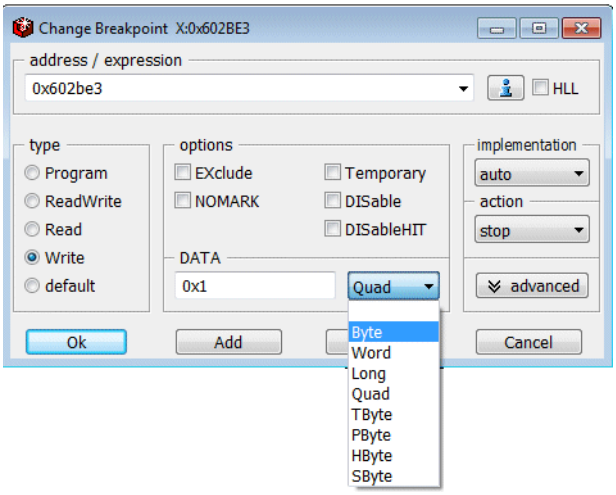
Example 1: Stop the program execution if a 1 is written to flags[3].





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

If an address or symbol is used the user has to specify the access width.



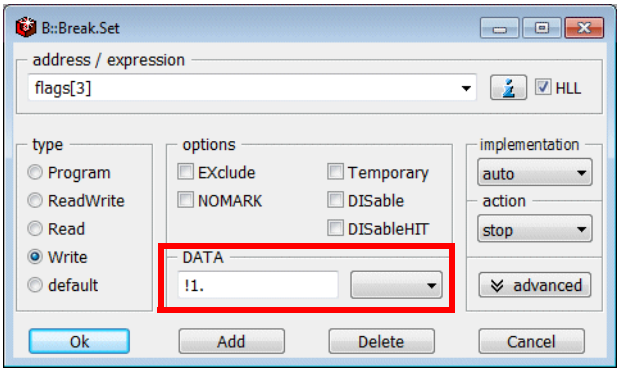
Units	
Byte	8-bit
Word	16-bit
Long	32-bit
Quad	64-bit
TByte	24-bit (TriByte)
PByte	40-bit (PentaByte)
HByte	48-bit (HexaByte)
SByte	56-bit (SeptuaByte)

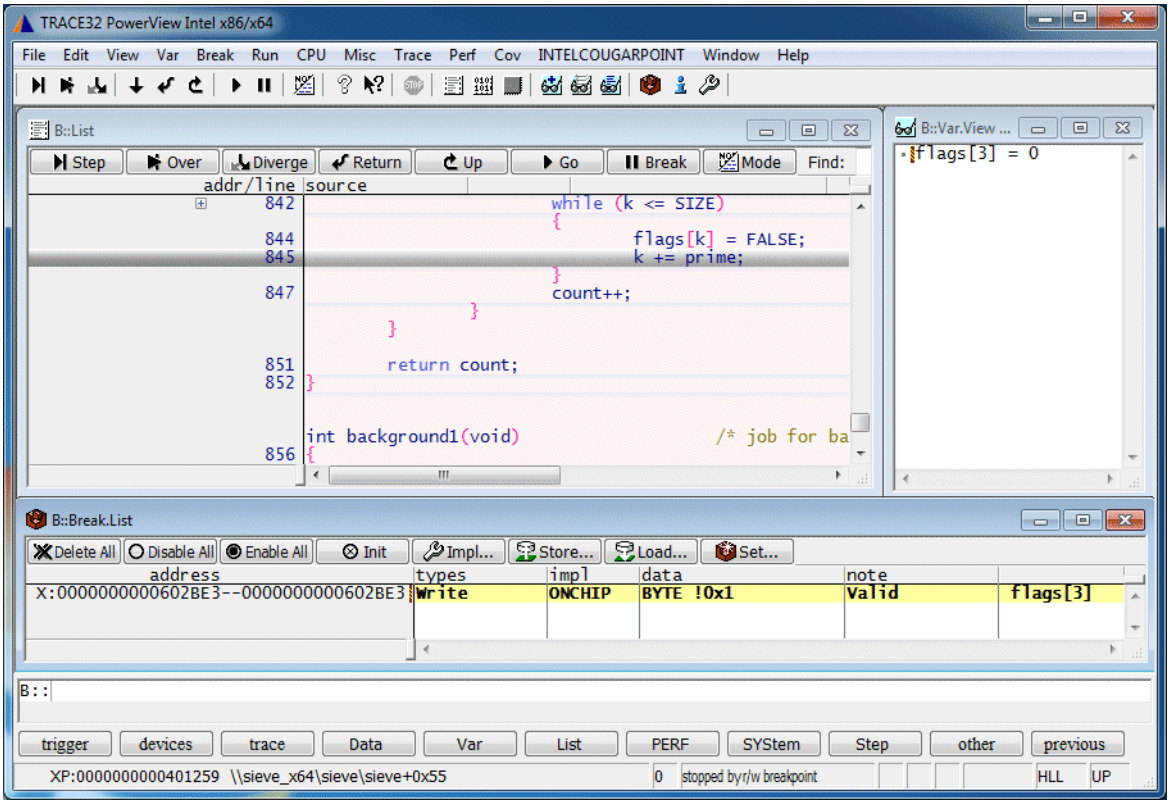
Var.Break.Set *<hll_expression>* **/[Write | ReadWrite] /DATA.auto** *<value>*

Break.Set *<address>* | *<range>* **/[Write | ReadWrite] /DATA.[Byte | Word | Quad]** *<value>*

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

Break.Set 0x602be3 /Write /DATA.Byte 0x1
```

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



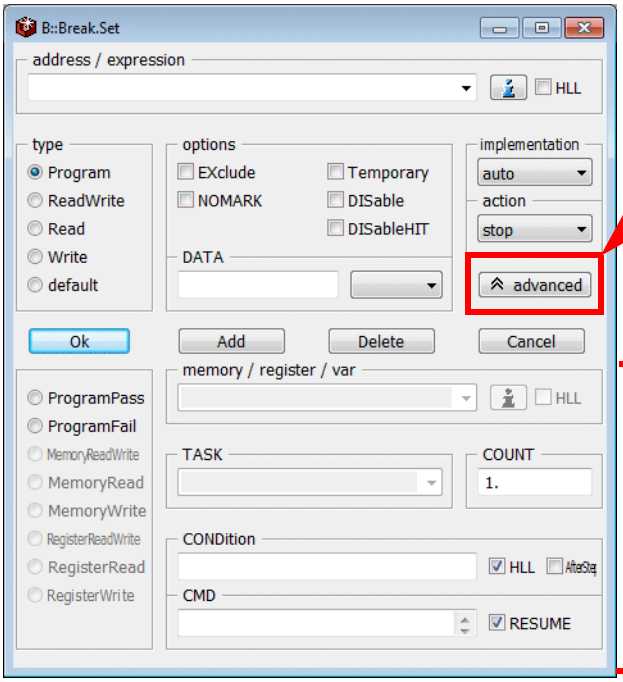


Var.Break.Set *<hll_expression>* **/[Write | ReadWrite] /DATA.auto !***<value>*

Break.Set *<address>* | *<range>* **/[Write | ReadWrite] /DATA.[Byte | Word | Quad] !***<value>*

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

Break.Set 0x602be3 /Write /DATA.Byte !0x1
```

Advanced Breakpoints



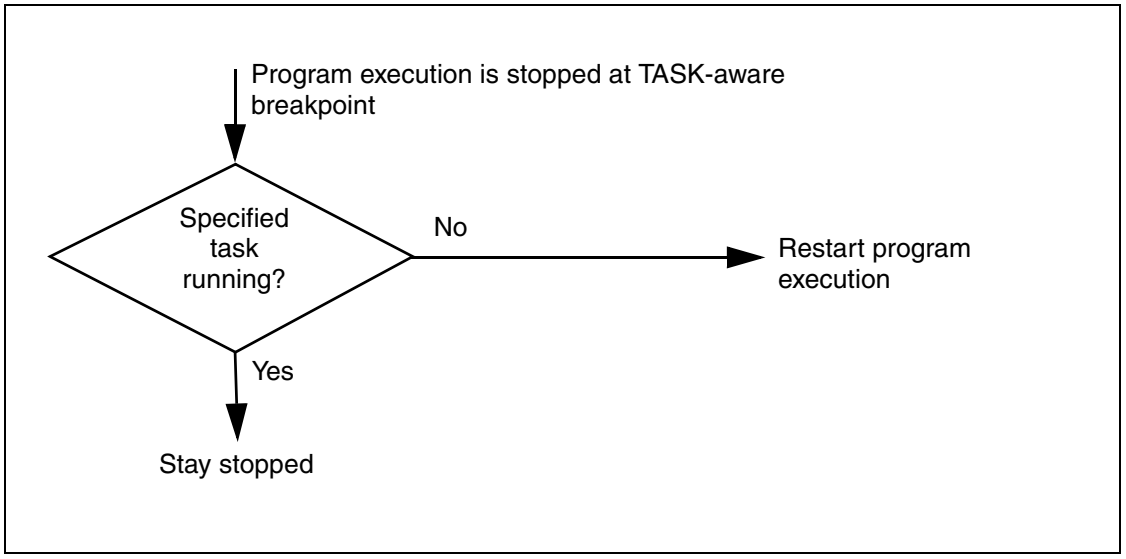
If the **advanced** button is pushed additional input fields are appended to the **Break.Set** dialog box to provide advanced breakpoint features

Advanced breakpoint input fields

TASK-aware breakpoints allow to stop the program execution at a breakpoint if the specified task/process is running.

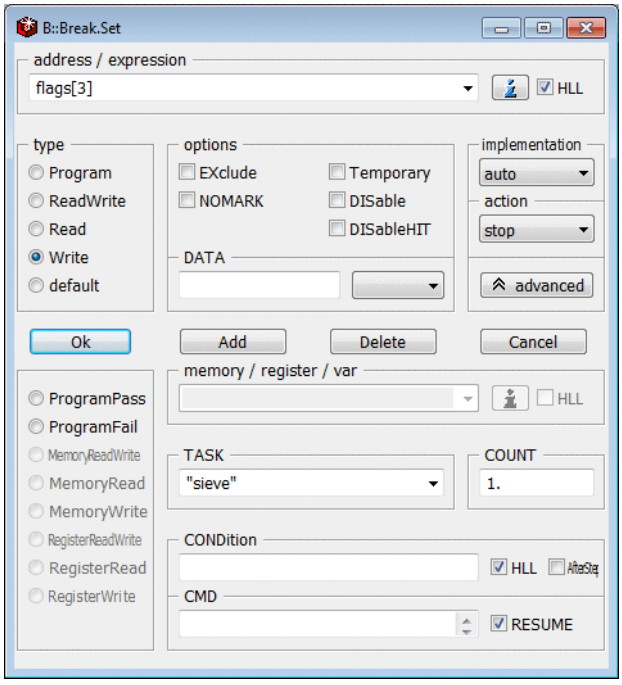
TASK-aware breakpoints are implemented as intrusive breakpoints.

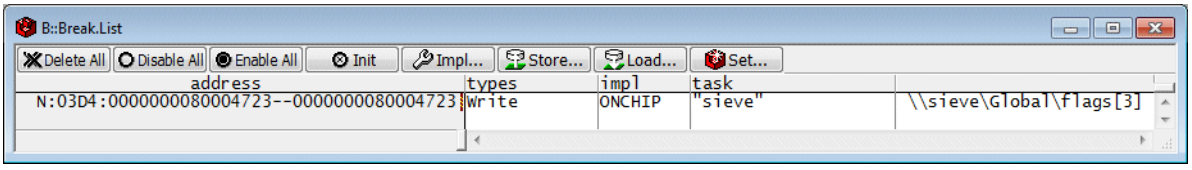
Processing:

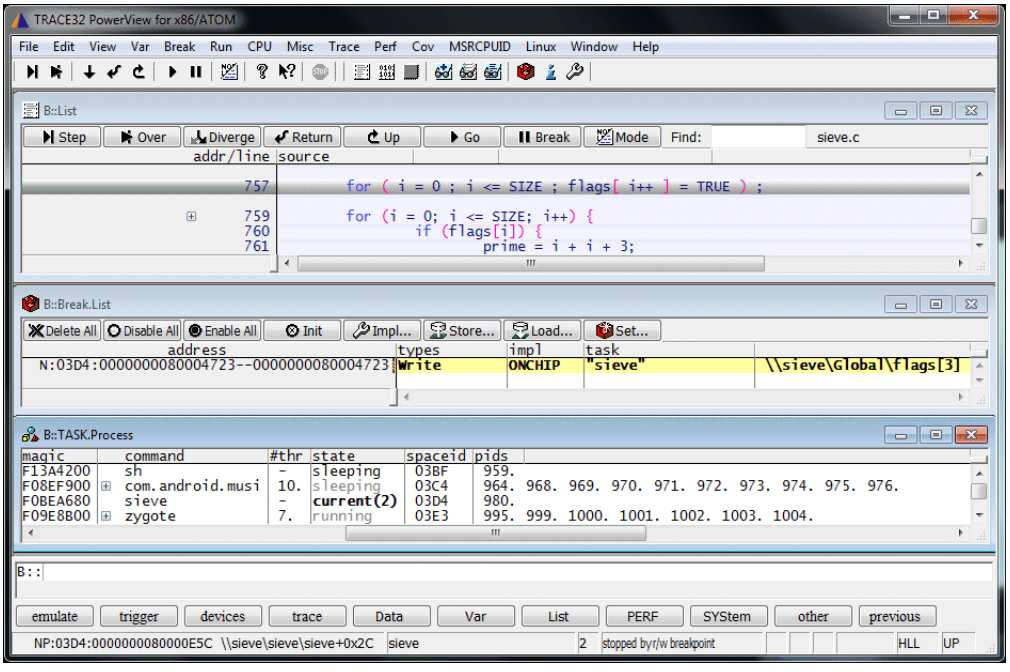


Each stop at the TASK-aware breakpoint takes at least 1. ms. This is why the red S is displayed in the Debugger Activity field of the TRACE32 PowerView state line whenever the breakpoint is hit.

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







Break.Set *<address>* | *<range>* **/[Program | Write | ReadWrite] /TASK** *<task_name>* | *<task_id>* | *<task_magic>*

Var.Break.Set *<hll_expression>* **/[Write | ReadWrite] /TASK** *<task_name>* | *<task_id>* | *<task_magic>*

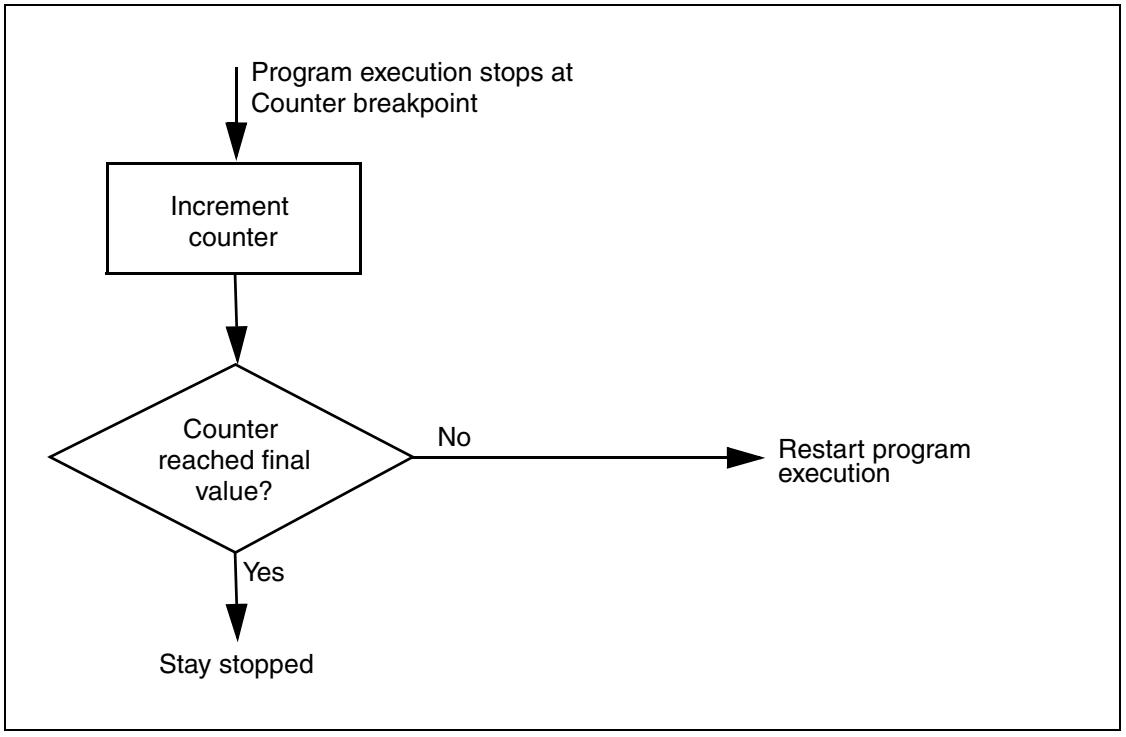
```
; use task ID to specify task
Break.Set 0x602be3 /Program /TASK 223.

; use task name to specify task
Var.Break.Set flags[3] /Write /TASK "sieve"
```

Counter

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

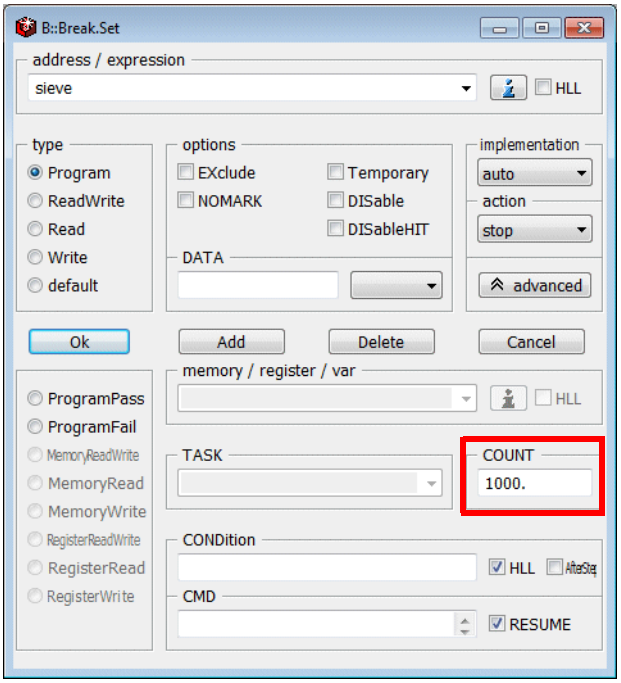
The onchip breakpoint logic of Intel® x86/x64 does not provide counters, so counters are implemented as software counters.

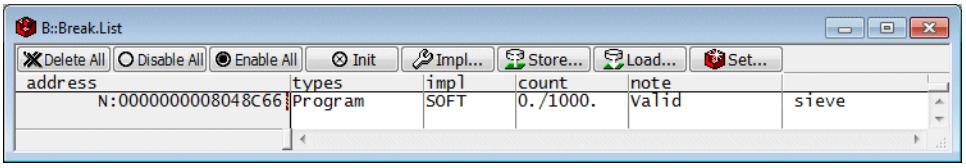
Processing:

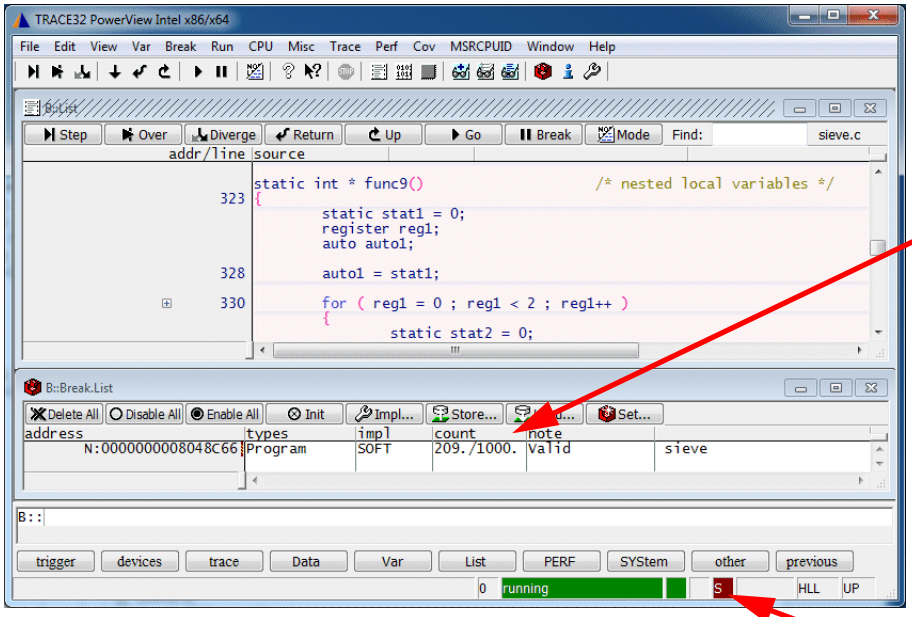


Each stop at a Counter breakpoint takes at least 1.ms. This is why the red S is displayed in the Debugger Activity field of the TRACE32 PowerView state line whenever the breakpoint is hit.

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

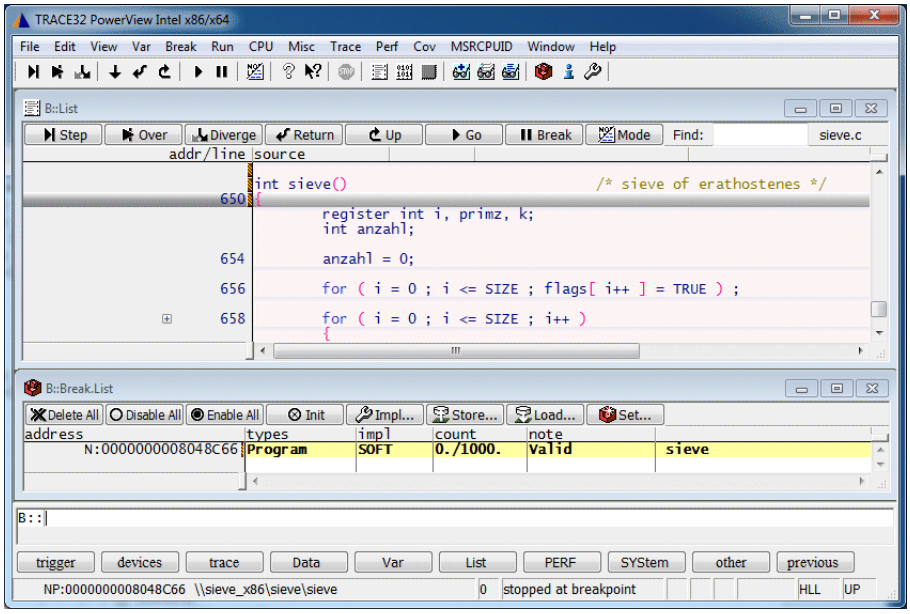






The current counter contents is permanently updated

The red S indicates an intrusive breakpoint



Break.Set *<address>* | *<range>* **/[Program | Write | ReadWrite] /COUNT** *<number>*

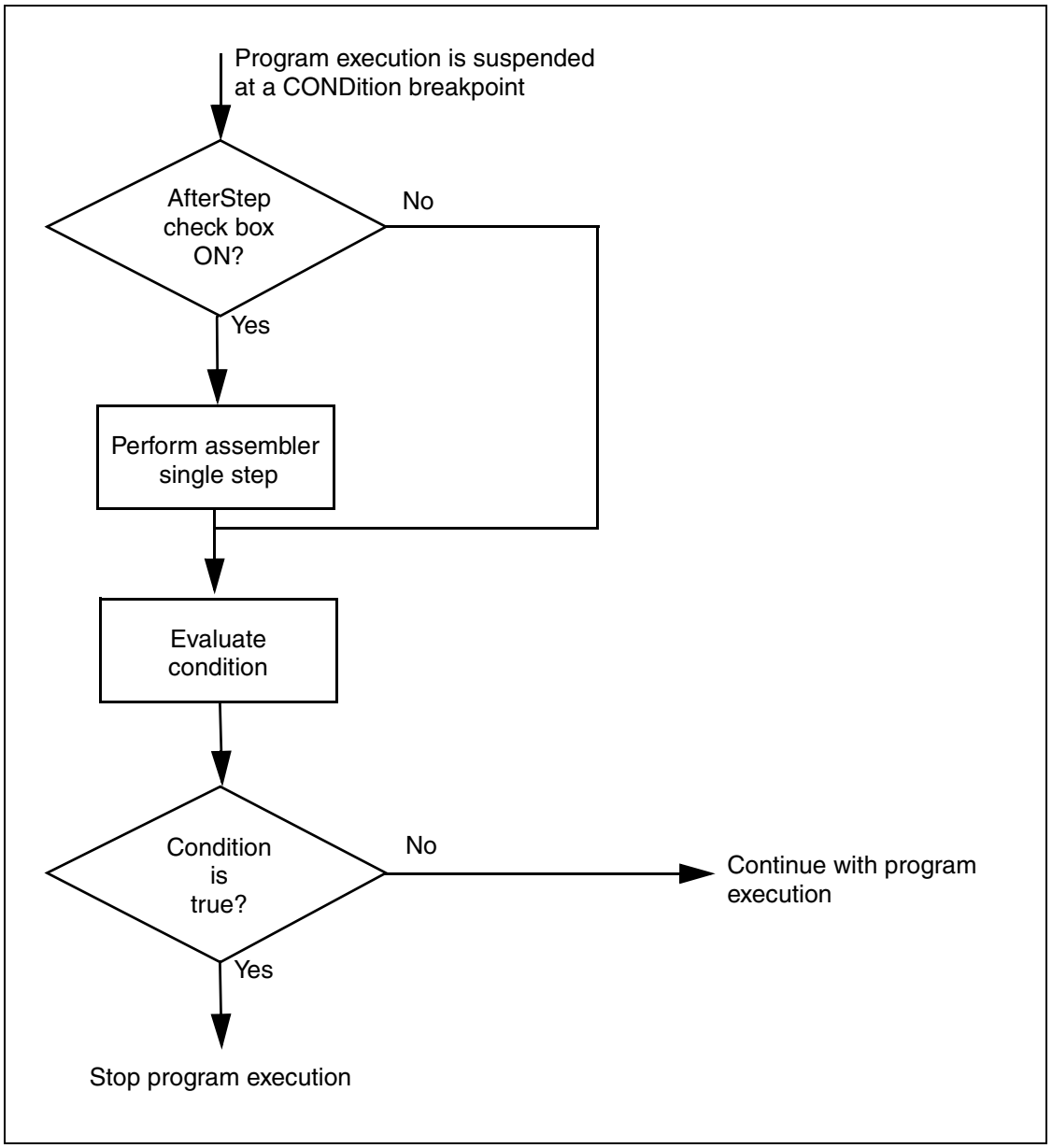
Var.Break.Set *<hll_expression>* **/[Write | ReadWrite] /COUNT** *<number>*

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

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

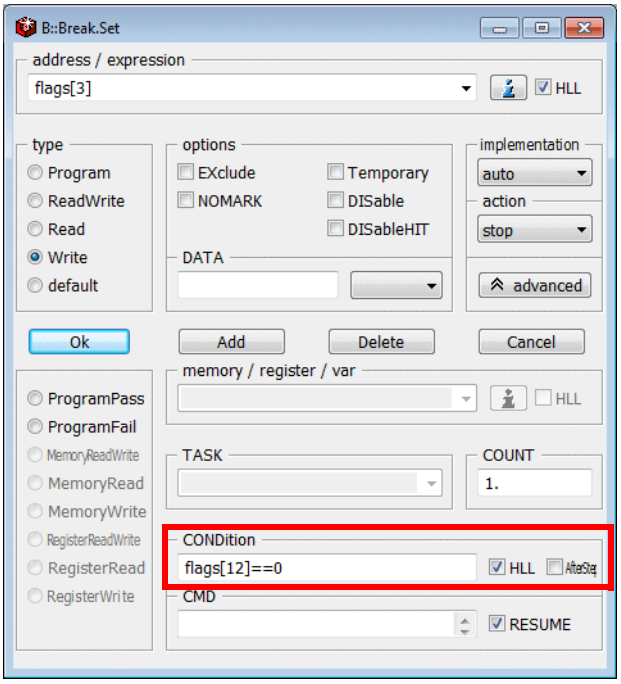
CONDition breakpoints are always intrusive.

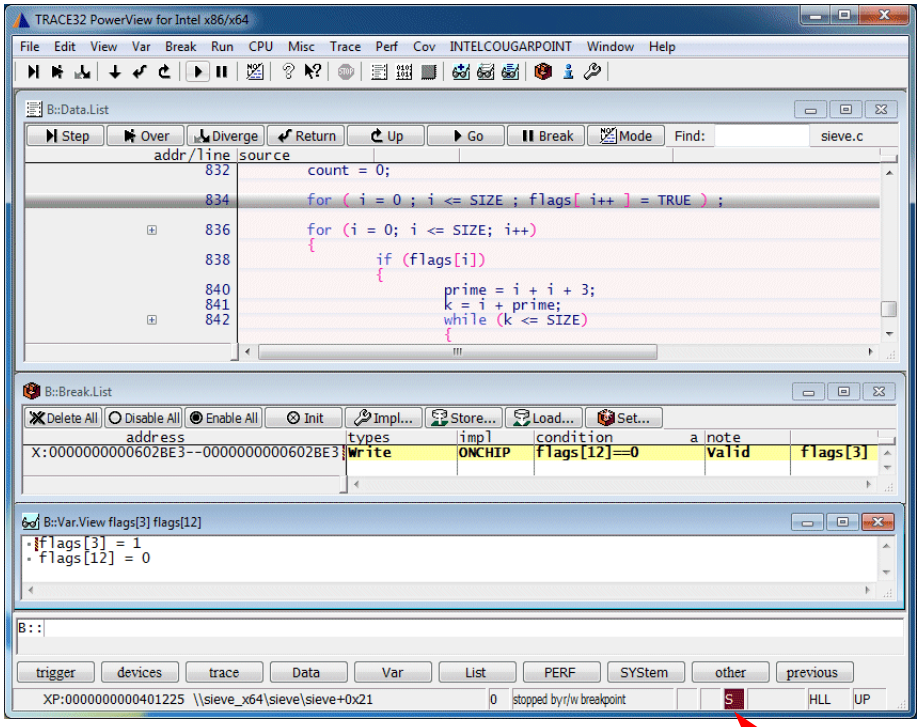
Processing:



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

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





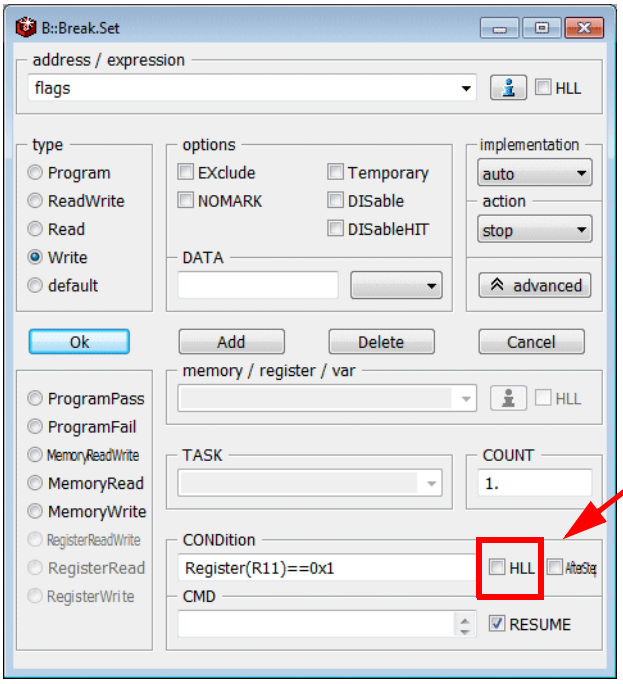
The red S indicates an intrusive breakpoint

Var.Break.Set *<hll_expression>* **/[Program | Write | ReadWrite] /VarCONDition** *<hll_condition>*

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

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

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

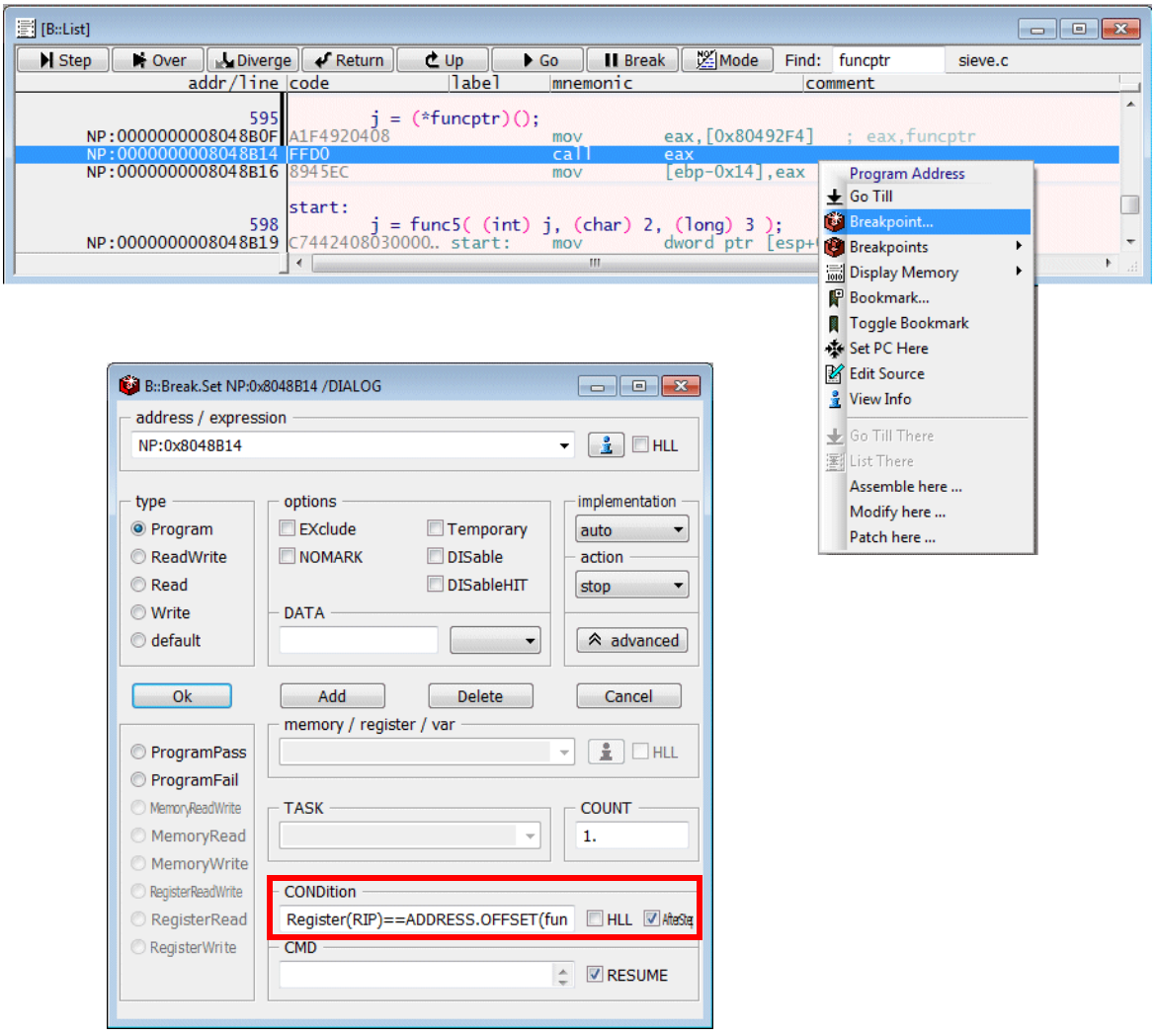


Switch HLL OFF ->
TRACE32 syntax can be used
to set the breakpoint

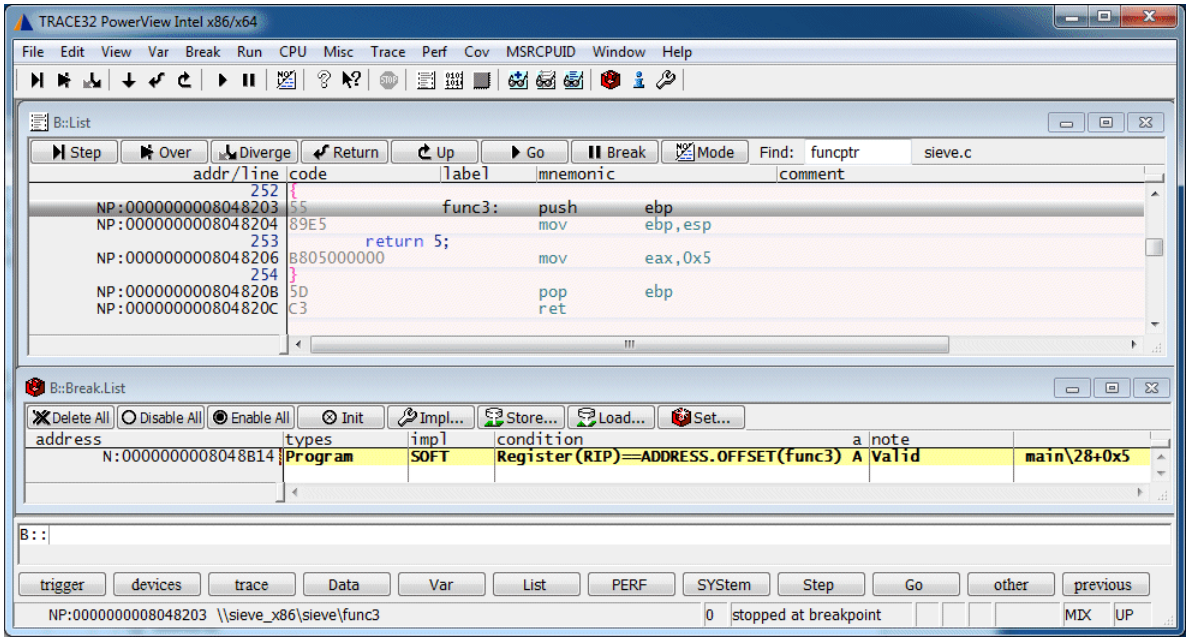
Break.Set *<address> | <range>* **/[Program | Write | ReadWrite] /CONDition** *<condition>*

```
; 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 ND:0x1000 is larger then 0x12345
Break.Set flags /Write /CONDition Data.Long(ND:0x1000)>0x12345
```

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



A indicates that TRACE32 performs an assembler step before it evaluates the condition when the program execution is suspended at the CONDition breakpoint.



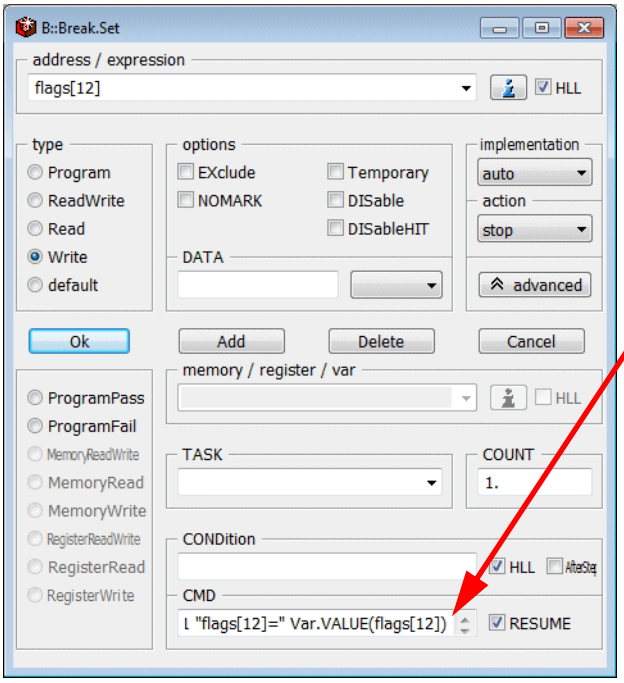
Break.Set *<address> | <range>* **/[Program | Write | ReadWrite] /CONDition** *<cond.>* **/AfterStep**

```
Break.Set main\44+0x7 /Program
/CONDition Register(RIP)==ADDRESS.OFFSET(func3) /AfterStep
```

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

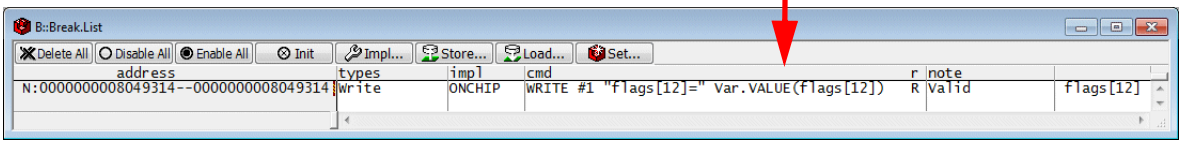
Example: Write the contents of flags[12] to a file whenever the write breakpoint at flags[12] is hit.

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



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

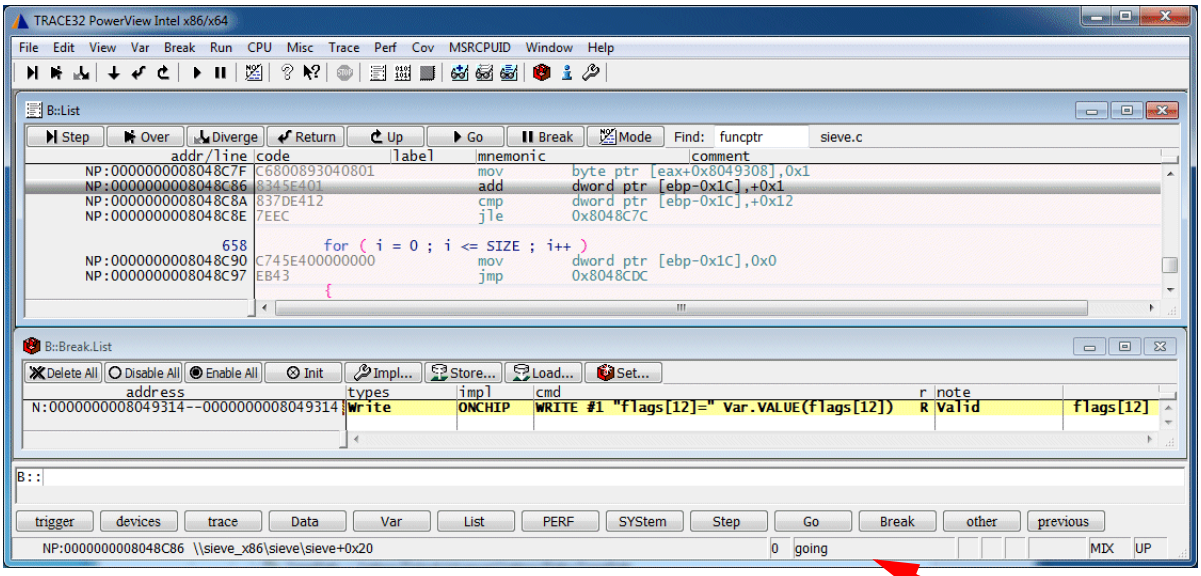
The **cmd** field in the Break.List window informs the user which command(s) is associated with the breakpoint. **R** indicates that RESUME is ON.



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** or **Trace.Chart.sYmbol**

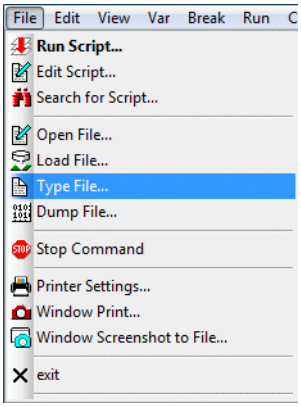
because the program execution is restarted before these commands are finished.

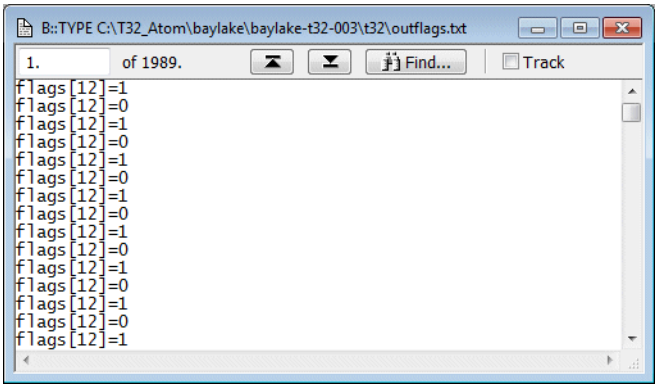


The state of the debugger toggles between **going** and **stopped**

```
CLOSE #1                                ; close the file when you are done
```

Display the result:



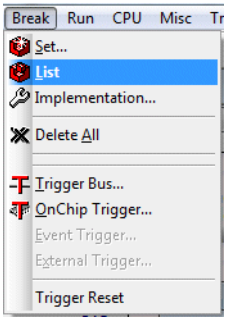


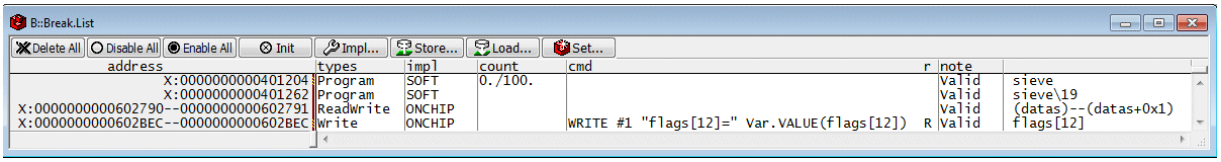
Break.Set *<address>* | *<range>* **/[Program | Write | ReadWrite] /CMD {***<command>***} [/RESUME]**

Var.Break.Set *<hll_expression>* **/[Write | ReadWrite] /CMD {***<command>***} [/RESUME]**

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

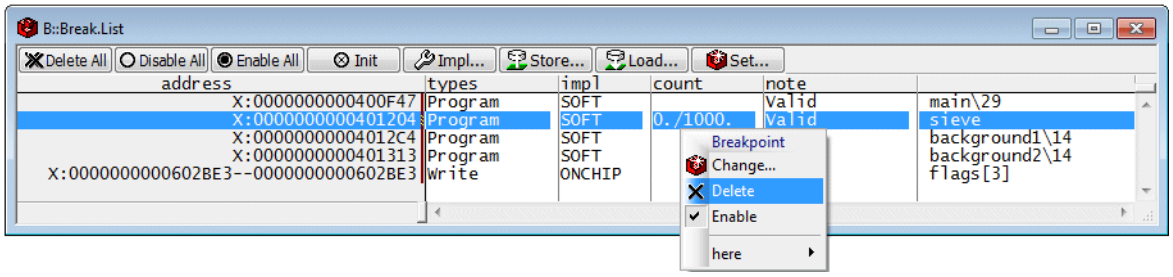
Display a List of all Set Breakpoints





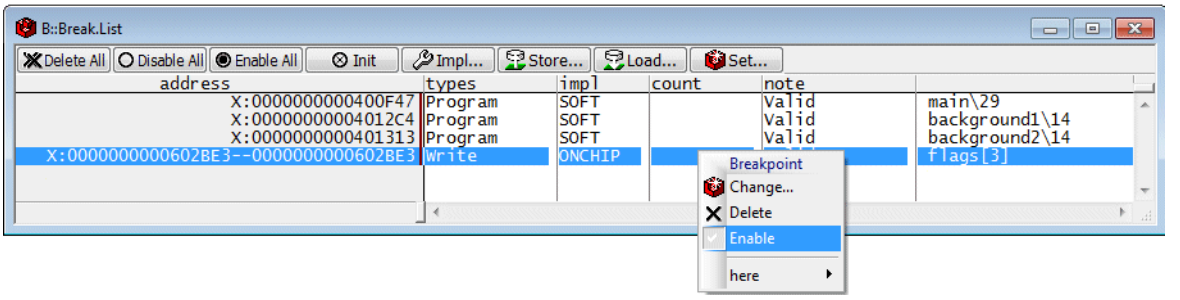
Break.List	List all breakpoints

Delete Breakpoints



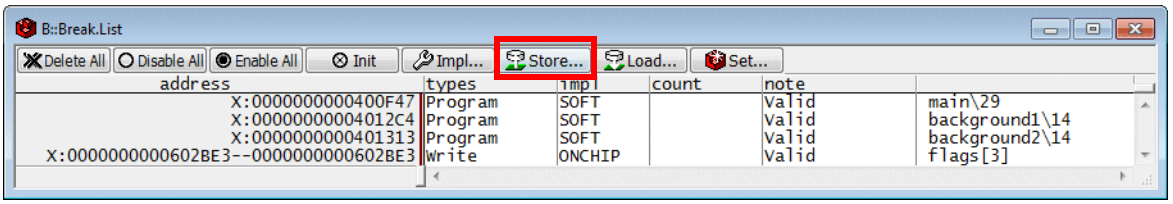
Break.Delete *<address>|<address_range>* [*/<type>*] [*/<implem.>*] [*/<option>*]	Delete breakpoint
Var.Break.Delete *<hll_expression>* [*/<type>*] [*/<implem.>*] [*/<option>*]	Delete HLL breakpoint

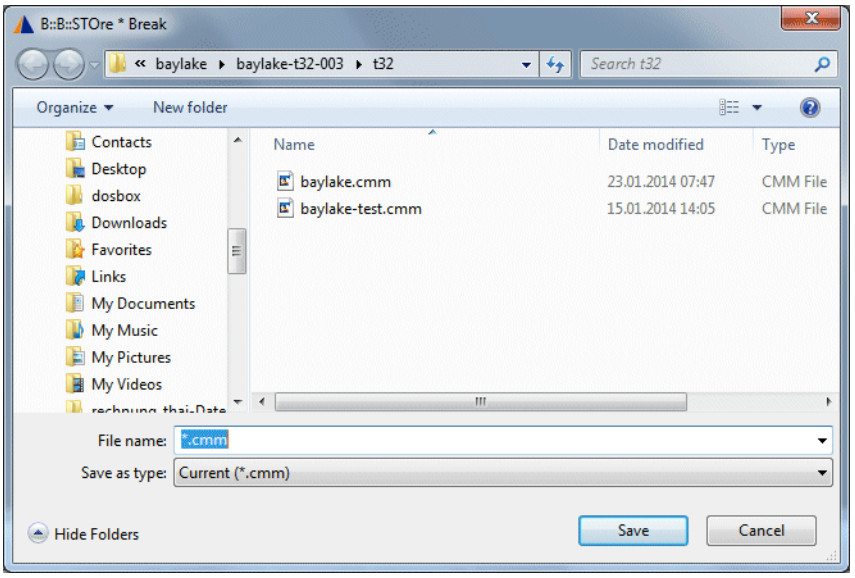
Enable/Disable Breakpoints



Break.ENable [*<address>*|*<address_range>*] [**/***<option>*]	Enable breakpoint
Break.DISable [*<address>*|*<address_range>*] [**/***<option>*]	Disable breakpoint

Store Breakpoint Settings



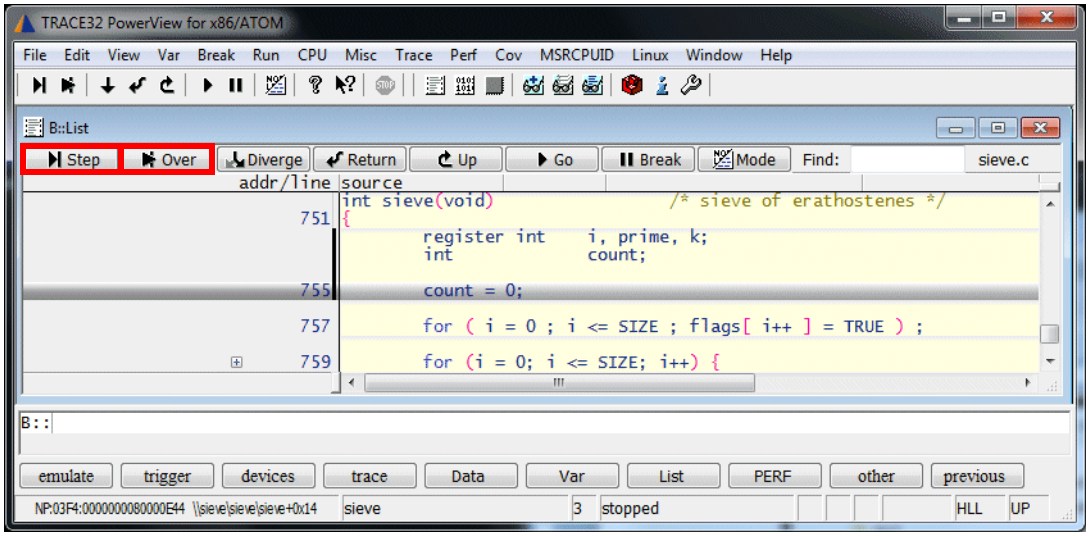



```
// T32_1000143 Thu Jan 23 12:02:56 2014

 B::

 Break.RESet
 Break.Set   main\48 /Program /DISable
 Var.Break.Set plot1; /Read /Write /TASK "sieve"
 Var.Break.Set datas.b[2]; /Write

 ENDDO
```

STOre *<filename>* **Break** Generate a script for breakpoint settings

Basic Debug Control

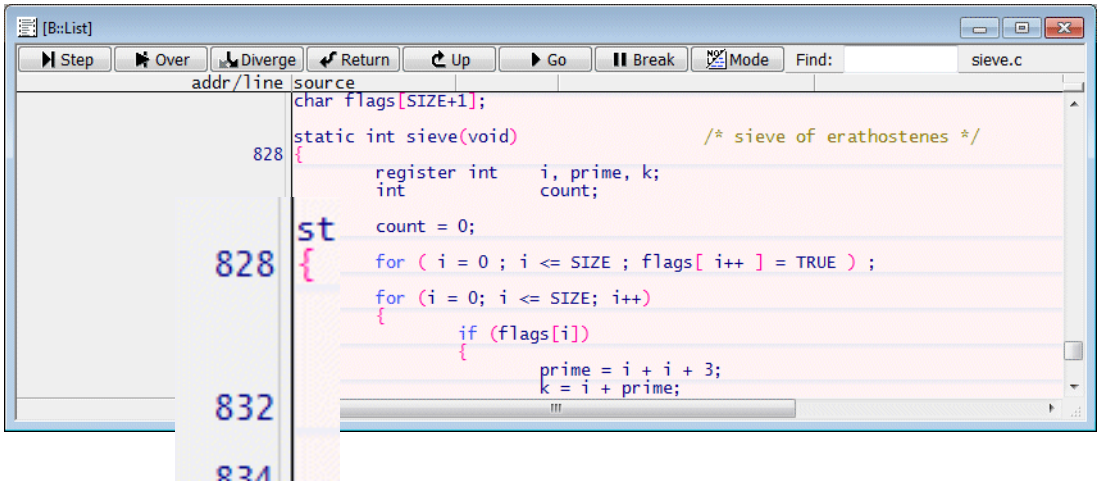
There are local buttons in the **List** window for all basic debug commands:



Step	Single stepping (command: **Step**) Please remember that assembler single steps are only performed on the selected core.
Over	Step over the call (command **Step.Over**)
Diverge	Exit loops or fast forward to not yet stepped code lines. **Step.Over** is performed repeatedly.

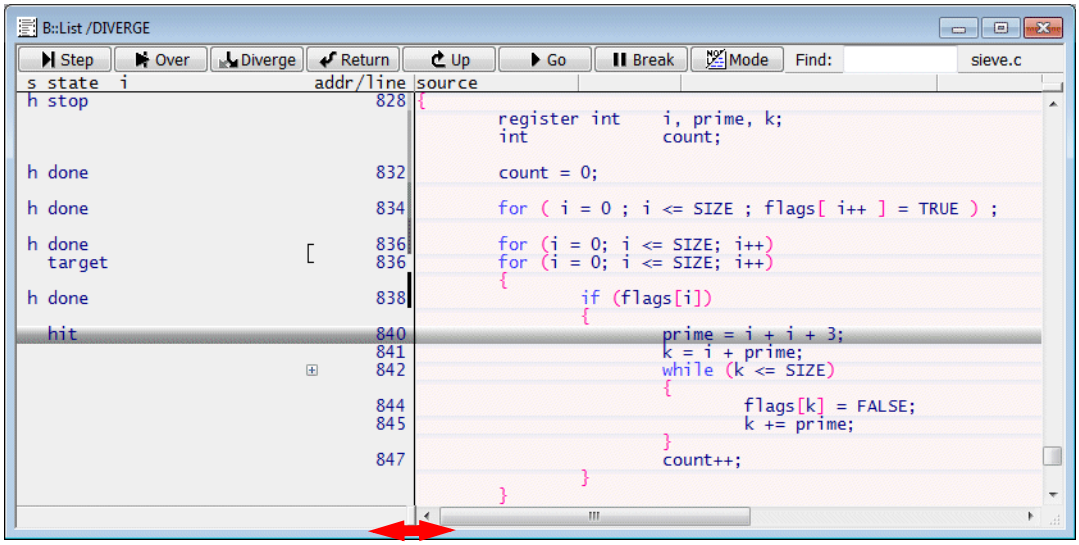
More details on Step.Diverge

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.

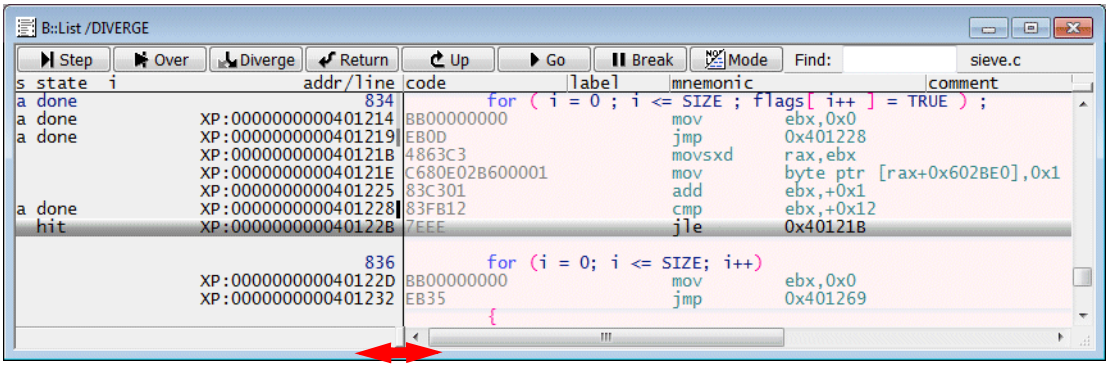


The following command allows you to get more details:

```
List.auto /DIVERGE
```



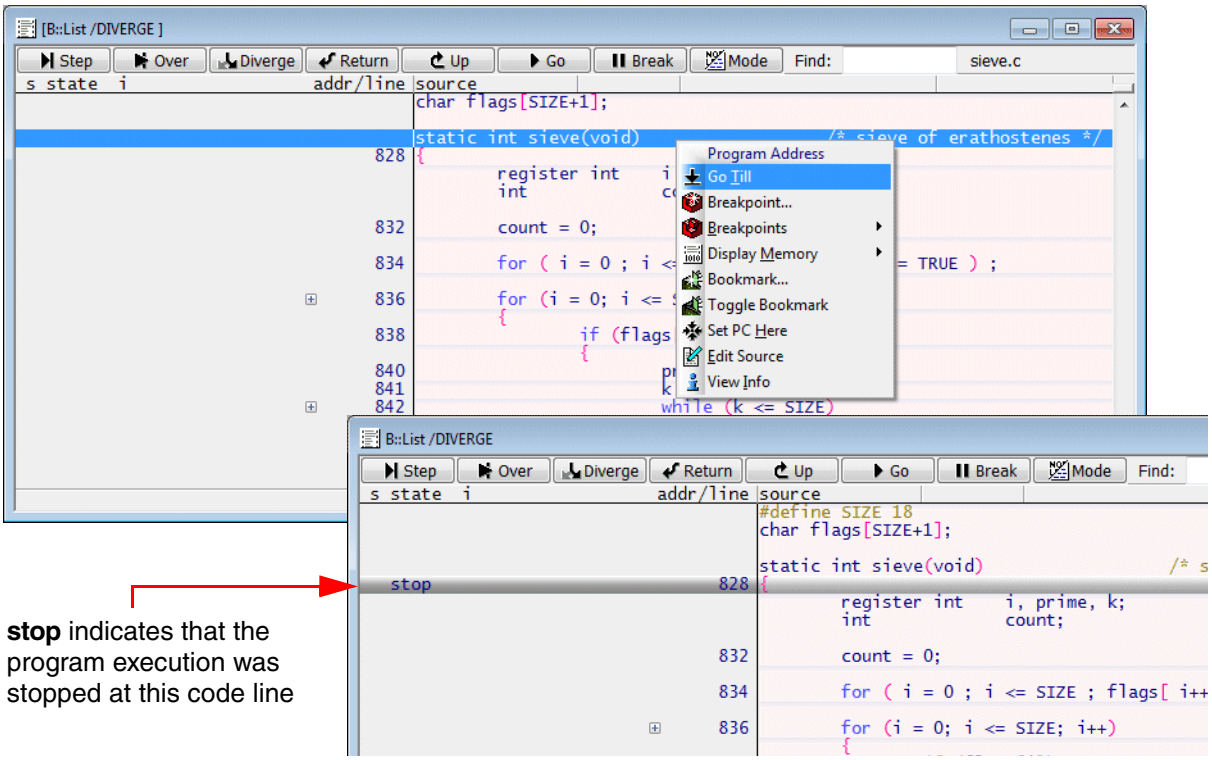
Drag this handle to see the DIVERGE details



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).

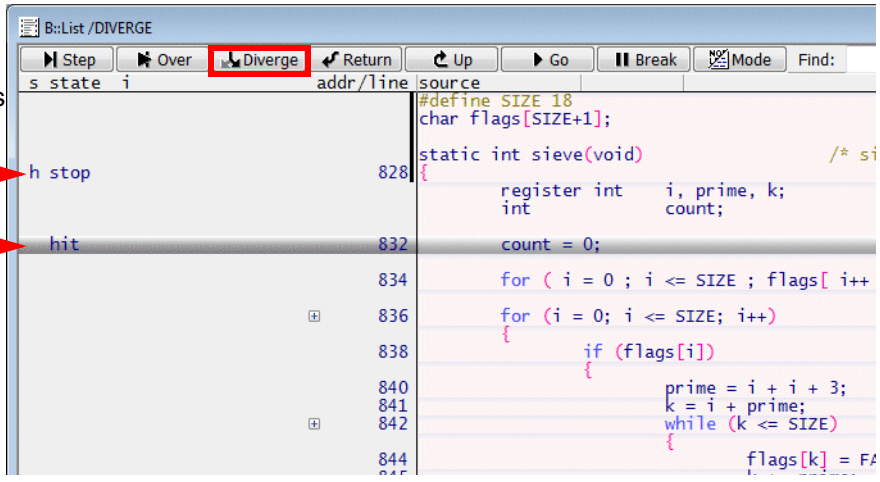
Example 1: Diverge through function sieve.

1. Run program execution until entry to function sieve.



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

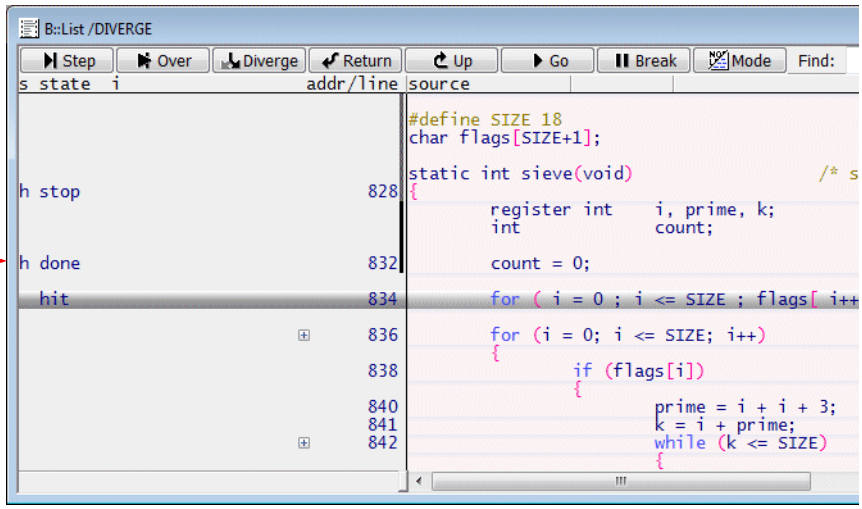
2. Start a Step.Diverge command.



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

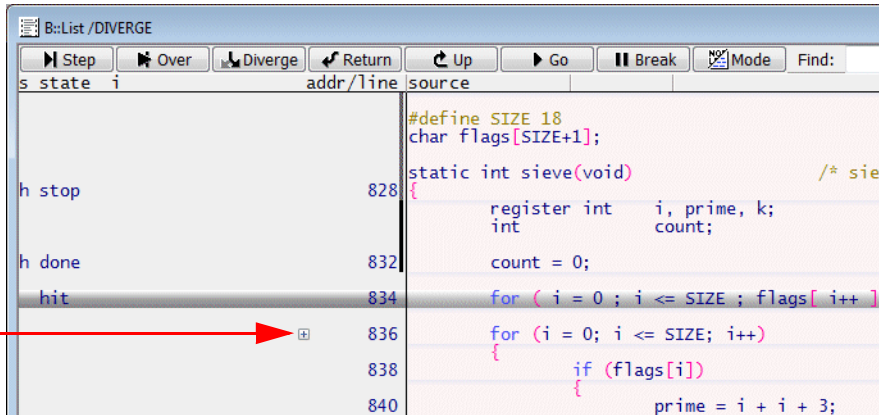
hit indicates that this code line was reached by Step command

3. Continue with Step.Diverge.

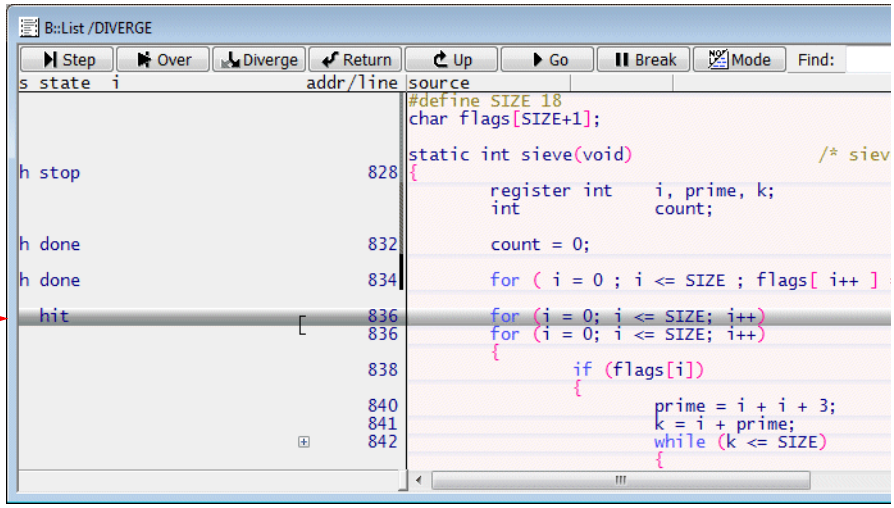


done indicates that the code line was reached by a Step command and that a Step command was started from this code line

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

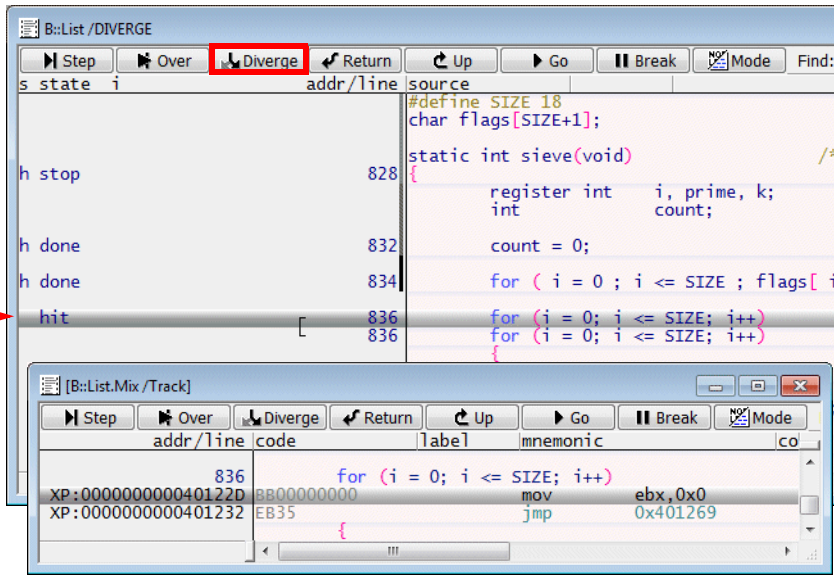


4. Continue with Step.Diverge.

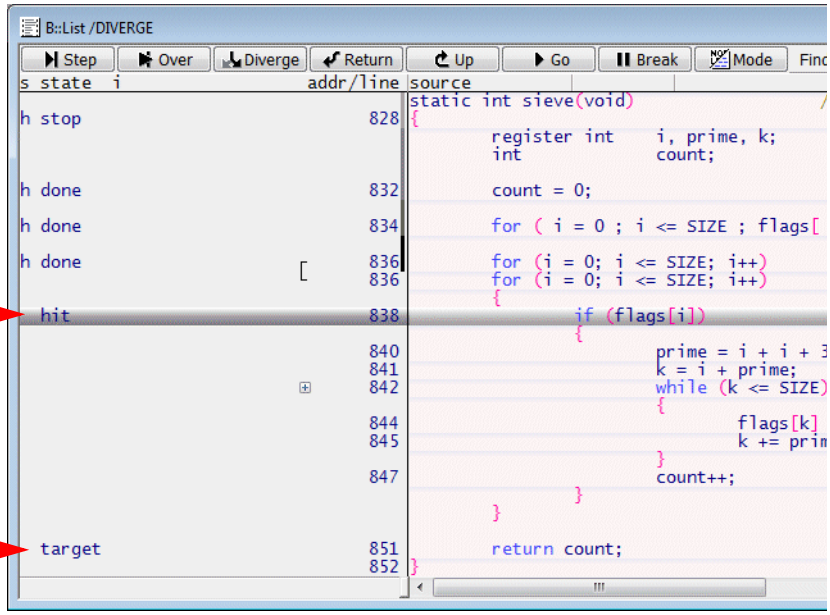


The drill-down tree is expanded and the HLL code line representing the reached block of assembler code is marked as **hit**

This assembler code generated for the HLL line includes a conditional branch



5. Continue with Step.Diverge.

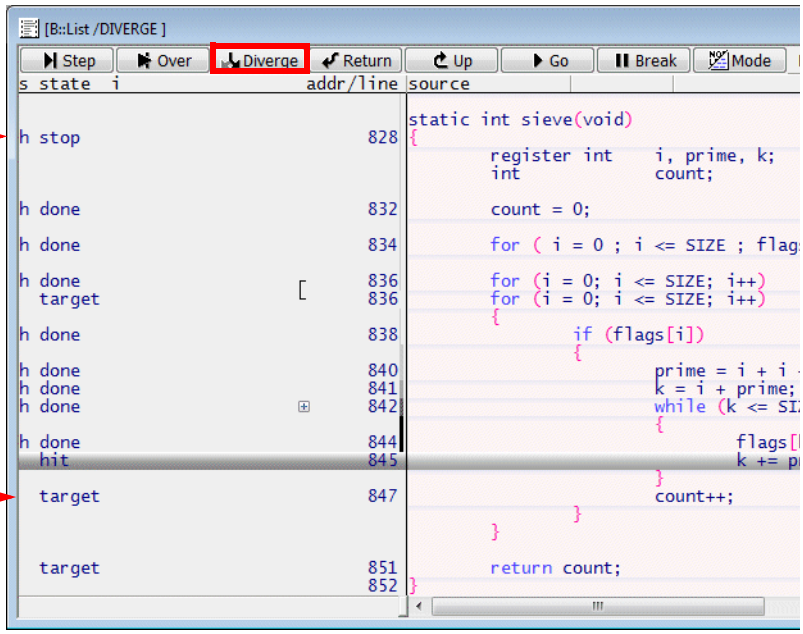


The reached code line is marked as **hit**

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

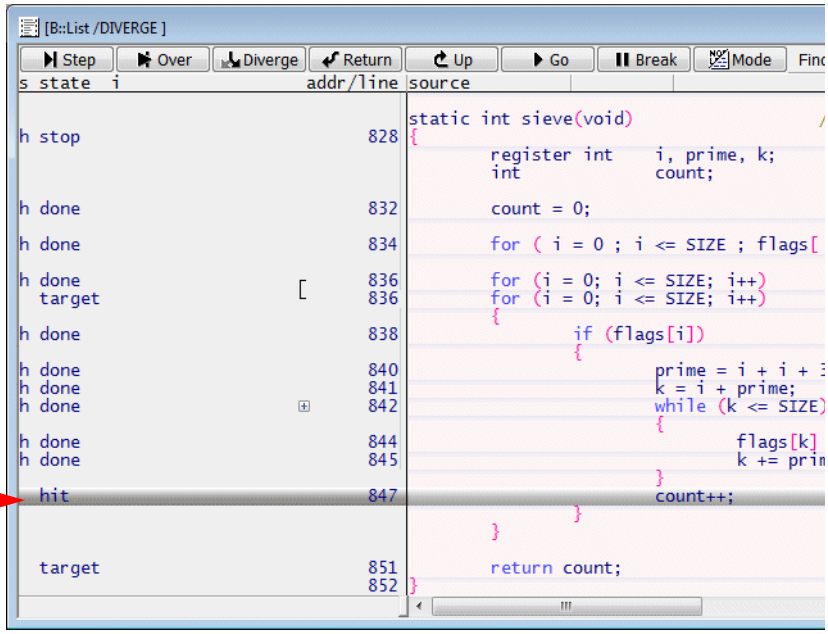
6. Continue with Step.Diverge (several times).

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



7. Continue with Step.Diverge.

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



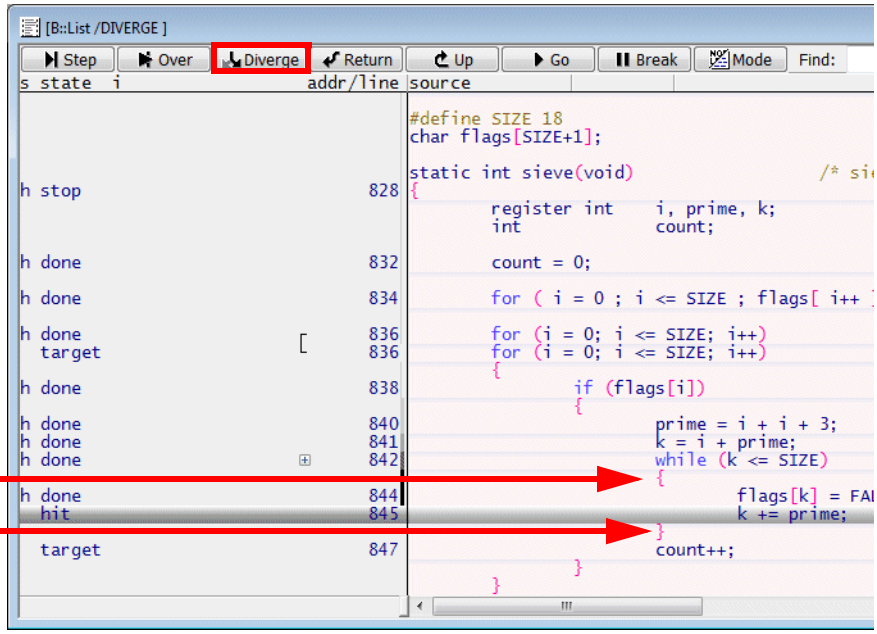
When all reachable code lines are marked as **done**, the following message is displayed:

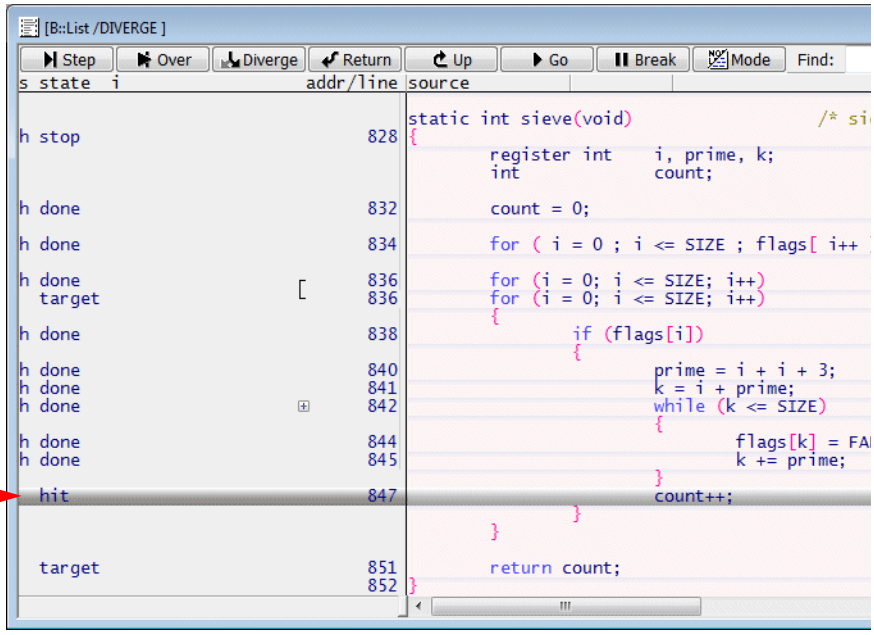
The **DIVERGE marking** is cleared when you use the **Go.direct** command without address or the **Break** command while the program execution is stopped.

Example 2: Exit a loop.

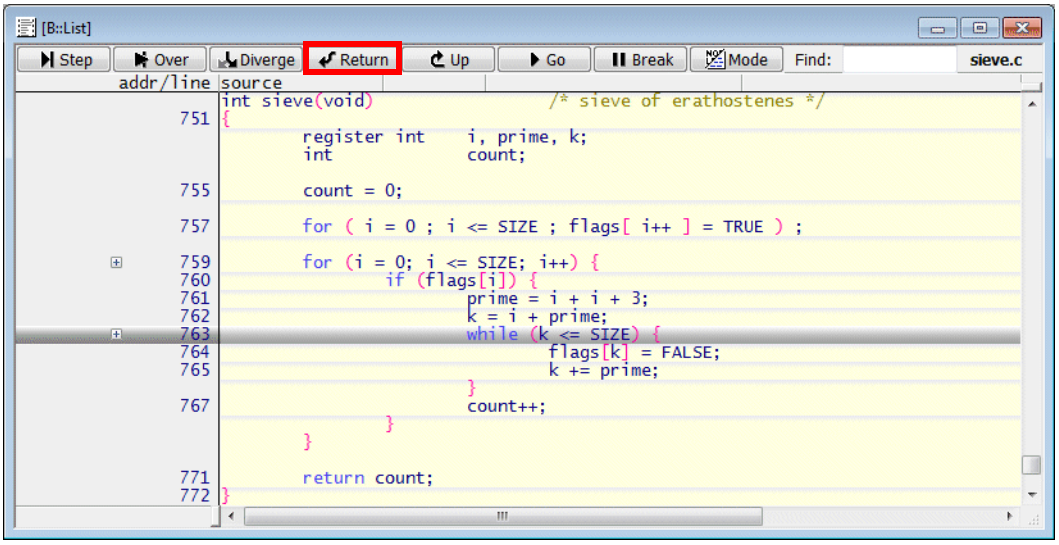
DIVERGE marking is done whenever you single step.

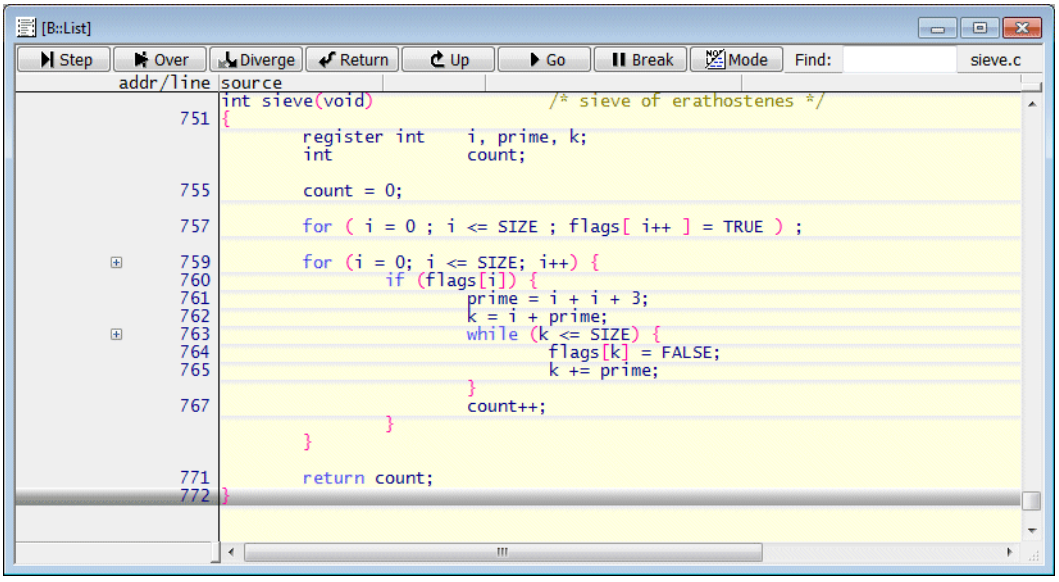
If all code lines of a loop are marked as **done/hit**, a Step.Diverge will exit the loop



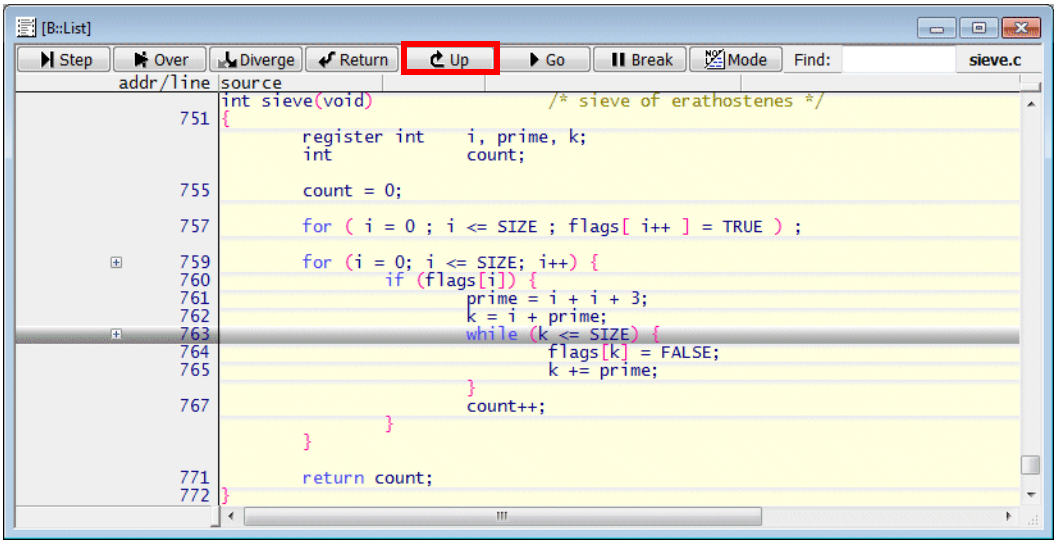


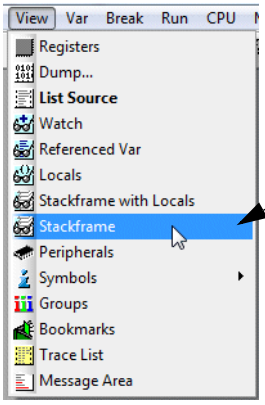
Return	**Return** sets a temporary breakpoint to the function exit and starts the program execution (command: **Go.Return**)



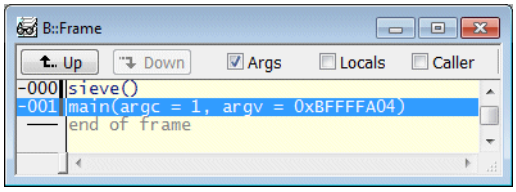


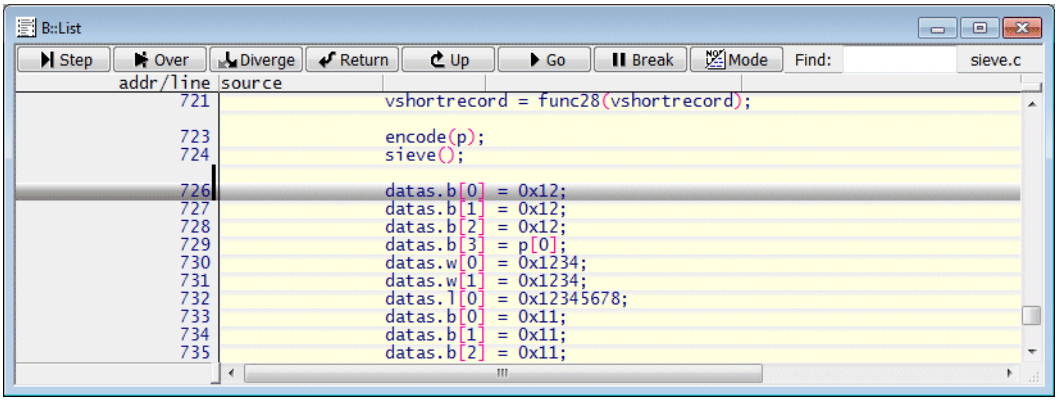
Up	**Up** is used to return to the function that called the current function. For this a temporary breakpoint is set to the instruction directly after the function call. Then the program execution is started. (command: **Go.Up**)





Display the HLL stack to see the function nesting





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

Step *<count>*	Single step is performed *<count>* times
Step.Change *<expression>*	Step until *<expression>* changes
Step.Till *<condition>*	Step until *<condition>* becomes true
Var.Step.Change *<hll_expression>*	Step until *<hll_expression>* changes
Var.Step.Till *<hll_condition>*	Step until *<hll_condition>* becomes true

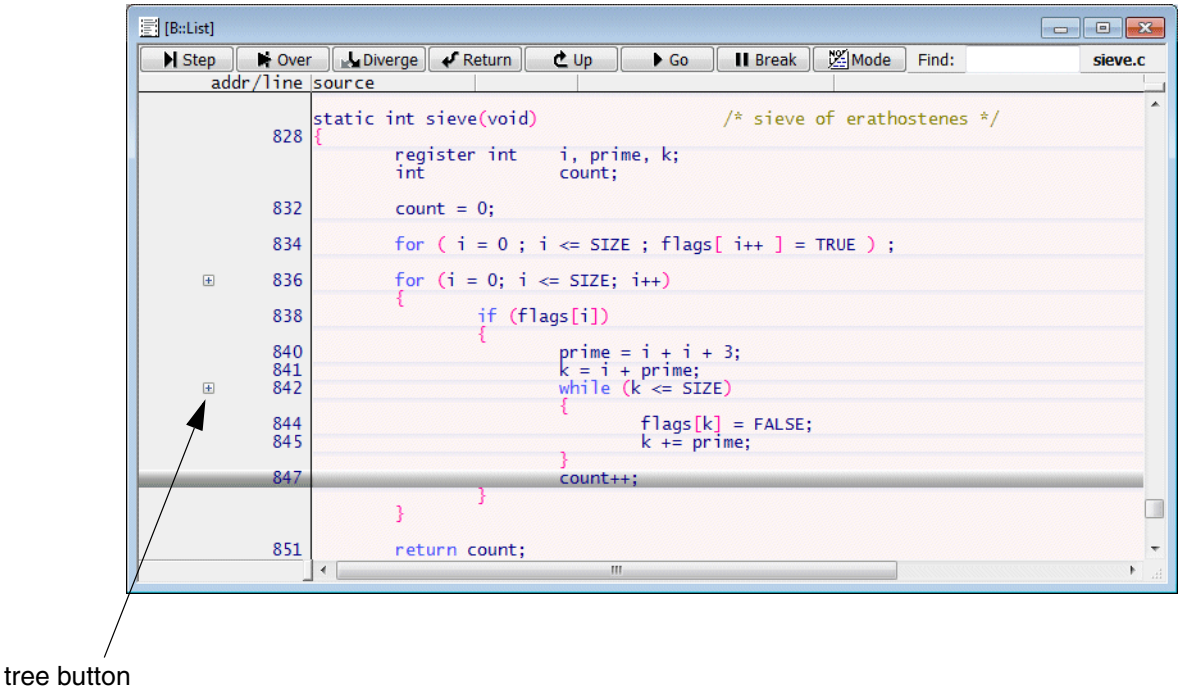
```
; step 1000. times
Step 1000.

; step until the contents of register R9 changes
Step.Change Register(R9)

; step until byte at address ND:80004723 is 1
Step.Till Data.Byte(ND:80004723)==1

; step until the contents of the variable mstatic1 changes
Var.Step.Change mstatic1

; step until the variable mstatic2 is larger the 3 and
; the variable flags[3] is unequal 1
Var.Step.Till (mstatic2>3)&&(flags[3]!=1)
```

Debugging of Optimized Code

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 complicates debugging.

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

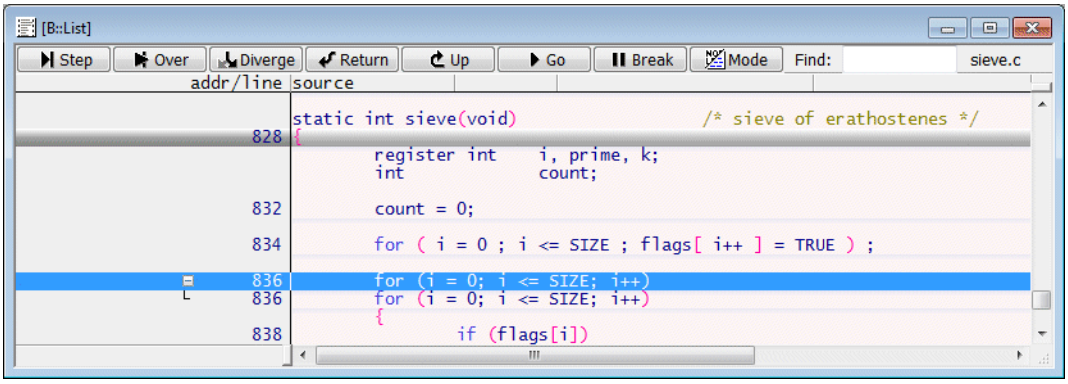


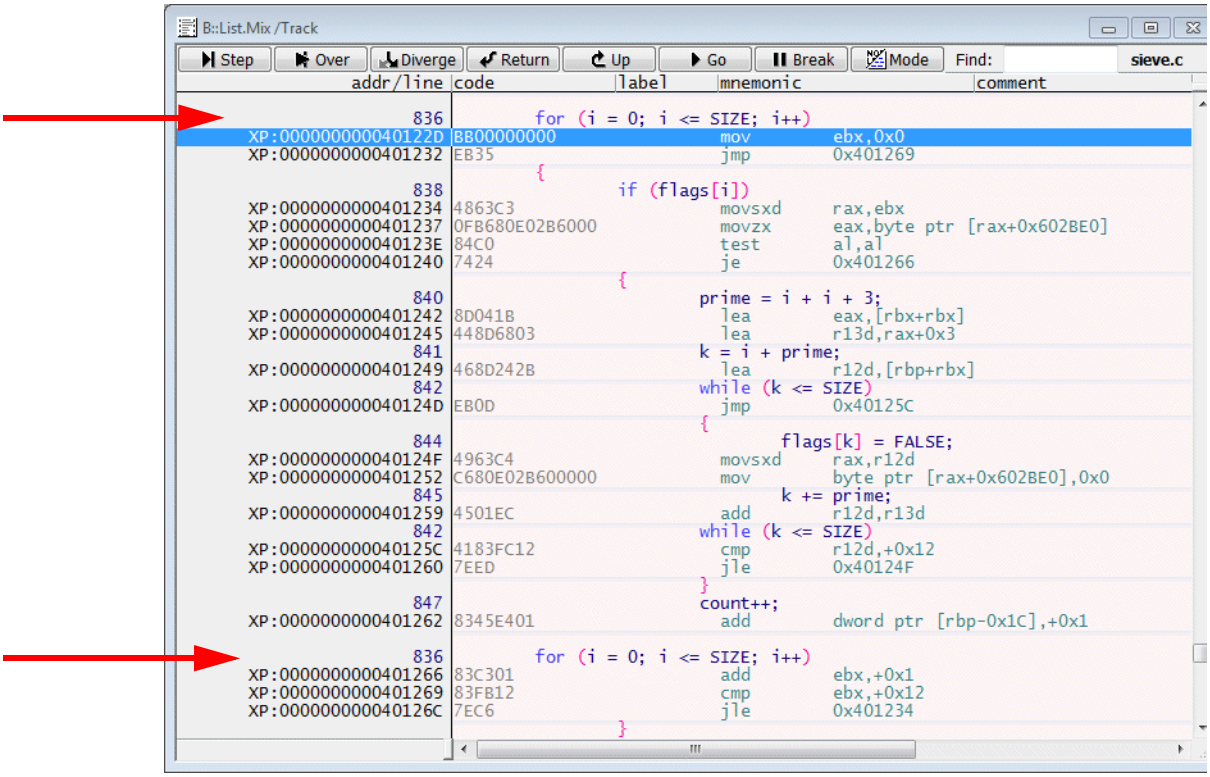
The following background information is fundamental if you want to debug optimized code:

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

The expansion of the tree 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).



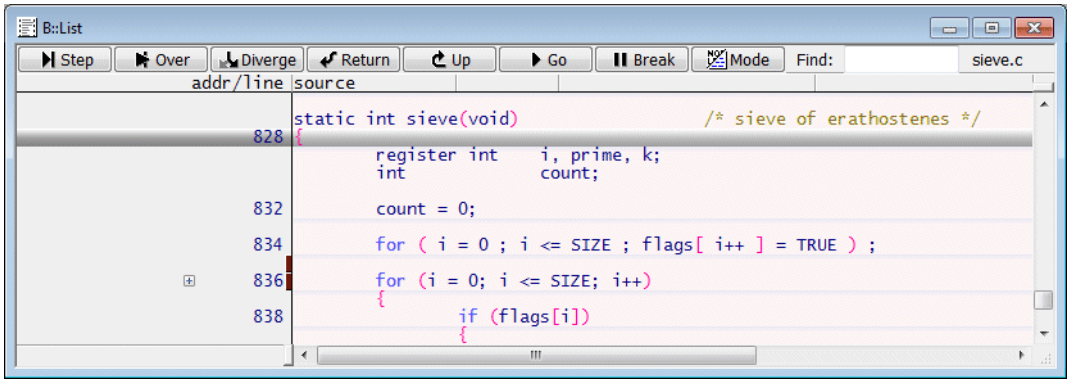


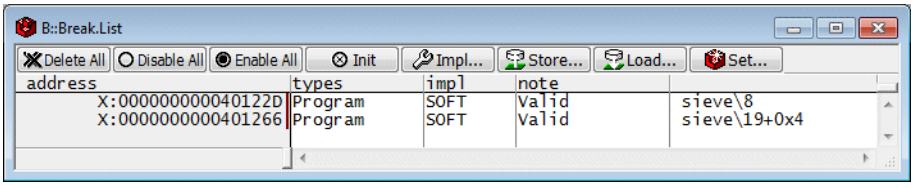
To keep track when debugging optimized code, it is recommended to work with an HLL mode and a MIX mode display of the source listing in parallel.

```
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.



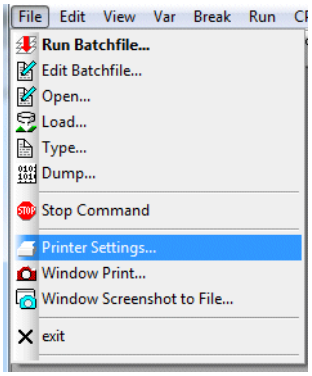


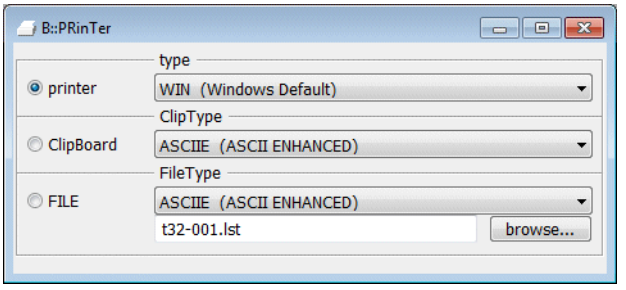
Settings

TRACE32 PowerView supports the following ways to document your results:

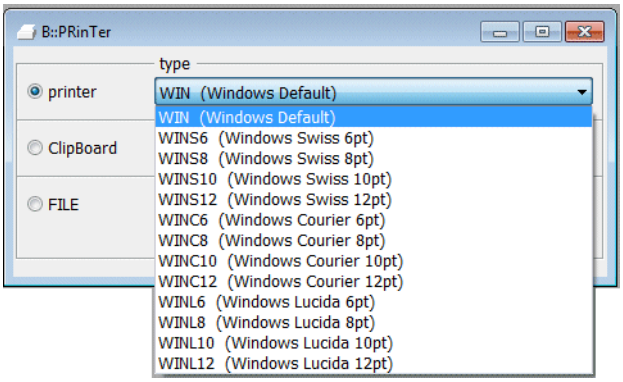
- Sending them to a printer
- Transferring them via the clipboard to other applications
- Saving them to a file

The output way and the output format is configured via the **Printer Settings …** dialog.



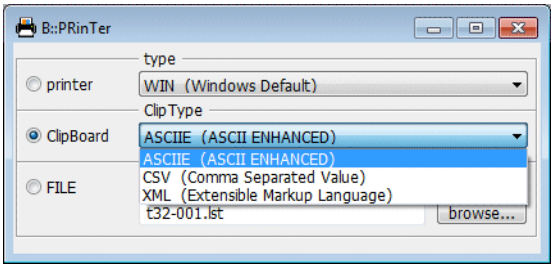


Print

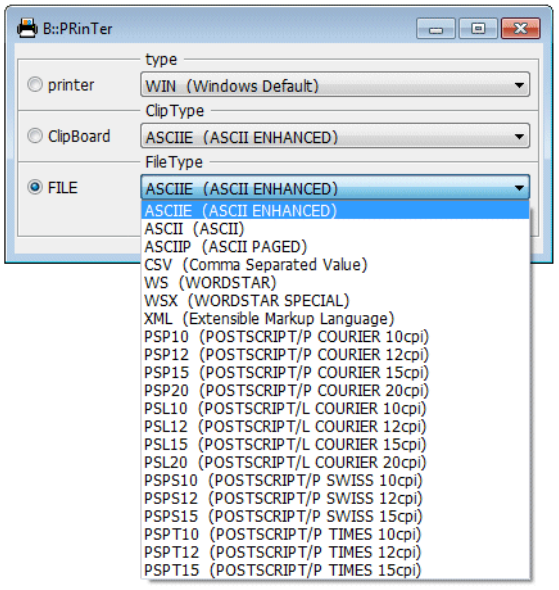


If you send your results to a printer, you can select Windows Default or other font sizes and families supported by your printer.

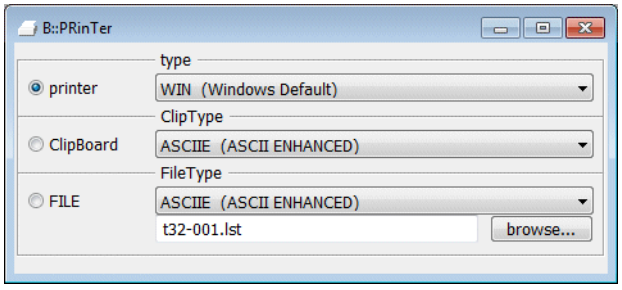
Clipboard



If you use the clipboard you can select ASCII ENANCED, Comma-Separated Values or XML.



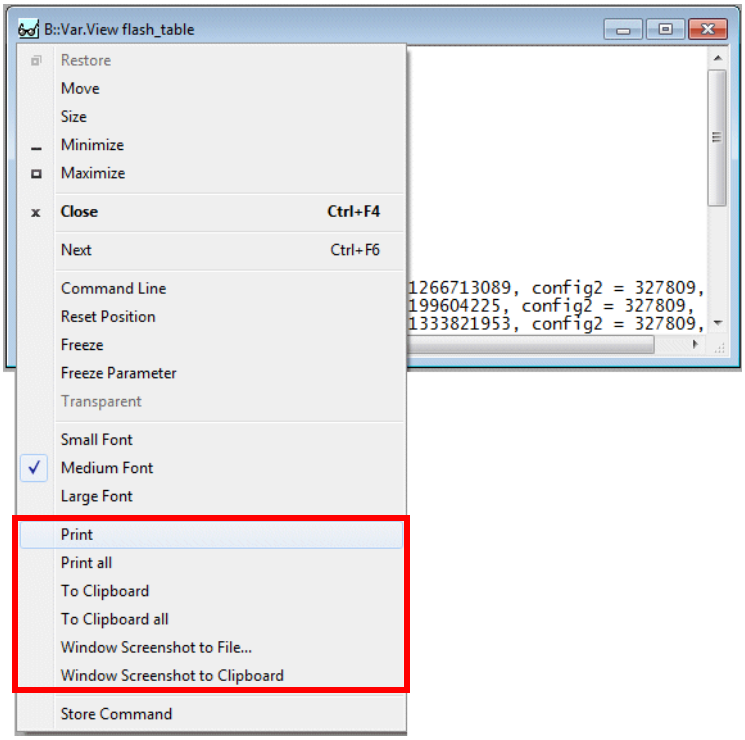
If you save your result to a file, you can choose between various ASCII formats, Comma-Separated Value, HML and various POSTSCRIPT formats.



If the file name contains numbers, the backmost number is incremented with each output.

If the file name does not contain a number, it is overwritten with each output.

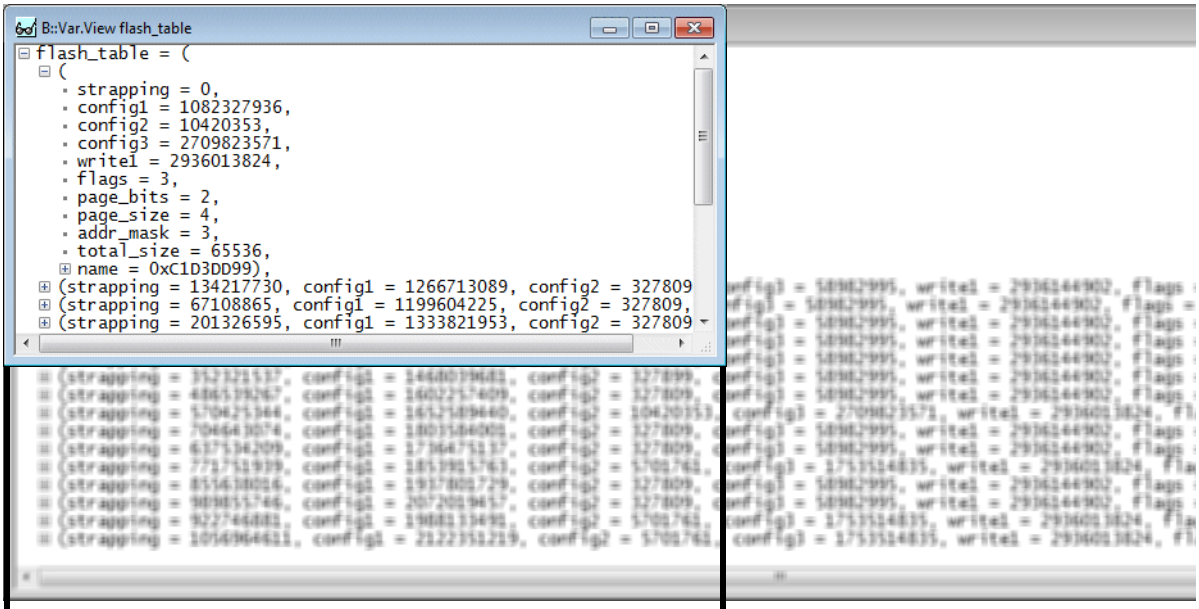
Quick Output



Print	Send window contents as visible to the selected output.
Print all	Send window contents - horizontal as visible - vertical all information to the selected output. (see explanation on next page)
To Clipboard	Send window contents as visible to clipboard.
To Clipboard all	Send window contents - horizontal as visible - vertical all information to clipboard.
Window Screenshot to File …	Make screenshot of window contents as visible and save it to file (various output formats supported).
Window Screenshot to Clipboard	Make screenshot of window contents as visible and send it to clipboard.

More details on **Print all**:

- **Print all** prints the visible horizontal information, but all vertical information (see picture below).



Visible horizontal information,
all vertical information

- To limit the output (especially when using a printer) it is recommended to limit the information displayed in the window.

  Examples:

```
Var.View %MultiLine flash_table[3..7]      ; display only
                                           ; flash_table[3]
                                           ; to flash_table[7]
                                           ; in window

Data.dump flags++0x1ff                     ; display hex dump from
                                           ; flags to flags+0x1ff
                                           ; in window
```

To send the complete output of a command to the selected output, use:

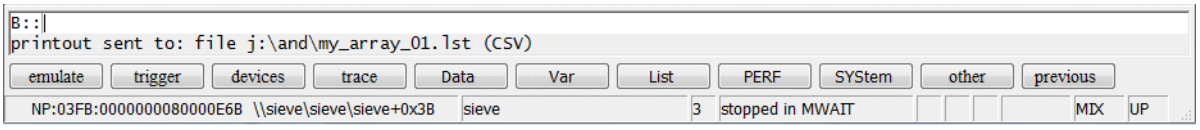
WinPrint.<command>

The horizontal size is currently 4K.

```
WinPrint.Var.View %MultiLine flash_table[3..7]

WinPrint.Data.dump flags++0x1ff
```

Complete script example:

```
PRinTer.FILE my_array_01

PRinTer.FileType CSV

WinPrint.Var.View %MultiLine flash_table[3..7]
```



If you want to collect several results in one file, use:

PRinTer.OPEN *[<filename>]*	Open permanent output file
PRinTer.CLOSE	Close permanent output file

```
PRinTer.OPEN my_array

PRinTer.FileType CSV

WinPrint.Var.View %MultiLine flash_table[3..7]

Go

Break

WinPrint.Var.View %MultiLine flash_table[3..7]

;…

PRinTer.CLOSE
```