

# Training Basic SMP Debugging for Intel® x86/x64

MANUAL

# Training Basic SMP Debugging for Intel® x86/x64

---

[TRACE32 Online Help](#)

[TRACE32 Directory](#)

[TRACE32 Index](#)

[TRACE32 Training](#) .....



[Training Intel® x86/x64](#) .....



[Training Basic SMP Debugging for Intel® x86/x64](#) .....

1

[Debug Configurations](#) .....

5

    CombiProbe 2 MIPI60-Cv2

6

        MIPI60-Cv2 Configuration

6

        MIPI60-Cv2 Features

7

    On-Chip Core Trace

9

    Off-Chip System/Core Trace

9

[Starting a TRACE32 PowerView Instance](#) .....

10

    Basic TRACE32 PowerView Parameters

10

        Configuration File

10

        Standard Parameters

11

        Examples for Configuration Files

12

        Additional Parameters

14

    Application Properties (Windows only)

15

    Configuration via T32Start (Windows only)

16

    About TRACE32

17

        Version Information (Debug Cable)

17

        Prepare Full Information for a Support Email

19

[Establish your Debug Session](#) .....

20

    Course of Action

20

        Run the Boot Loader until the Target Configuration is Done

22

        Establish the Debug Communication

22

        Load the Debug Symbols for the Application and/or the OS

26

        Configure the TRACE32 OS Awareness for Your OS

27

        Stop the Program Execution

27

    Start-Up Script

28

        Write a Start-Up Script

28

        Run a Start-up Script

29

        Automated Start-up Scripts

30

[TRACE32 PowerView](#) .....

31

    SMP Concept

31

    TRACE32 PowerView Components

35

Main Menu Bar and Accelerators	36
Main Tool Bar	38
Window Area	40
Command Line	43
Message Line	47
Softkeys	48
State Line	49
Further Documentation	50
<b>Basic Debugging (SMP) .....</b>	<b>51</b>
Go/Break	51
Single Stepping on Assembler Level	53
Single Stepping on High-Level Language Level	54
<b>Registers .....</b>	<b>56</b>
Core Registers	56
Display the Core Registers	56
Colored Display of Changed Registers	57
Modify the Contents of a Core Register	58
Further Register Sets	59
Special Function Register	60
Display the Special Function Registers	60
The PER Definition File	63
Modify a Special Function Register	64
<b>Memory Display and Modification .....</b>	<b>65</b>
The Data.dump Window	67
Basics	67
Modify the Memory Contents	72
Run-time Memory Access	73
Colored Display of Changed Memory Contents	77
The List Window	78
Displays the Source Listing Around the PC	78
Displays the Source Listing of a Selected Function	79
<b>Breakpoints .....</b>	<b>81</b>
Breakpoint Implementations	81
Software Breakpoints in RAM (Program)	81
Onchip Breakpoints in NOR Flash (Program)	82
Onchip Breakpoints (Read/Write)	84
Onchip Breakpoints for Intel® x86/x64	85
Breakpoint Types	86
Program Breakpoints	87
Read/Write Breakpoints	89
<b>Breakpoint Behavior .....</b>	<b>91</b>
Breakpoint Setting at Run-time	91

Breakpoints after Reset/Power Cycle	92
Onchip Breakpoints Changed by Target Program	94
<b>Breakpoint Handling .....</b>	<b>95</b>
Real-time Breakpoints vs. Intrusive Breakpoints	95
ProgramPass/ProgramFail Breakpoints	97
Break.Set Dialog Box	100
The HLL Check Box	101
Implementations	105
Actions	106
Options	107
DATA Breakpoints	111
Advanced Breakpoints	115
TASK-aware Breakpoints	116
Counter	119
CONDITION	122
CMD	128
Display a List of all Set Breakpoints	131
Delete Breakpoints	131
Enable/Disable Breakpoints	132
Store Breakpoint Settings	133
<b>Debugging .....</b>	<b>134</b>
Basic Debug Control	134
Debugging of Optimized Code	147
<b>Document your Results .....</b>	<b>150</b>
Settings	150
Print	151
Clipboard	151
File	152
Quick Output	153
Advanced Output	155

## 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.

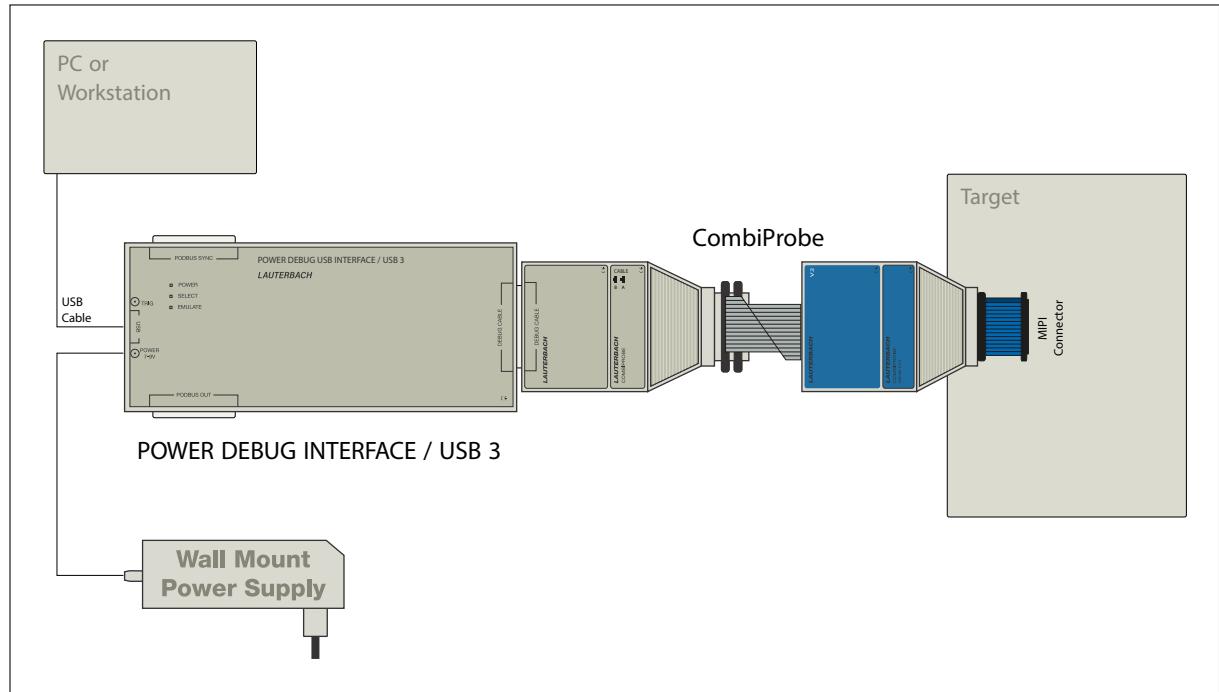
- **Core traces**

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

---

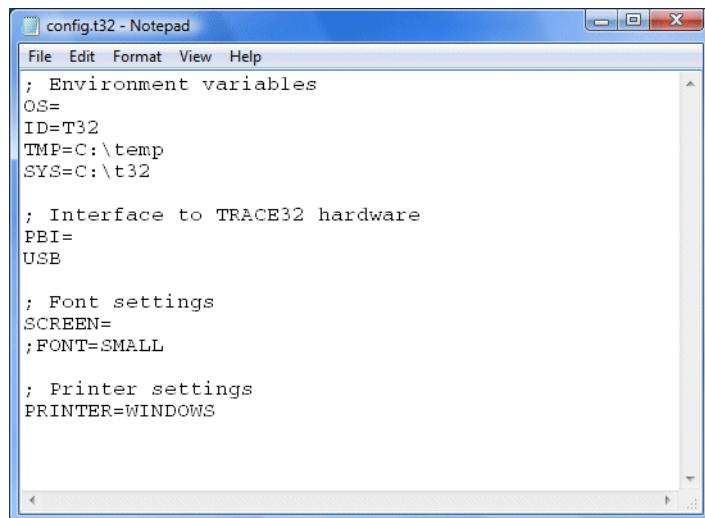
This chapter describes the basic parameters required to start a TRACE32 PowerView instance.

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.



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

; Interface to TRACE32 hardware
PBI=
USB

; Font settings
SCREEN=
;FONT=SMALL

; Printer settings
PRINTER=WINDOWS
```

The following rules apply to the configuration file:

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

## Standard Parameters

Parameter	Syntax	Description
Host interface	PBI=<host_interface>	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)

	<p>(*) In order to display source code information TRACE32 PowerView creates a copy of all loaded source files and saves them into the TMP directory.</p> <p>(**) The TRACE32 online help is PDF-based.</p>
-------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 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

---

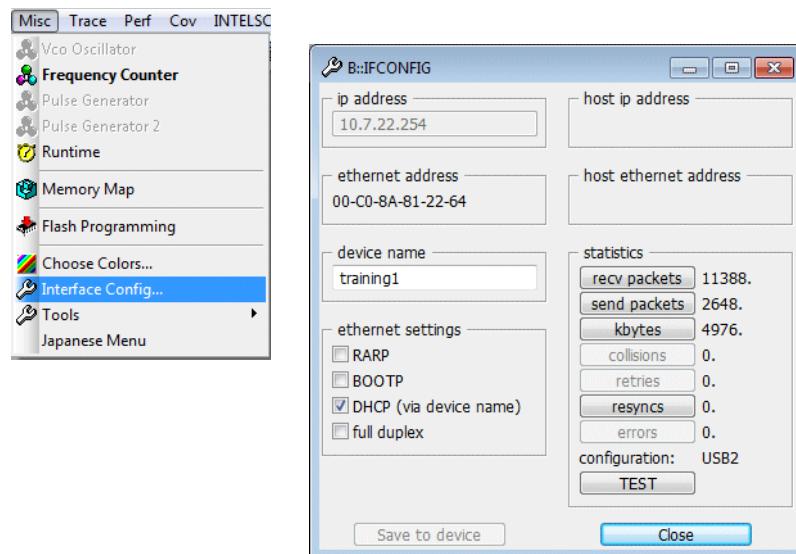
TRACE32 allows to communicate with a POWER DEBUG INTERFACE USB from a remote PC. For an example, see “[Example: Remote Control for POWER DEBUG INTERFACE / USB](#)” in TRACE32 Installation Guide, page 48 (installation.pdf).

```
; 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

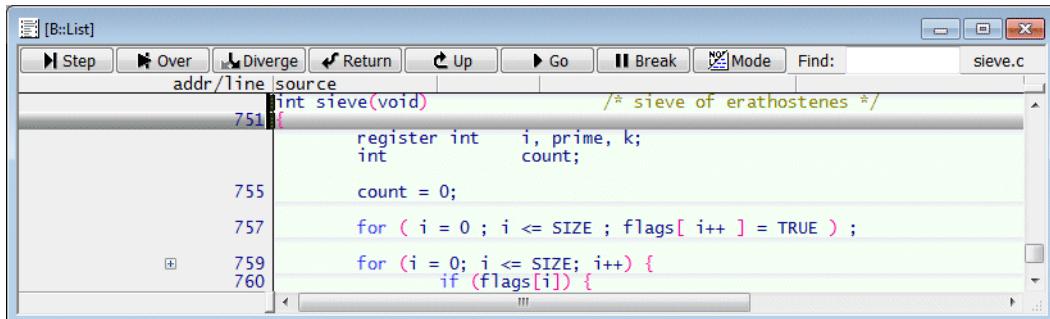
Display and change information for the Ethernet interface

## Additional Parameters

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

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

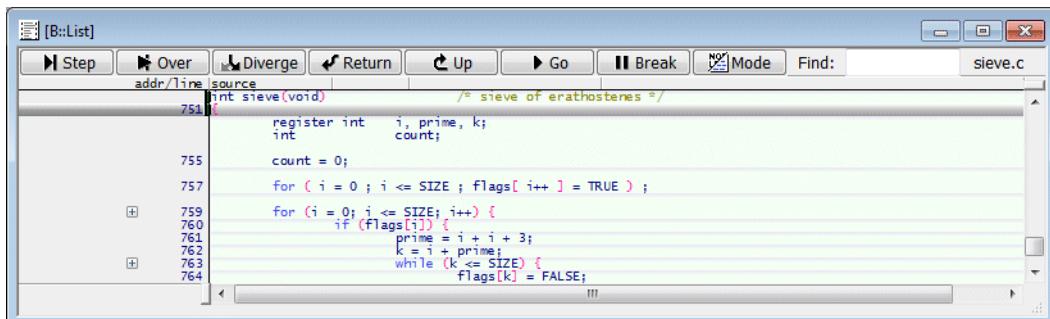
### Display with normal fonts:



A screenshot of the TRACE32 debugger interface. The window title is [B::List]. The menu bar includes Step, Over, Diverge, Return, Up, Go, Break, Mode, and Find, with Mode currently selected. The file name is sieve.c. The code window displays the following C code:

```
int sieve(void) /* sieve of erathostenes */
751 {
    register int i, prime, k;
    int count;
    count = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
    for ( i = 0; i <= SIZE; i++) {
        if (flags[i]) {
```

### Display with small fonts:



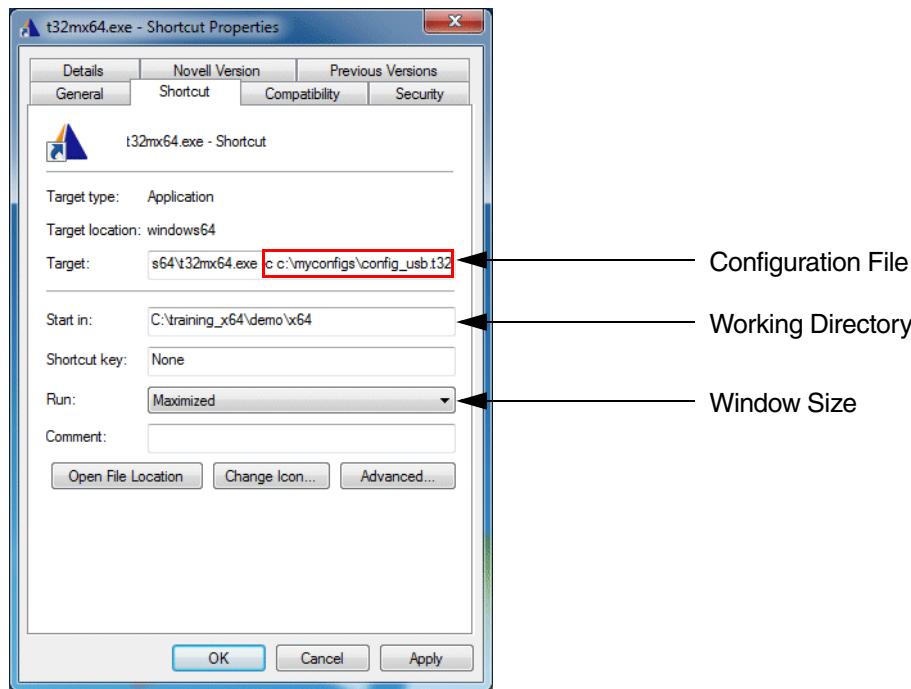
A screenshot of the TRACE32 debugger interface, identical to the previous one but with a smaller font size for the text. The window title is [B::List]. The menu bar includes Step, Over, Diverge, Return, Up, Go, Break, Mode, and Find, with Mode currently selected. The file name is sieve.c. The code window displays the same C code as the previous screenshot.

```
int sieve(void) /* sieve of erathostenes */
751 {
    register int i, prime, k;
    int count;
    count = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
    for ( i = 0; i <= SIZE; i++) {
        if (flags[i]) {
```

# 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

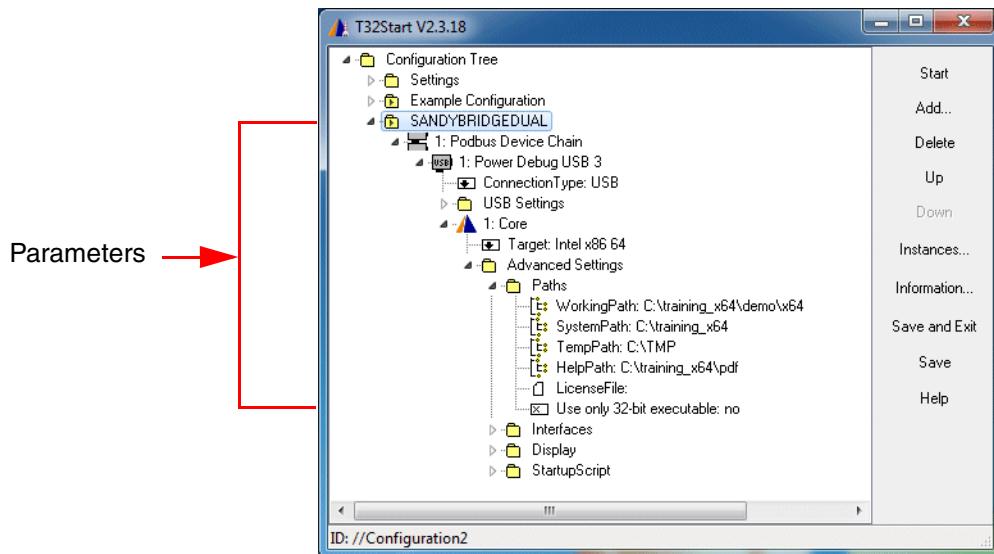
---

You can choose between Normal window, Minimized and Maximized.

## Configuration via T32Start (Windows only)

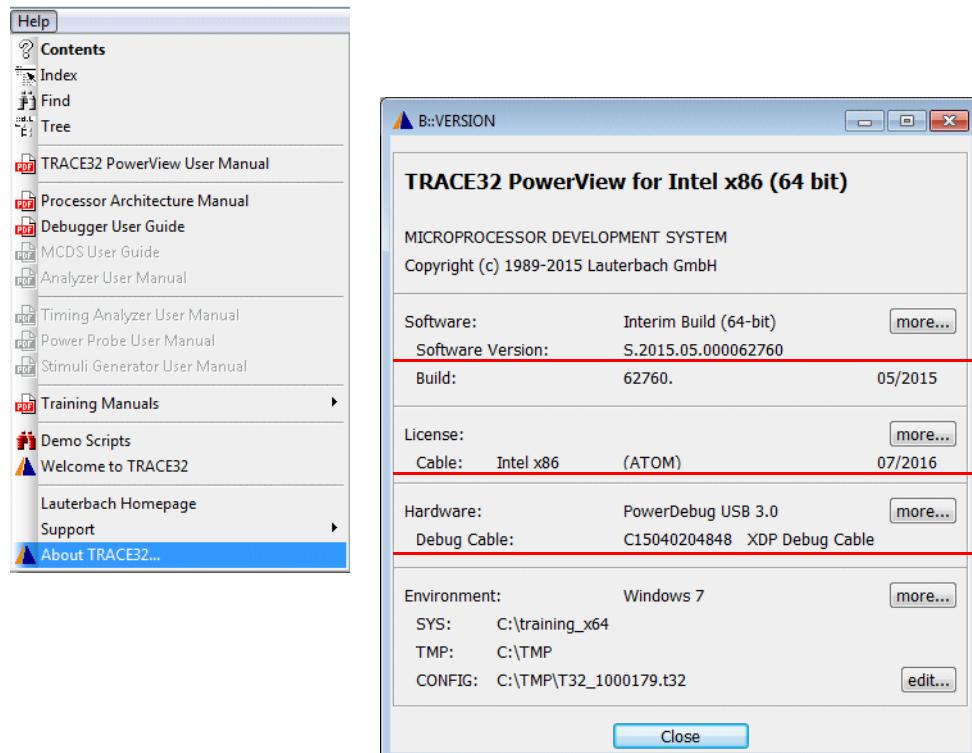
The basic parameters can also be set up in an intuitive way via **T32Start**.

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



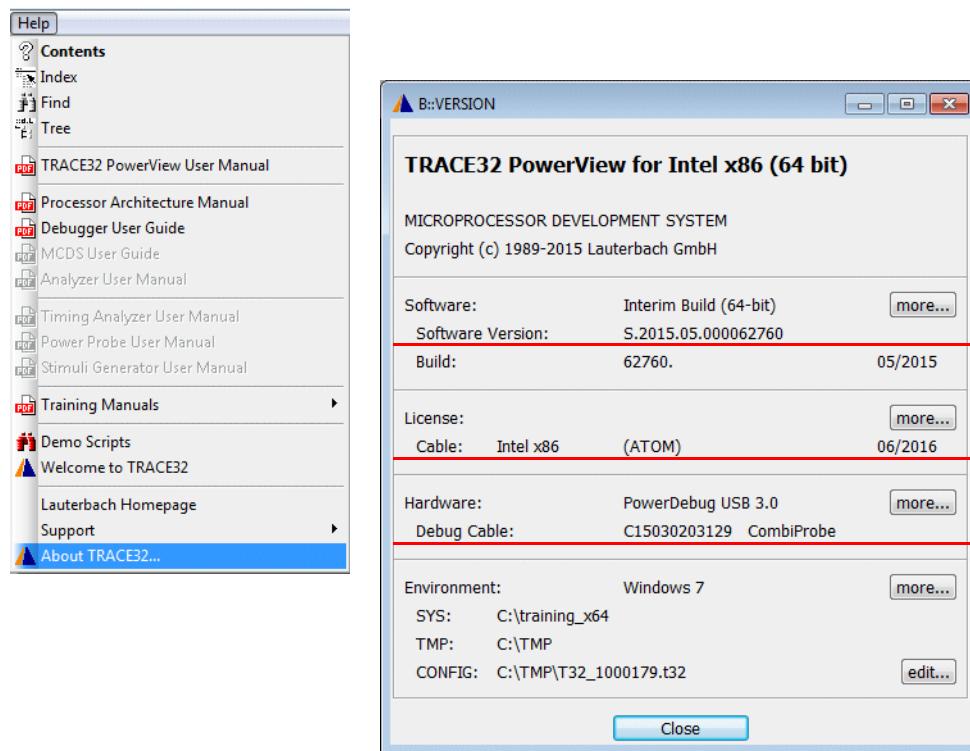
If you want to contact your local Lauterbach support, it might be helpful to provide some basis information about your TRACE32 tool.

## 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.

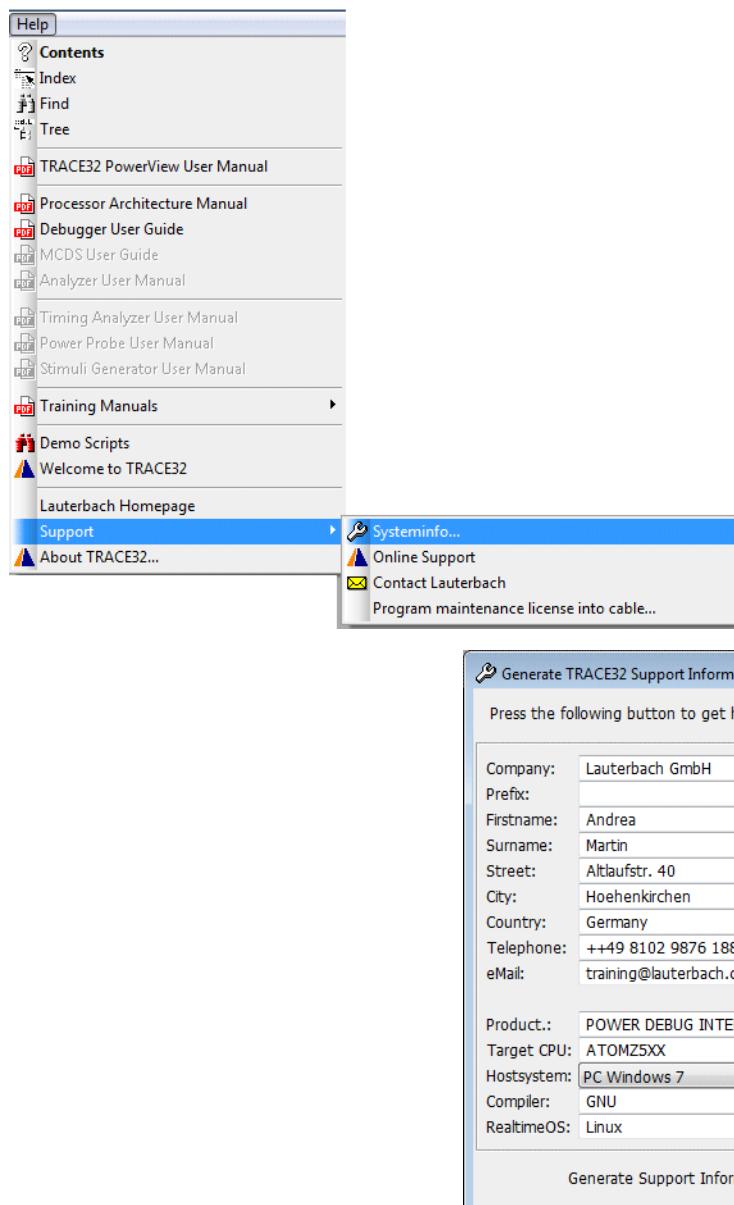
**VERSION.SOFTWARE**

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

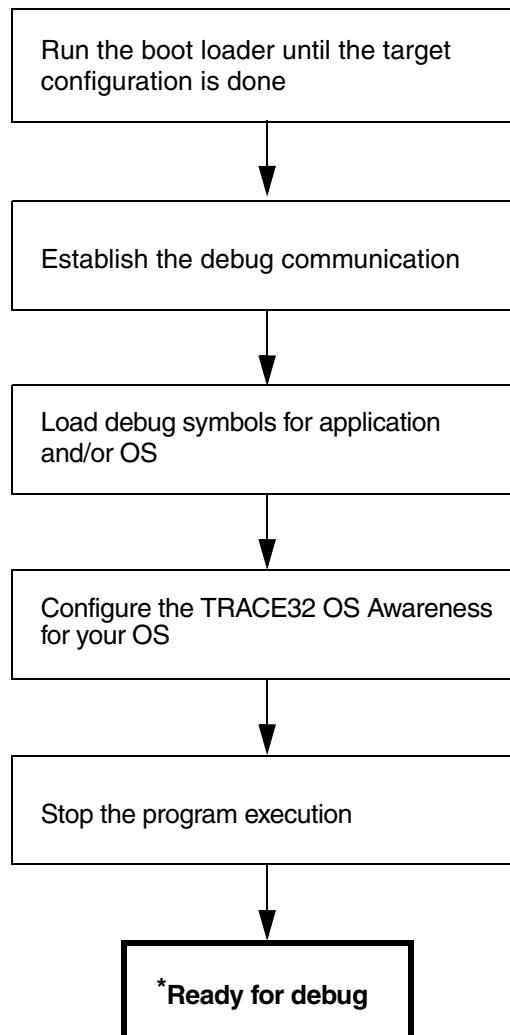
---

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

## Course of Action

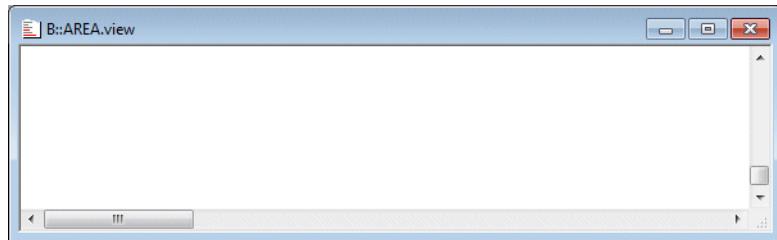
---

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**

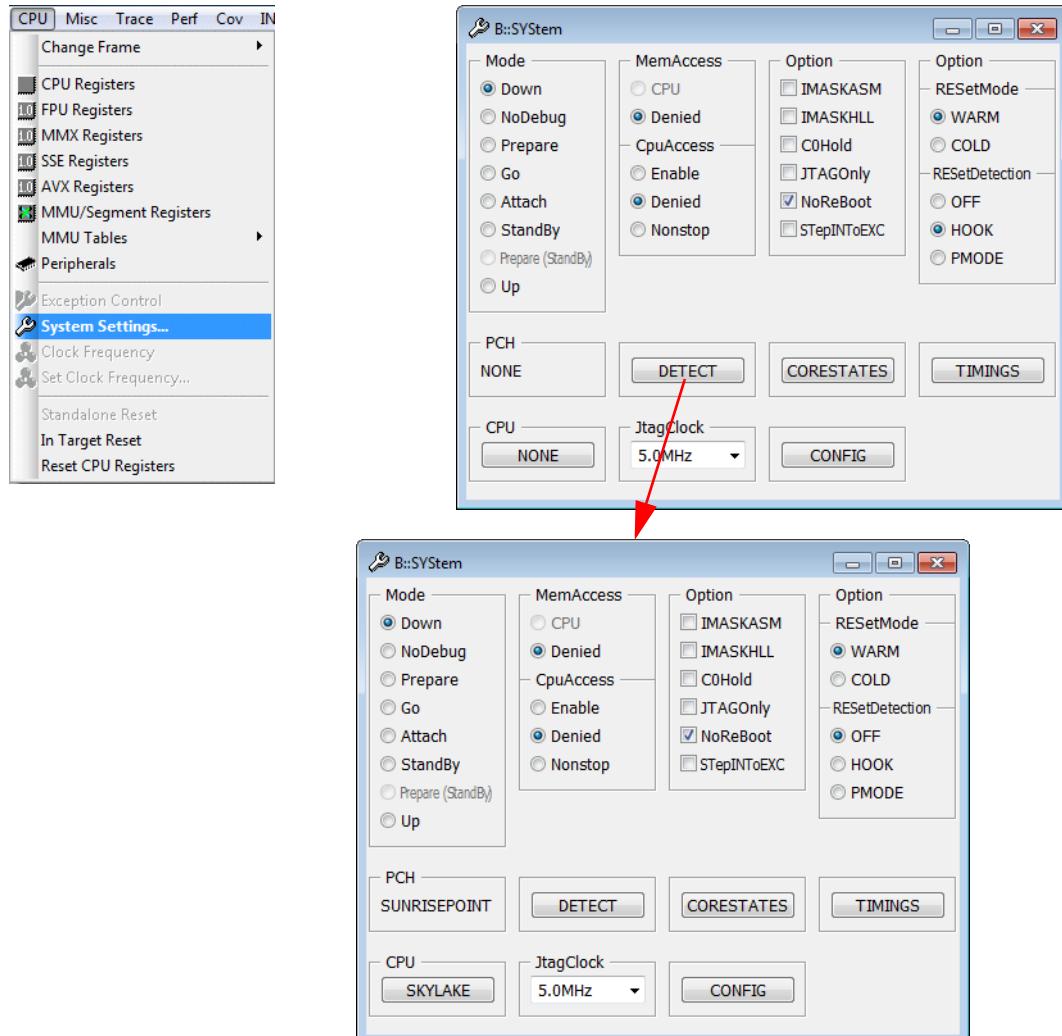
Open an AREA window

# Run the Boot Loader until the Target Configuration is Done

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

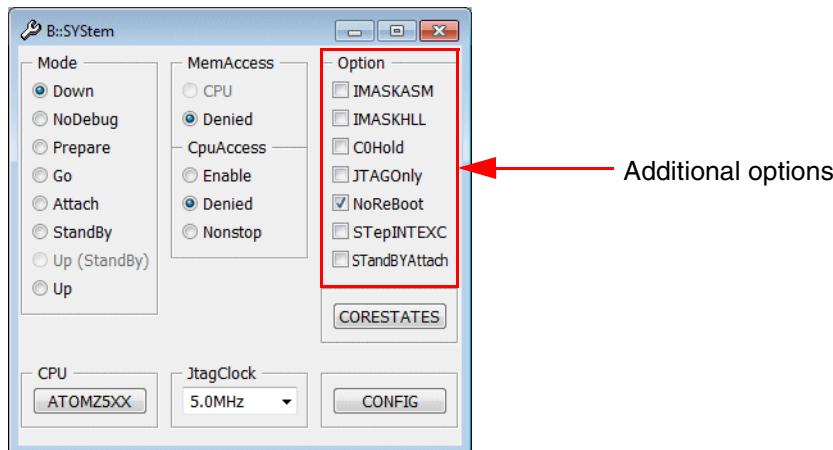
## Establish the Debug Communication

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:

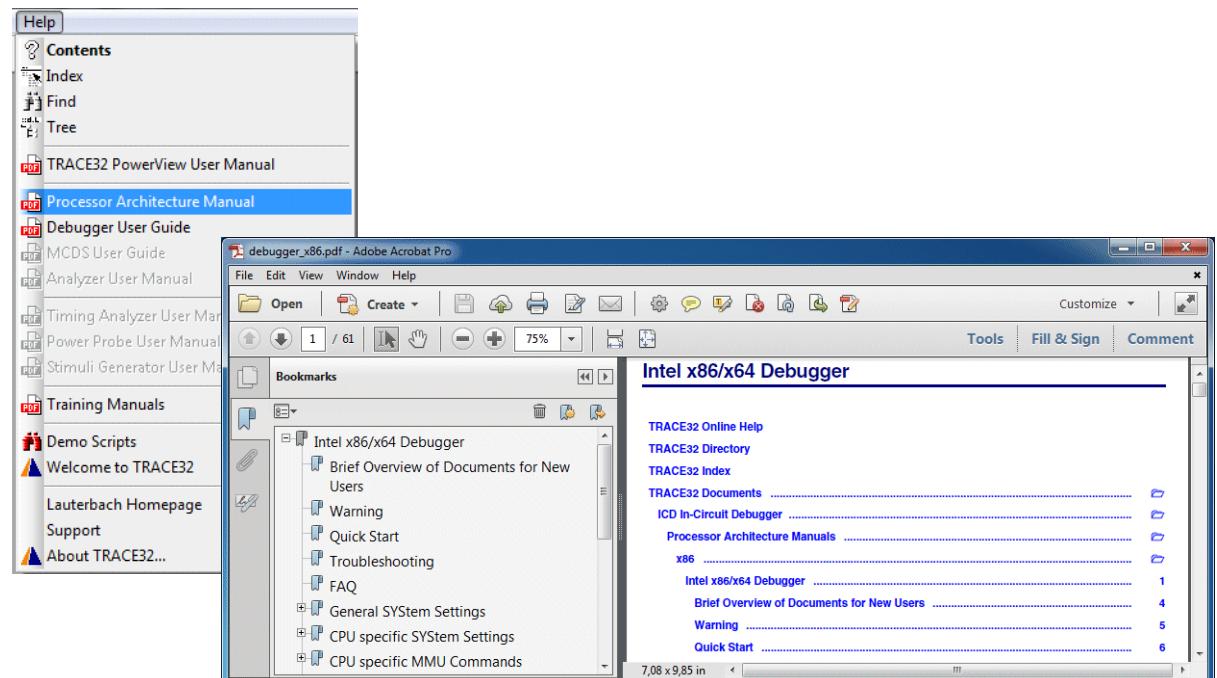


SYStem.DETECT TARGET

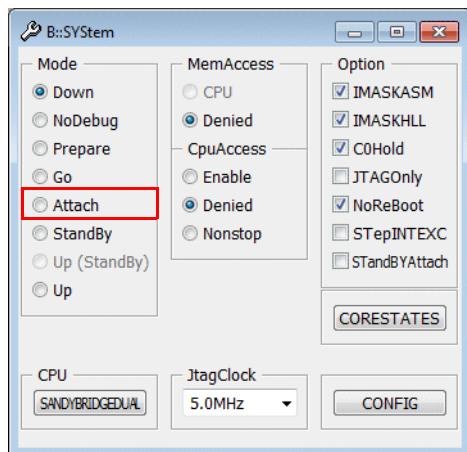
Then the options required for your chip have to be set.



For details on the available options, refer to [“Intel® x86/x64 Debugger”](#) (debugger\_x86.pdf).

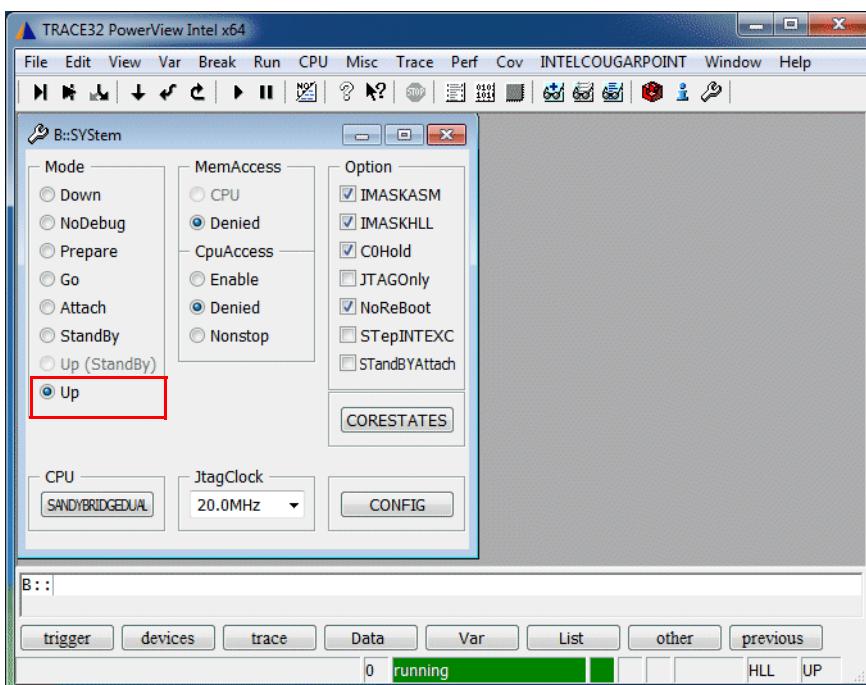


Choose the **Attach** radio button to establish the communication between the debugger and the target chip.



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**

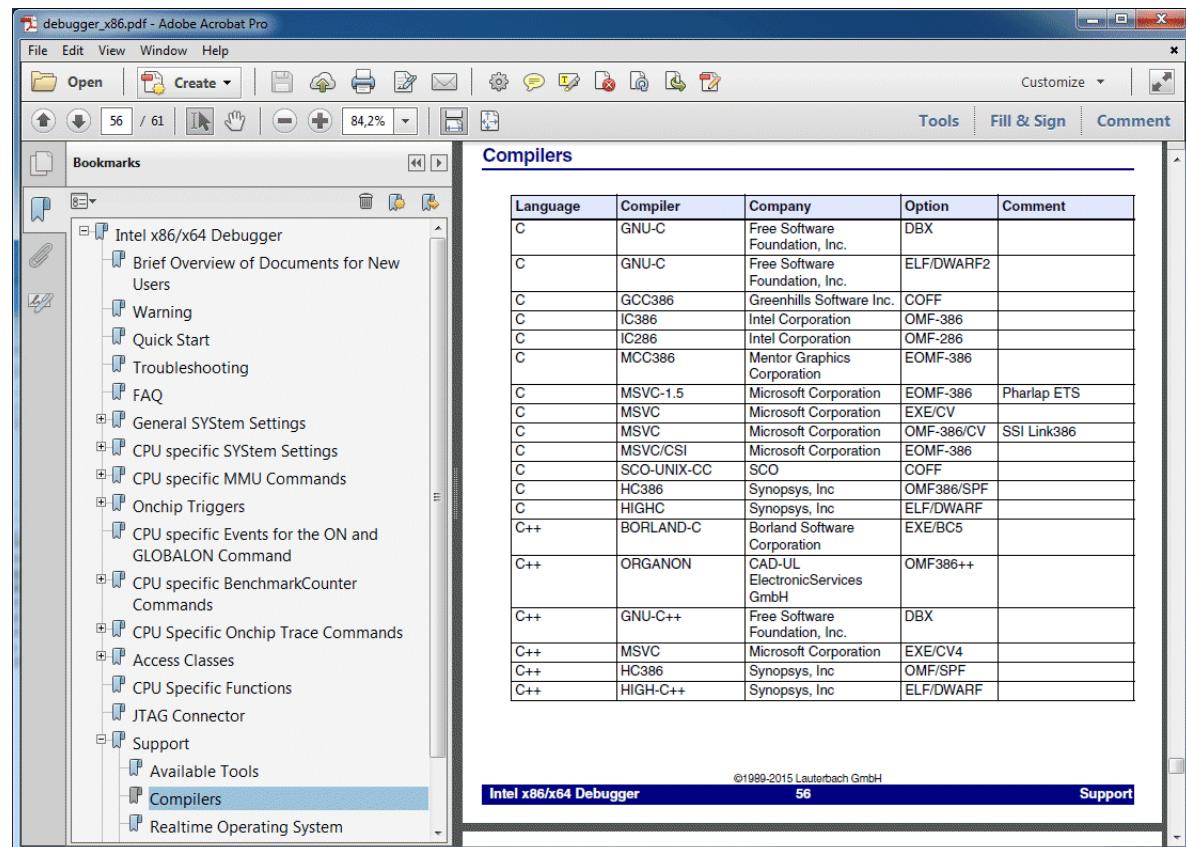
Establish the communication between the debugger and the target chip.



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**.

# Load the Debug Symbols for the Application and/or the OS

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.

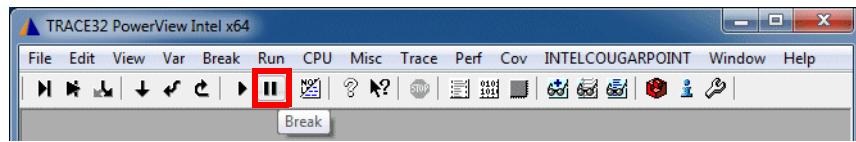
Please refer to “[OS Awareness Manual Windows Standard](#)” (rtos\_windows.pdf) on how to activate the TRACE32 Windows awareness on your target.

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

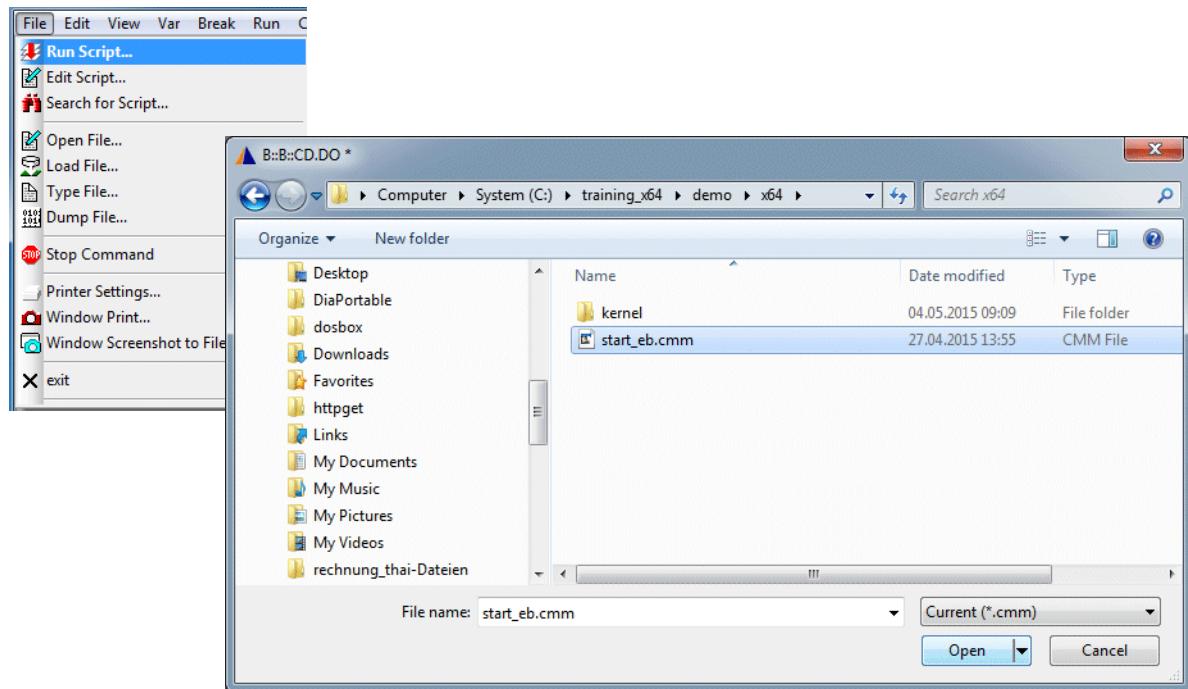
```
PEDIT system_settings
```

; 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
```

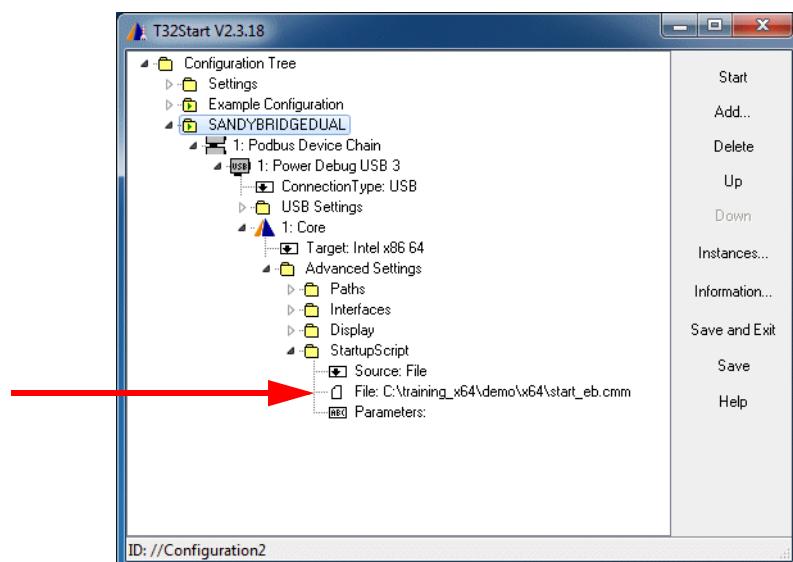
There are two ways to define a start-up script, that is automatically started, when the debugger is started.

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

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

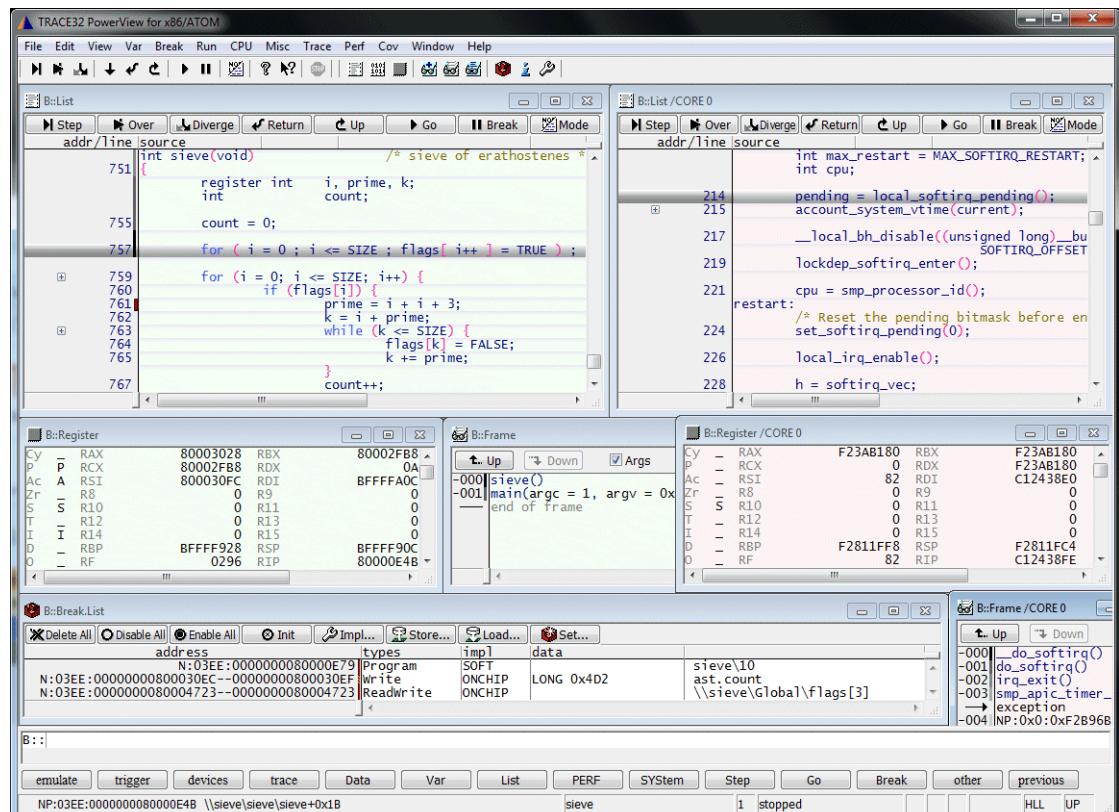
```
c:\t32\t32mx64.exe -s g:\and\training\start.cmm
```

## 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:



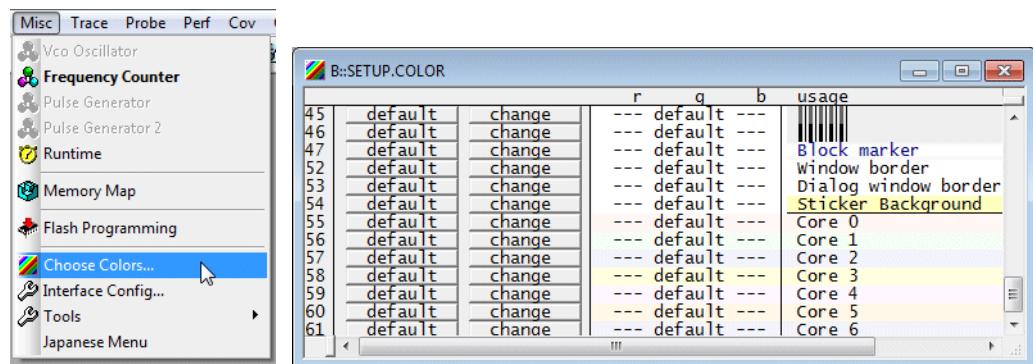
**CORE.select <number>**      Select a different core

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.

TRACE32 PowerView for x86/ATOM

File Edit View Var Break Run CPU Trace Perf Cov Window Help

B:List

Step Over Diverge Return Up Go Break Mode

addr/line source

```

214     pending = local_softirq_pending();
215 account_system_vtime(current);
217 __local_bh_disable((unsigned long)_builtin_return
218                      SOFTIRQ_OFFSET);
219 lockdep_softirq_enter();
220 cpu = smp_processor_id();
221 restart: /* Reset the pending bitmask before enabling irqs
222         set_softirq_pending(0);

```

B:Register

CY	RAX	F23AB180	RBX	F23AB180
P	RCX	0	RDX	F23AB180
Ac	RSI	82	RDI	C12438E0
Zr	R8	0	R9	0
S	R10	0	R11	0
R	R12	0	R13	0
I	R14	0	R15	0
D	RBP	F2811FF8	RSP	F2811FC4
O	RF	82	RIP	C12438FE
PL	0			
NT	CRO	8005003B	DR0	0
CR2	425C5000	DR1		
VM	003	01F80000	DR2	0
AC	CR4	001007D8	DR3	0
CR8	0	DR6		FFFF0FF0
XCR0	0	DR7		0400

B:Break.List

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

address	types	impl	data
N:03EE:00000000800000E79	Program	SOFT	sieve\10
N:03EE:000000008000030EC	Write	ONCHIP	ast.count
N:03EE:0000000080004723	Readwrite	ONCHIP	\sieve\Global\flags[3]

B:Var.View ast

ast = (

- word = 0x0,
- left = 0x800030E8,
- right = 0x0,
- field1 = -1,
- field2 = 2,

B:List /CORE 1

Step Over Diverge Return Up Go Break Mode

addr/line source

```

755     count = 0;
756
757     for (i = 0 ; i <= SIZE ; flags[i]++)
758     {
759         if (Flags[i])
760         {
761             prime = i + i + 3;
762             k = i + prime;
763             while (k <= SIZE)
764             {
765                 flags[k] = F

```

B:Register /CORE 1

CY	RAX	80003028	RBX	80002FB8
P	RCX	80002FB8	RDX	0A
Ac	RSI	800030FC	RDI	BFFFFFFA
Zr	R8	0	R9	0
S	R10	0	R11	0
R	R12	0	R13	0
I	R14	0	R15	0
D	RBP	BFFF928	RSP	BFFF90C
O	RF	0296	RIP	80000E4B
PL	0			
NT	CRO	8005003B	DR0	0
CR2	428375E0	DR1		
VM	28095000	DR2		
AC	CR4	001007D8	DR3	0
CR8	0	DR6		FFFF4FF0

B:Data.dump (flags) /DIALOG

ND:0x0EE:0x80004720

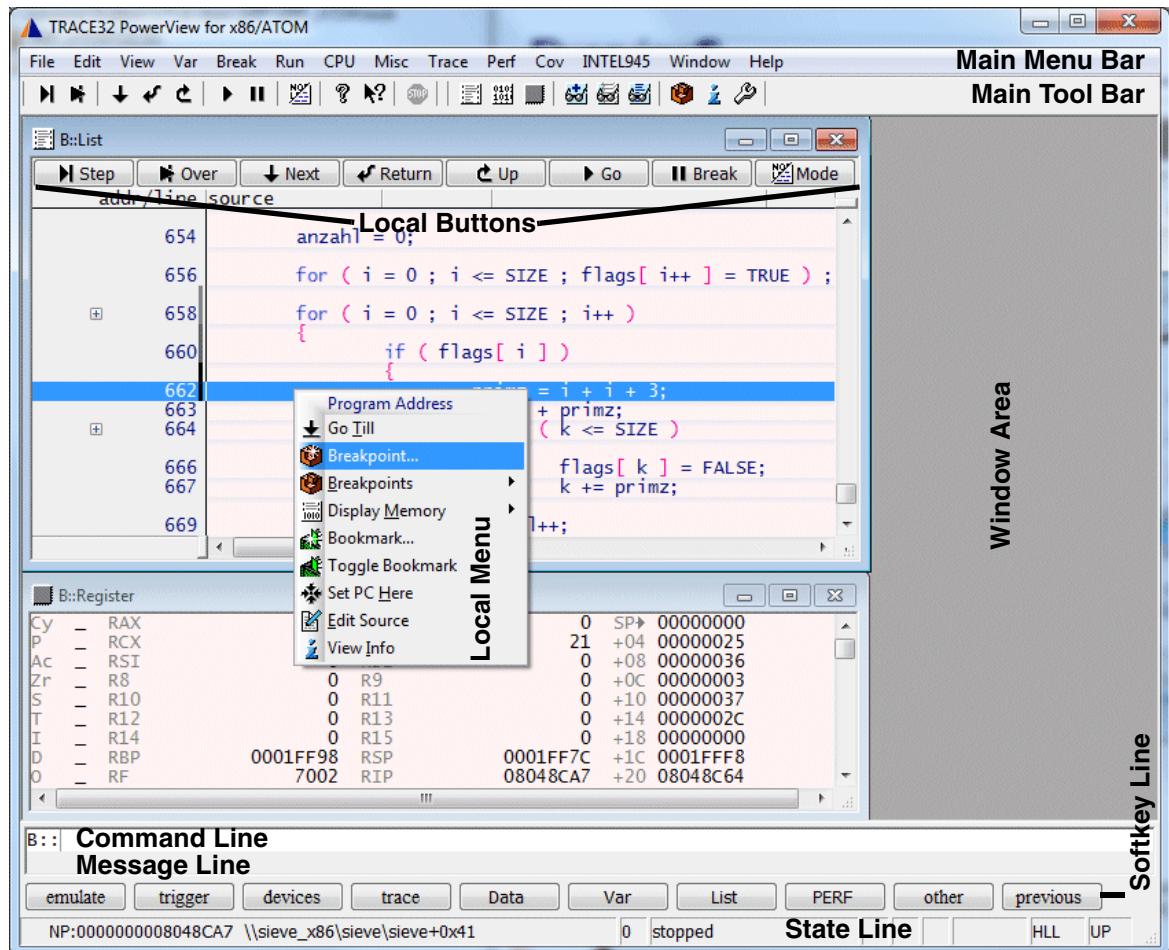
address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ND:03EE:00000000800004720	01	01	01	00	01	00	01	00	00	01	00	00	00	00	00	00
ND:03EE:0000000080004730	00	01	00	01	00	01	00	01	00	01	00	00	00	00	00	00
ND:03EE:0000000080004740	00	01	00	01	00	01	00	01	00	00	00	00	00	00	00	00
ND:03EE:0000000080004750	00	01	00	01	00	01	00	00	00	00	00	00	00	00	00	00

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



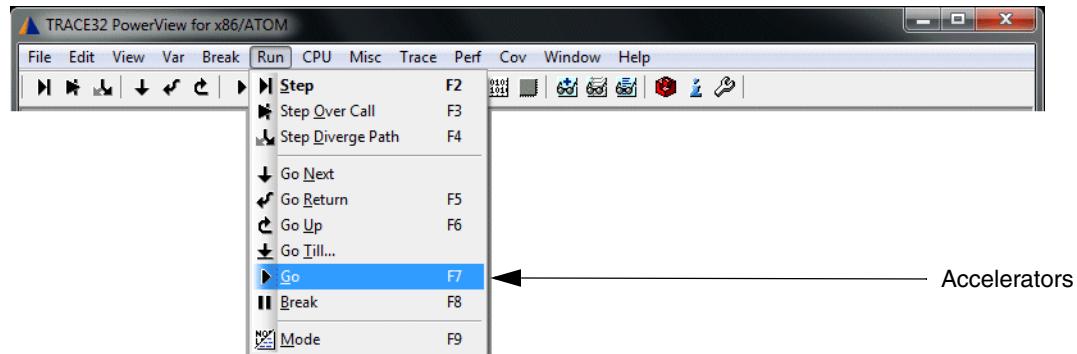
 The structure and functionality of TRACE32 PowerView is largely defined by the file **t32.men** which is located in the TRACE32 system directory.

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.

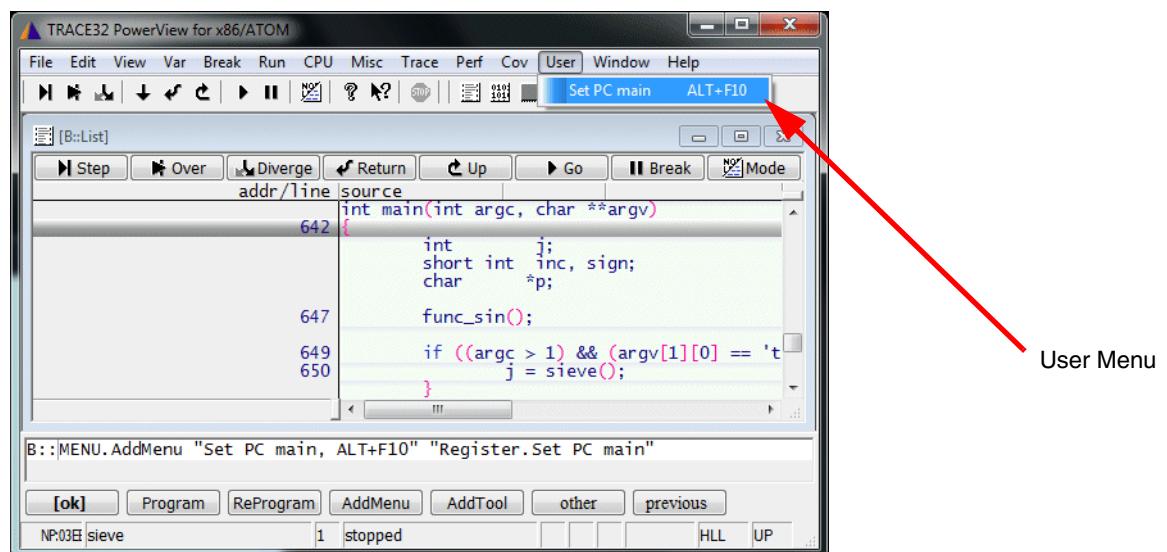
For often used commands accelerators are defined.



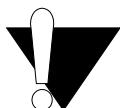
A user specific menu can be defined very easily:

<b>MENU.AddMenu</b> <name> <command>	Add a user menu
<b>MENU.RESET</b>	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"
```



User Menu



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

## Main Tool Bar

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

The user can add his own buttons very easily:

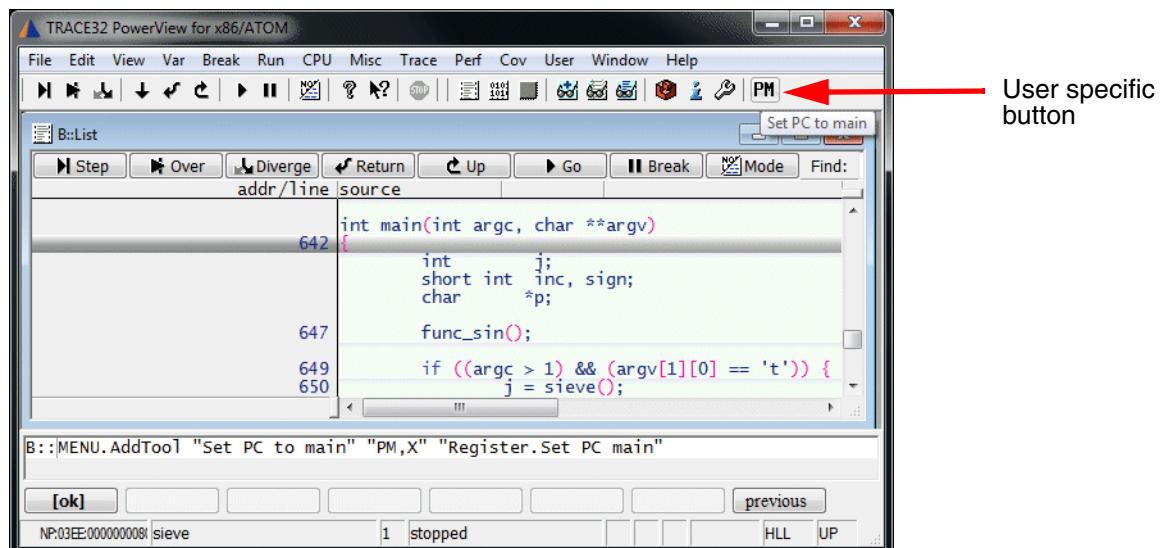
**MENU.AddTool** <tooltip text> <tool image> <command>

Add a button to the toolbar

**MENU.RESet**

Reset menu to default

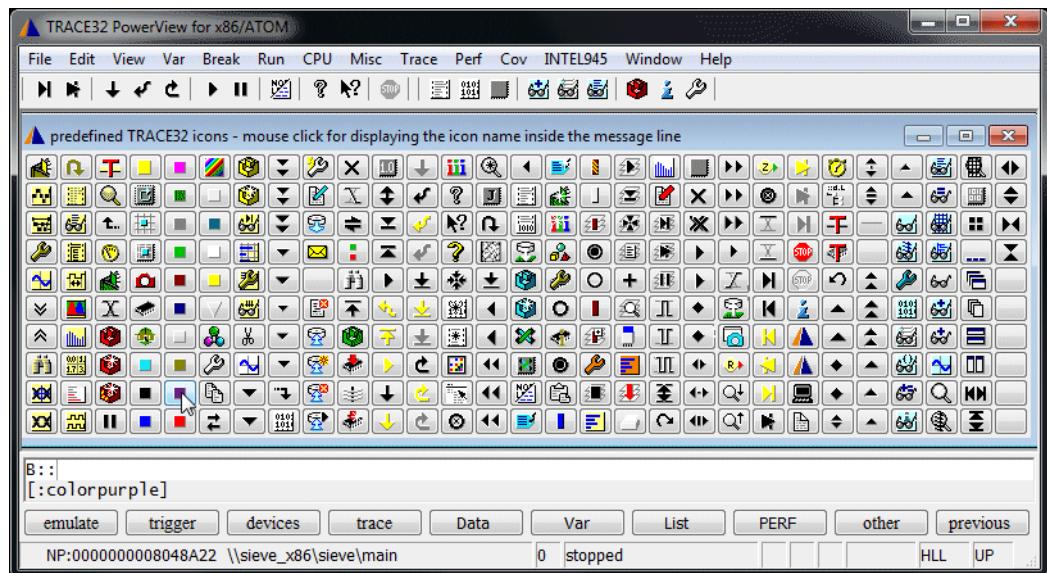
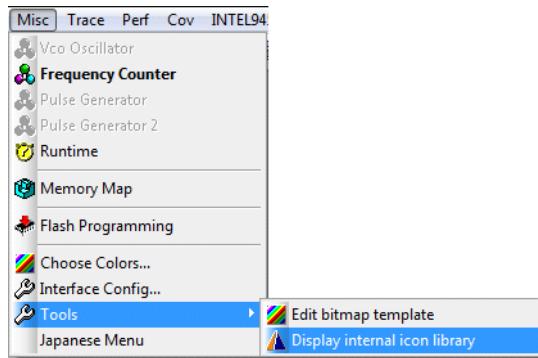
```
MENU.AddTool "Set PC to main" "PM,X" "Register.Set PC main"
```



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

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

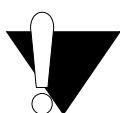
All predefined TRACE32 icons can be inspected as follows:



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

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

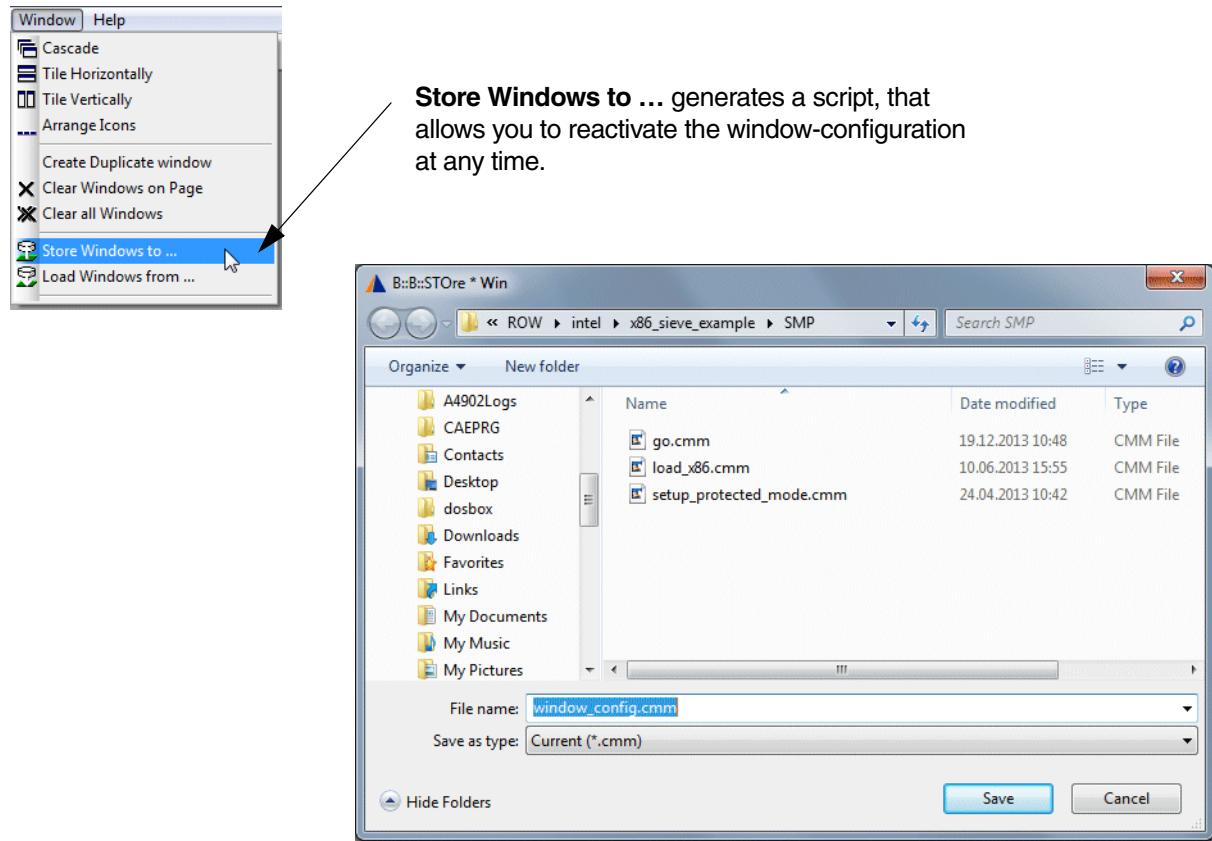
```
; overprint the icon colorpurple with the character v in White color
Menu.AddTool "Set PC to main" "v,W,colorpurple" "Register.Set PC main"
```



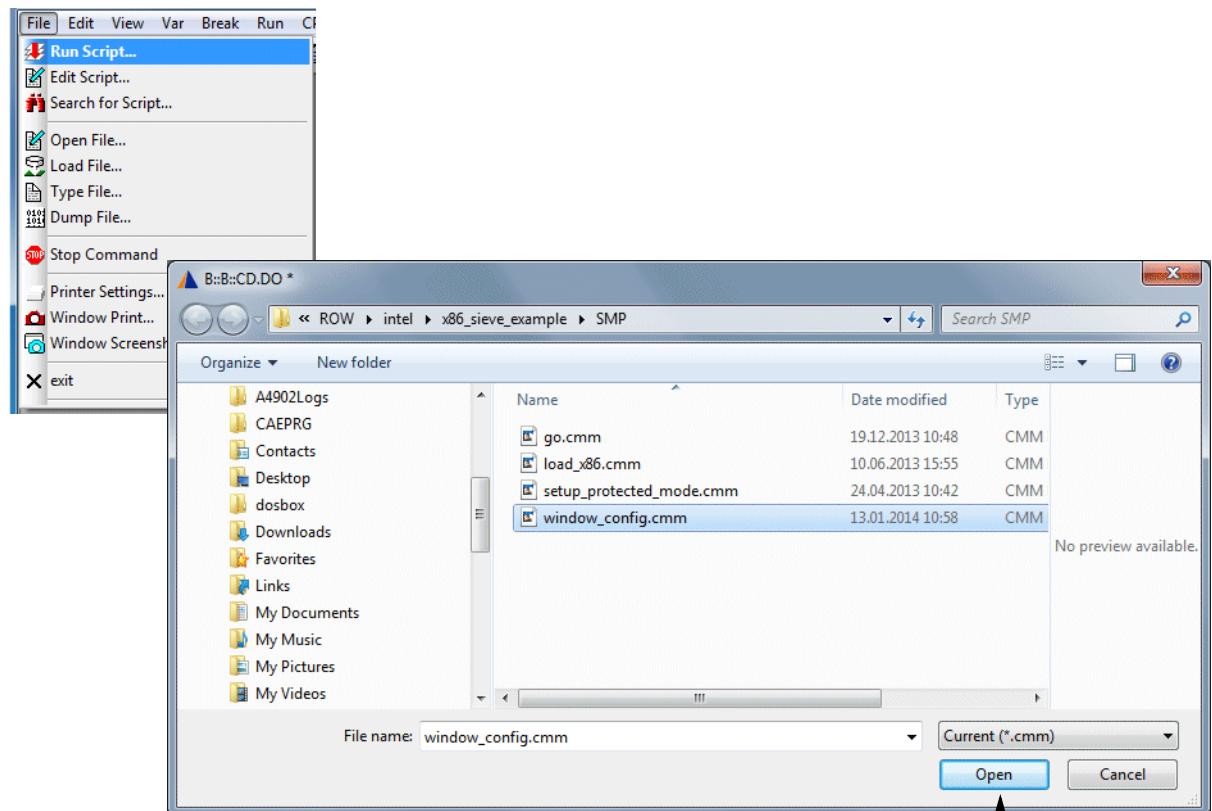
For more complex changes to the main tool bar refer to [“Training Menu Programming”](#) (training\_menu.pdf).

## 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.

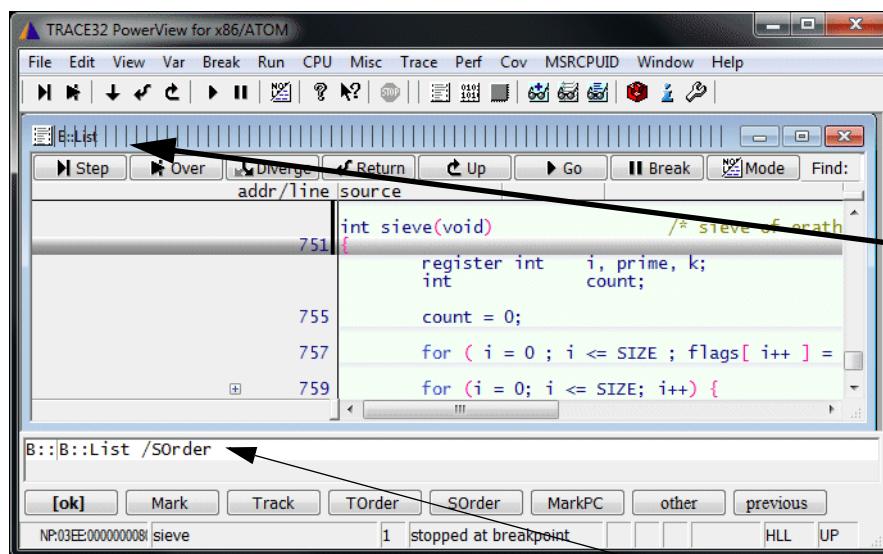


```
// andT32_1000003 Sat Jul 21 16:59:55 2012
B:::
TOOLBAR ON
STATUSBAR ON
FramePOS 68.0 5.2857 107. 45.
WinPAGE.RESet
WinCLEAR
WinPOS 0.0 0.0 80. 16. 15. 1. W000
WinTABS 10. 10. 25. 62.
List
WinPOS 0.0 21.643 80. 5. 25. 1. W001
WinTABS 13. 0. 0. 0. 0. 0. 0.
Break.List
WinPAGE.select P000
ENDDO
```



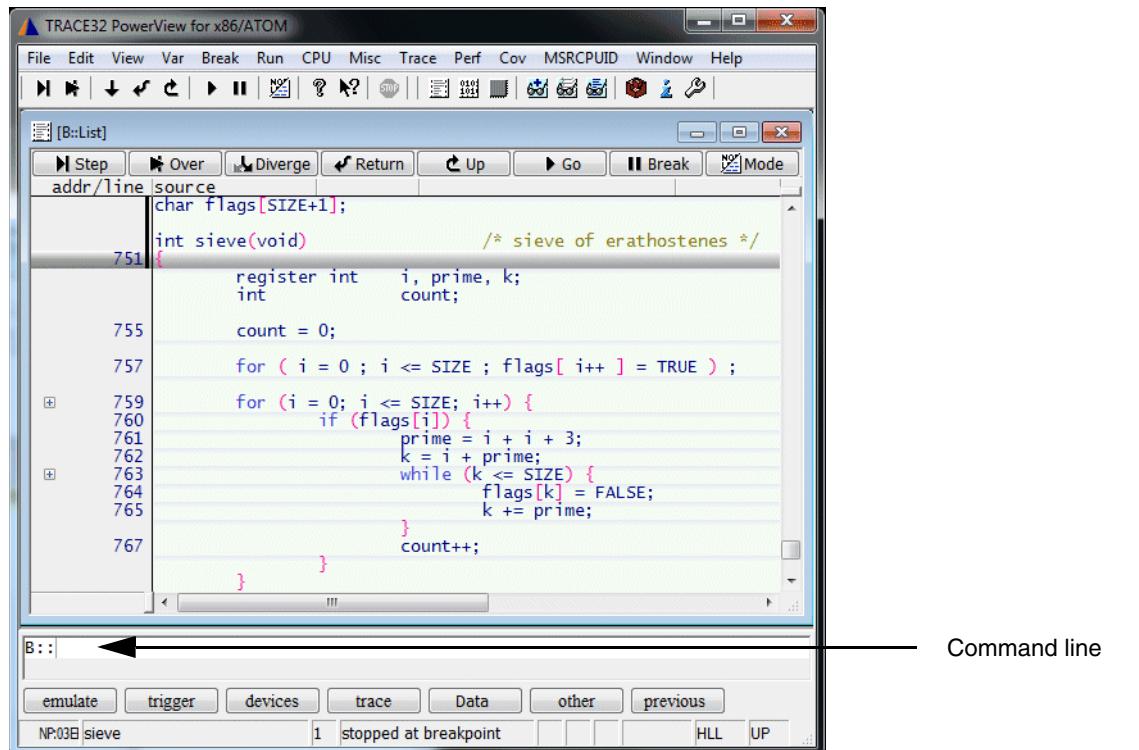
Run the script to reactivate the stored window-configuration

## Modify Window



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

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



## Command 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)

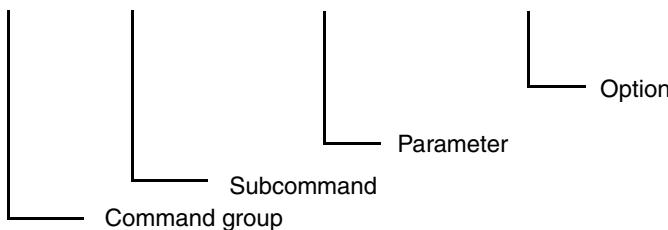
## Command Examples

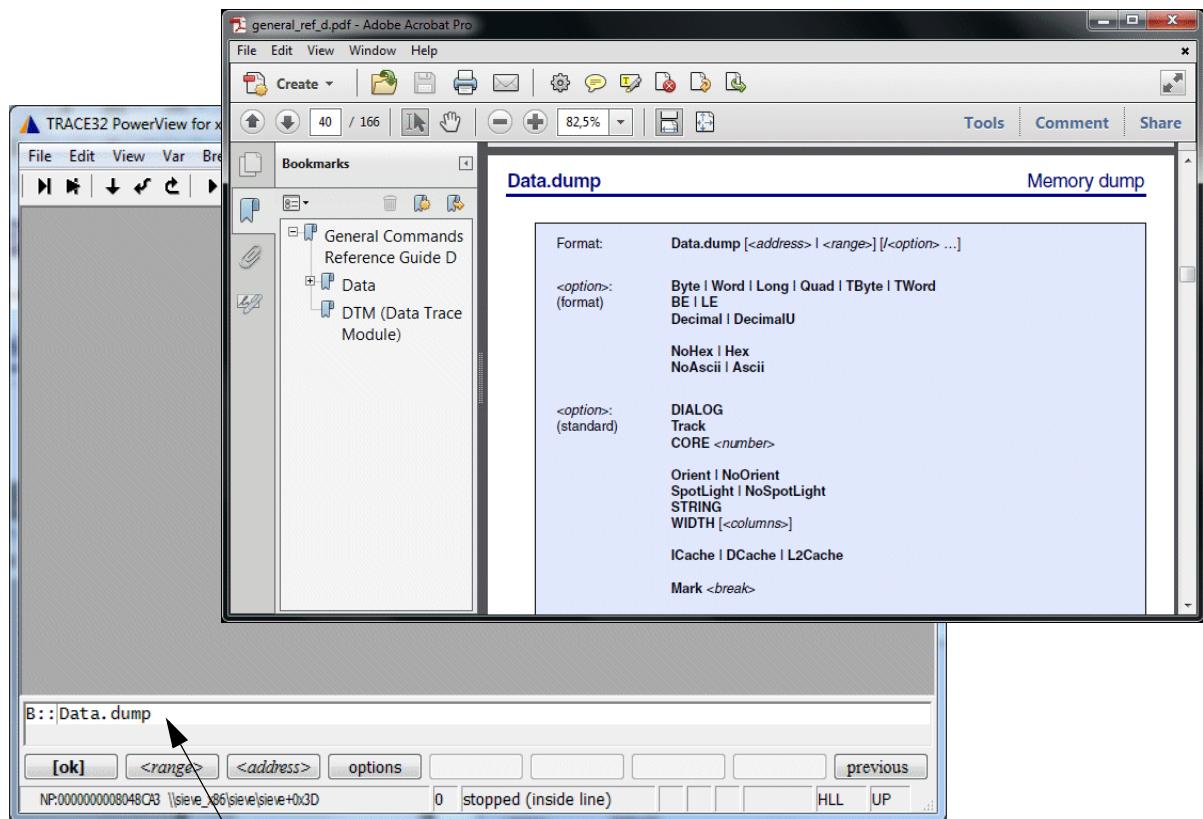
<b>Data</b>	<b>Command group to display, modify ... memory</b>
Data.dump	Displays a hex dump
Data.Set	Modify memory
Data.LOAD.auto	Loads code to the target memory

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

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

Data.dump 0x1000--0x2000 /Byte





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

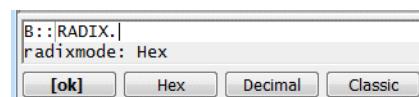
The RADIX defines the input format for numeric values.

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

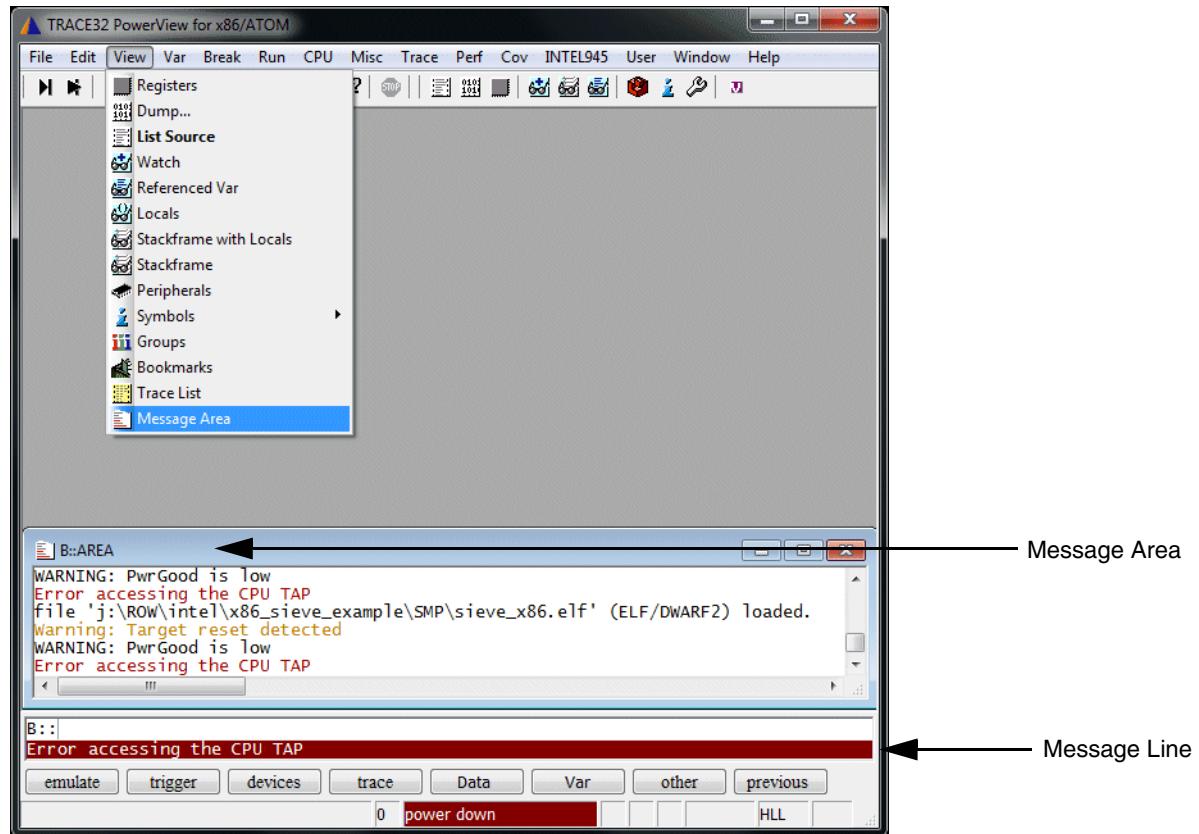
Examples:

	<b>Decimal</b>	<b>Hex</b>
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.



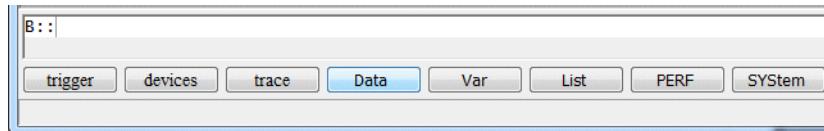
# Message Line



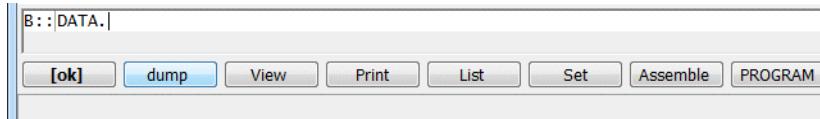
- **Message line** for system and error messages
- **Message Area window** for the display of the last system and error messages

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

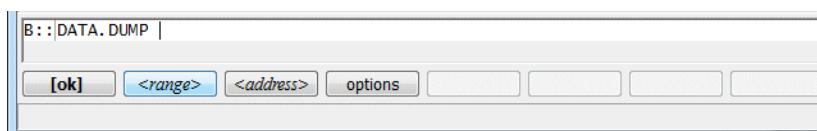
Select the command group, here **Data**.



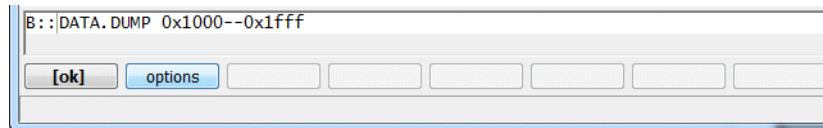
Select the subcommand, here **dump**.



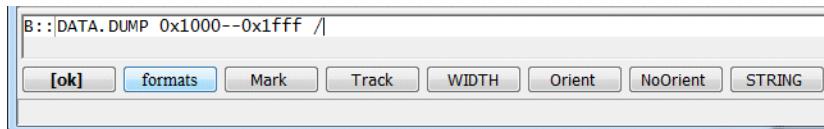
Angle brackets request an entry from the user,  
here e.g. the entry of a *<range>* or an *<address>*.



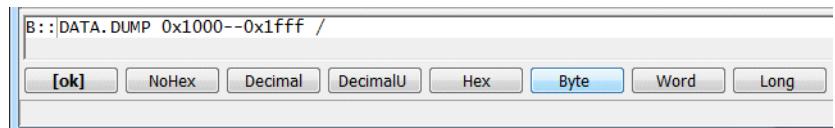
The display of the hex. dump can be adjusted to your needs by an option.



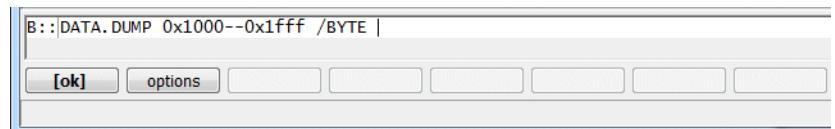
Select the option **formats** to get a list of all format options.

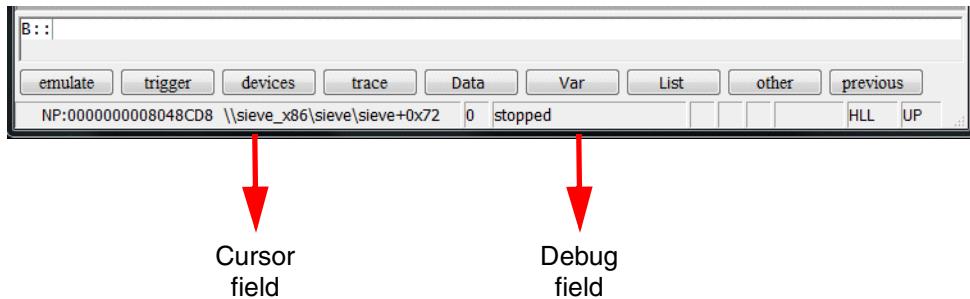


Select a format option, here **Byte**.



The command is complete now.



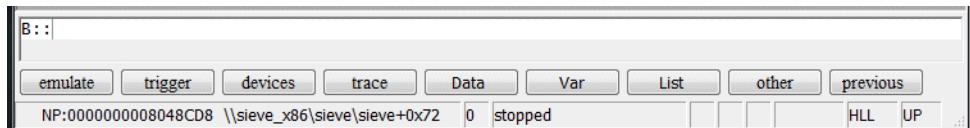


The **Cursor** field provides:

- 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.)



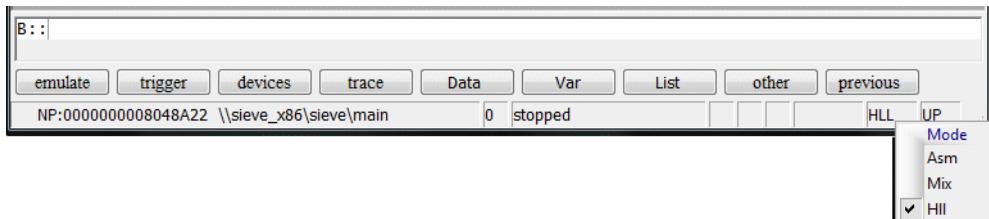
Mode  
field

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

---

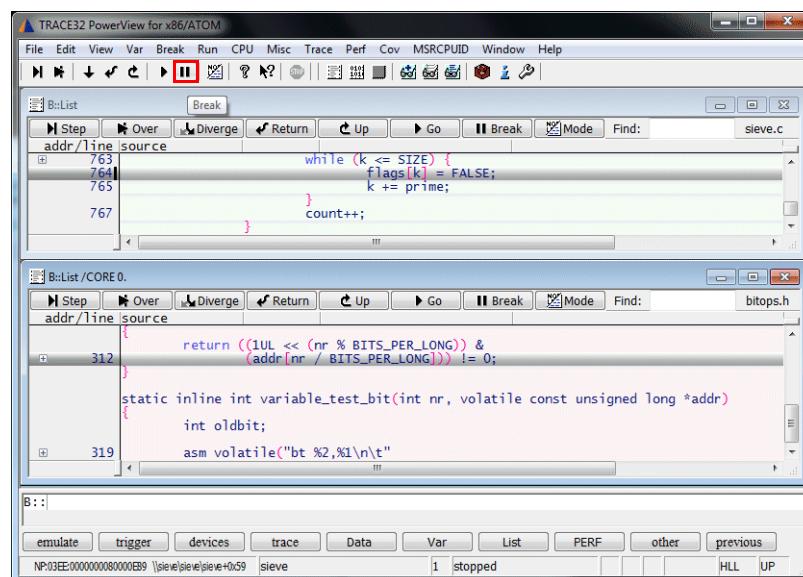
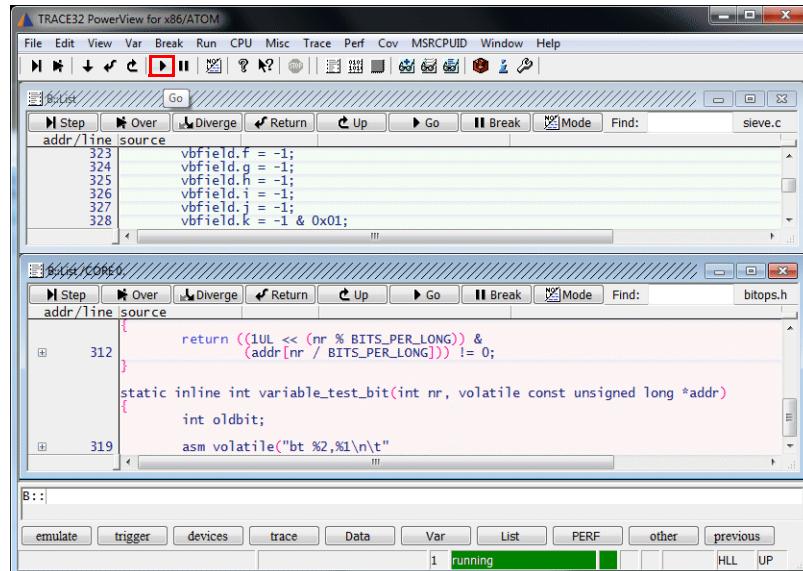
The following PDFs provide detailed information on TRACE32 PowerView:

- [“PowerView User’s Guide”](#) (ide\_user.pdf)
- [“PowerView Command Reference”](#) (ide\_ref.pdf)

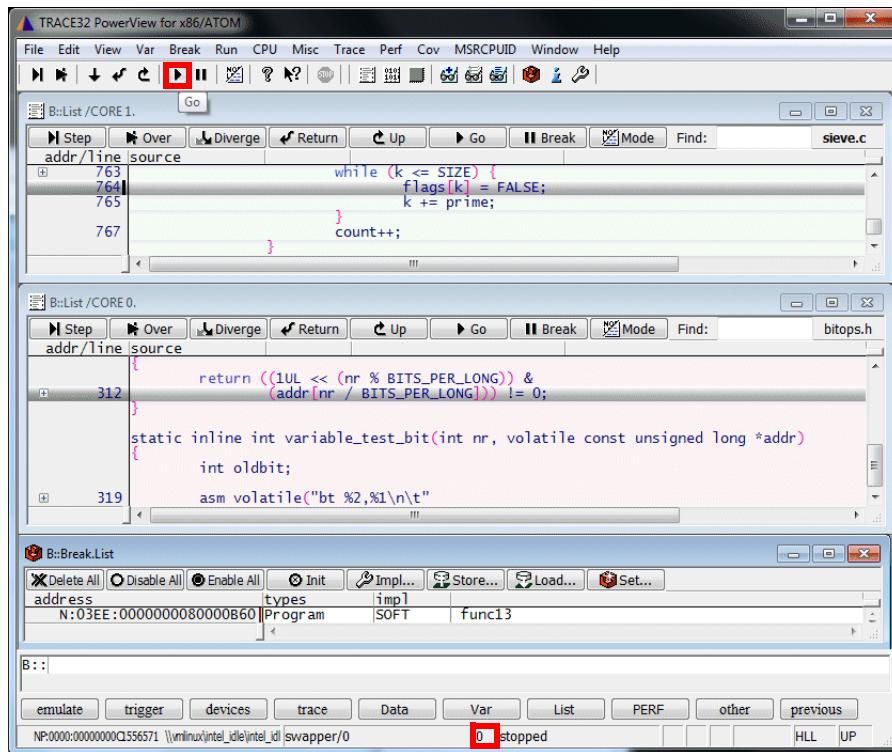
# Basic Debugging (SMP)

## Go/Break

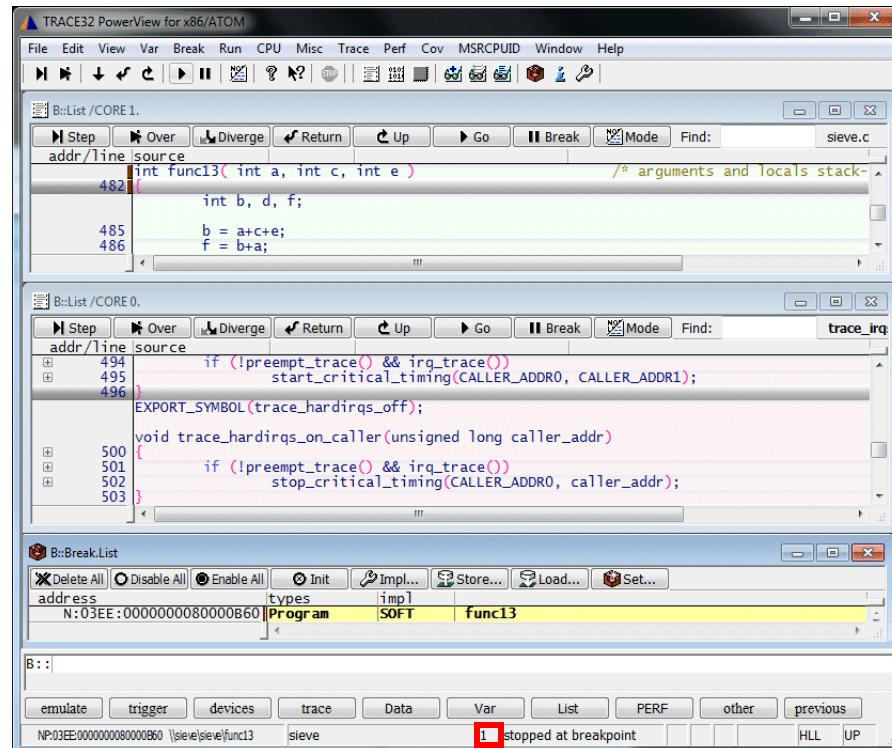
On an SMP systems the program execution is started on all cores with **Go** and is stopped on all cores with **Break**.



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



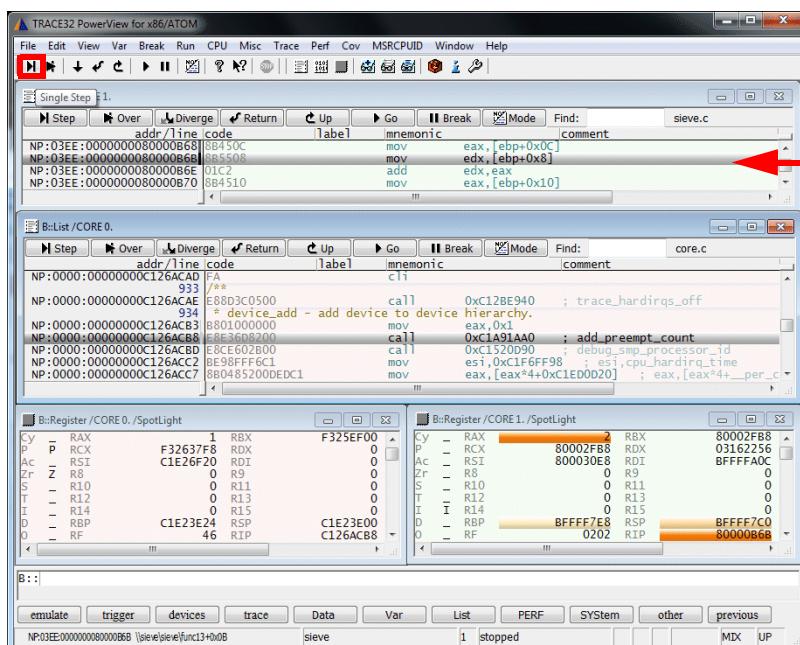
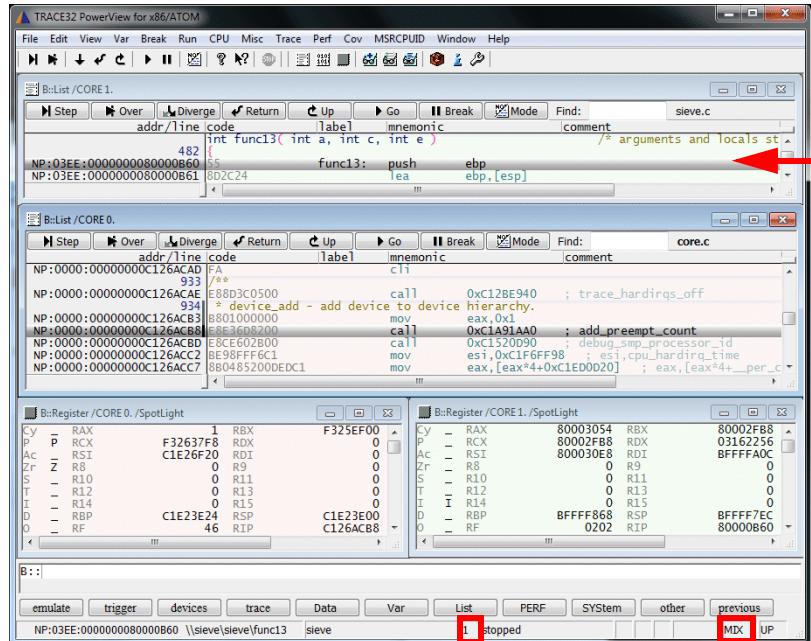
CORE 0 is the selected one when the program execution is started.



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

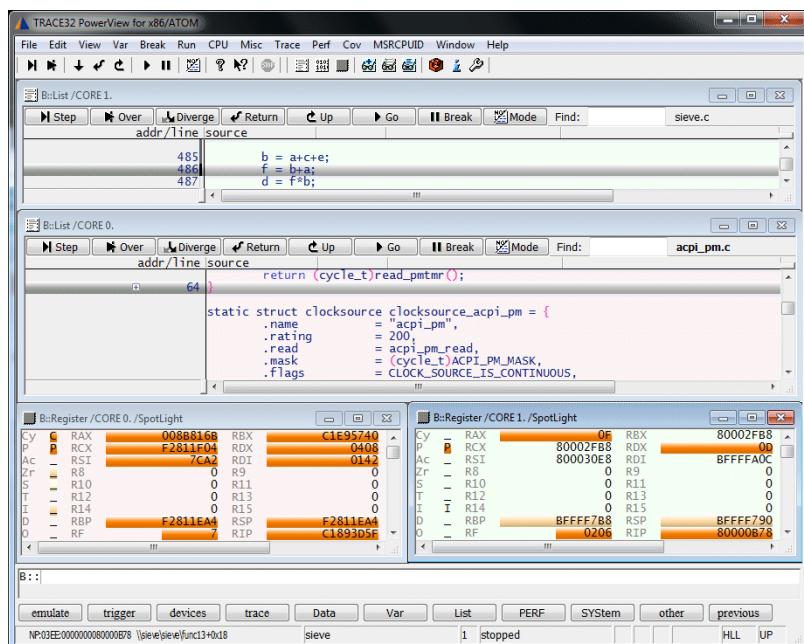
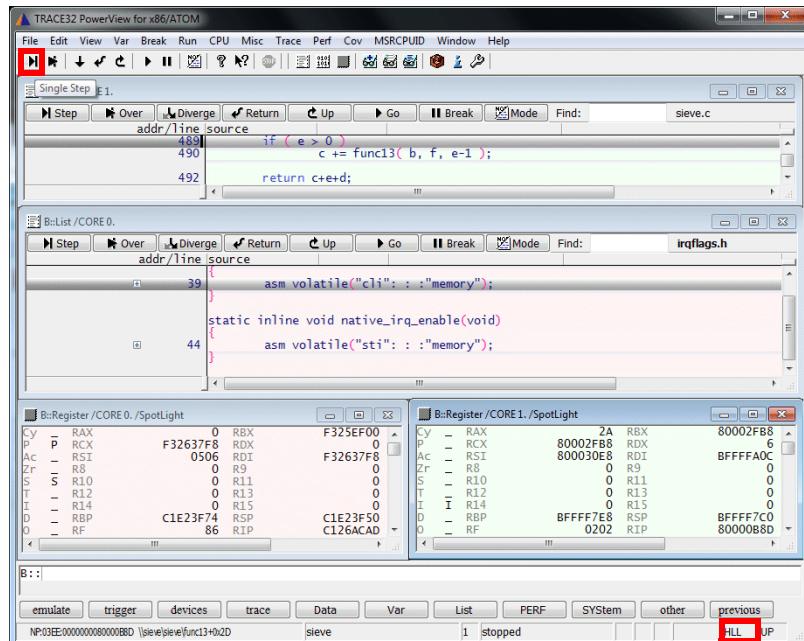
Select Mix mode for debugging and perform a single step on the selected core.

## Step.Asm

Perform an assembler single step on the selected core.

## Single Stepping on High-Level Language Level

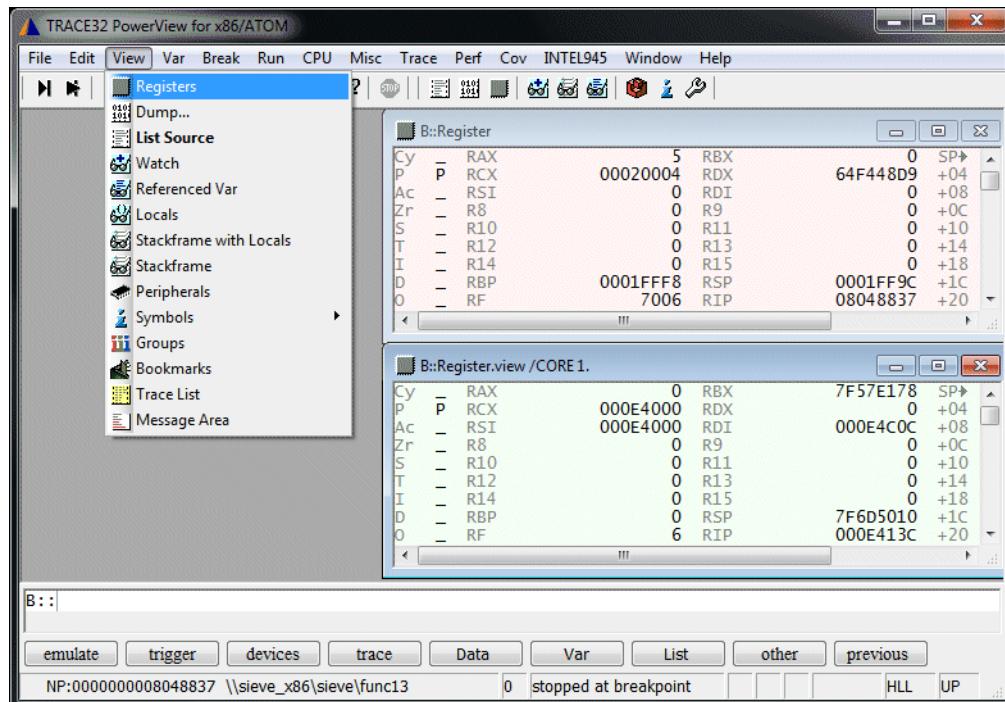
An HLL single step is performed on the selected core. All other cores are started and will stop, when this HLL single step is done.



<b>Mode.HLL</b>	Select High-level language mode for debugging and perform a single step.
<b>Step.HLL</b>	Perform an HLL single step.
<b>SETUP.StepWithinTask</b> 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
```

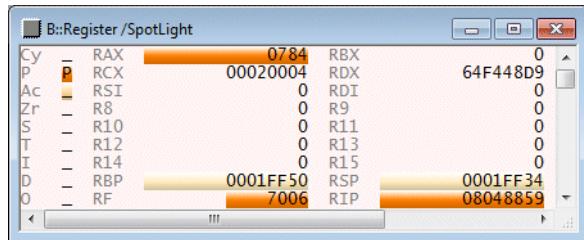
# Colored Display of Changed Registers

Register.view /SpotLight

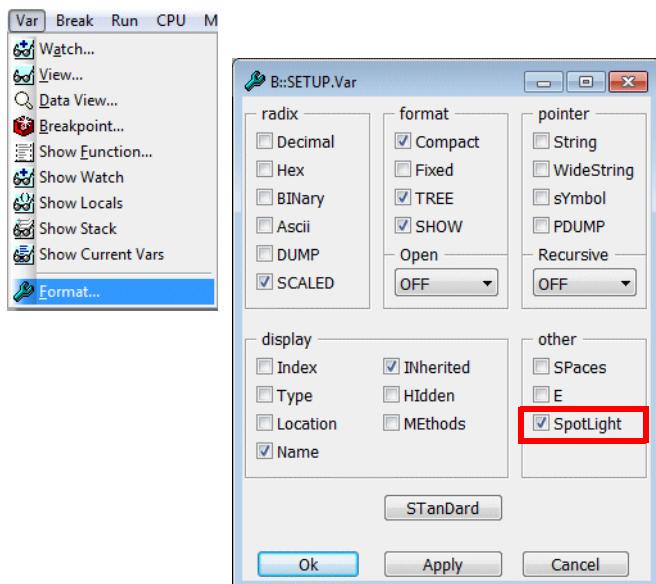
; The registers changed by the last  
; step are marked in dark red.

; The registers changed by the  
; step before the last step are  
; marked a little bit lighter.

; This works up to a level of 4.



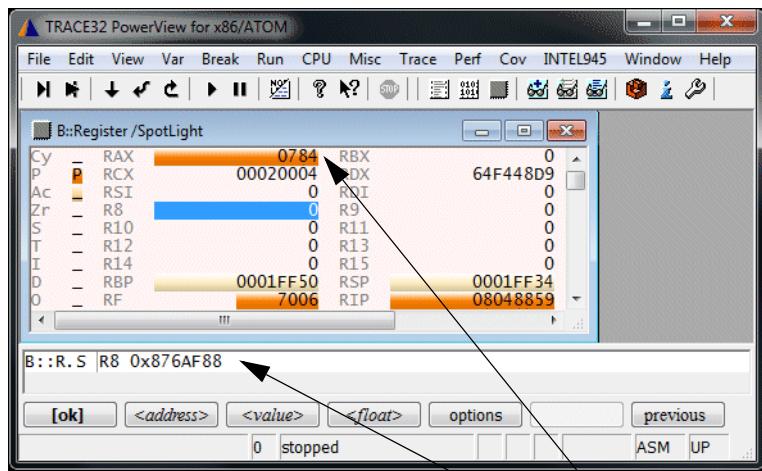
## Establish /SpotLight as default option



### 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>

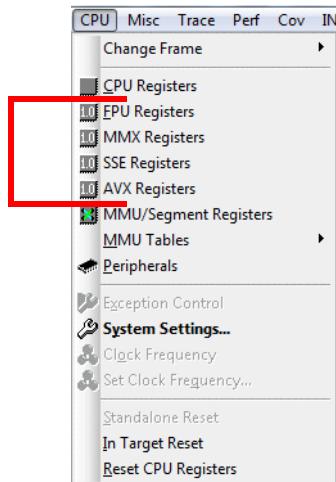
Modify core register of selected core

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

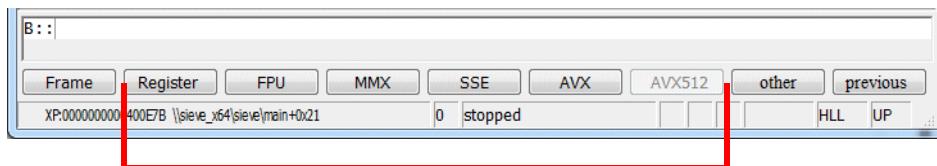
Modify core register of specified core

# 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.



# Special Function Register

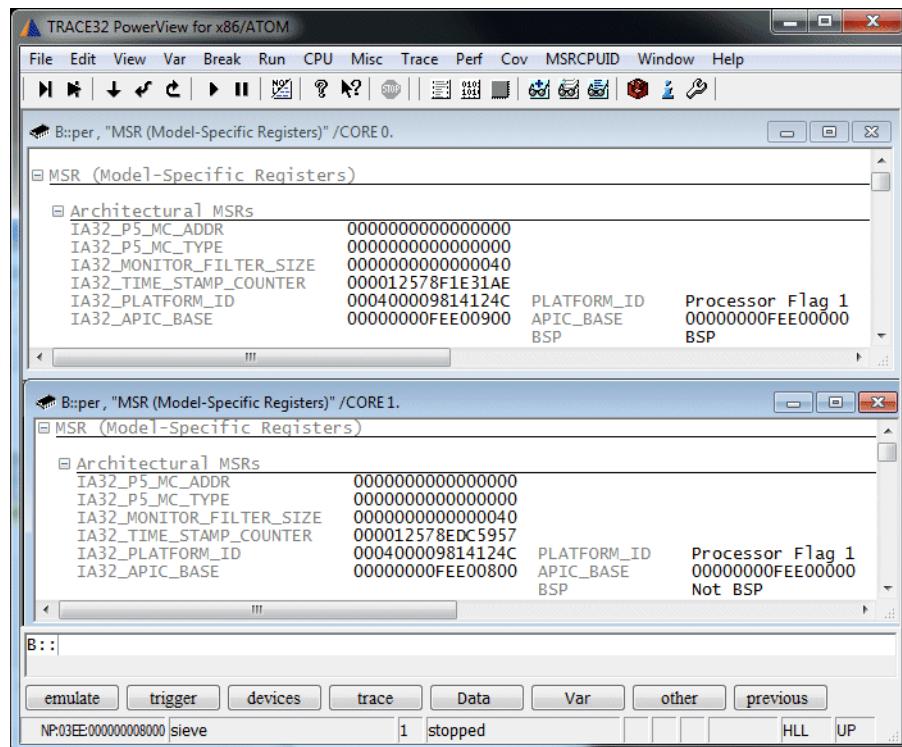
## Display the Special Function Registers

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.

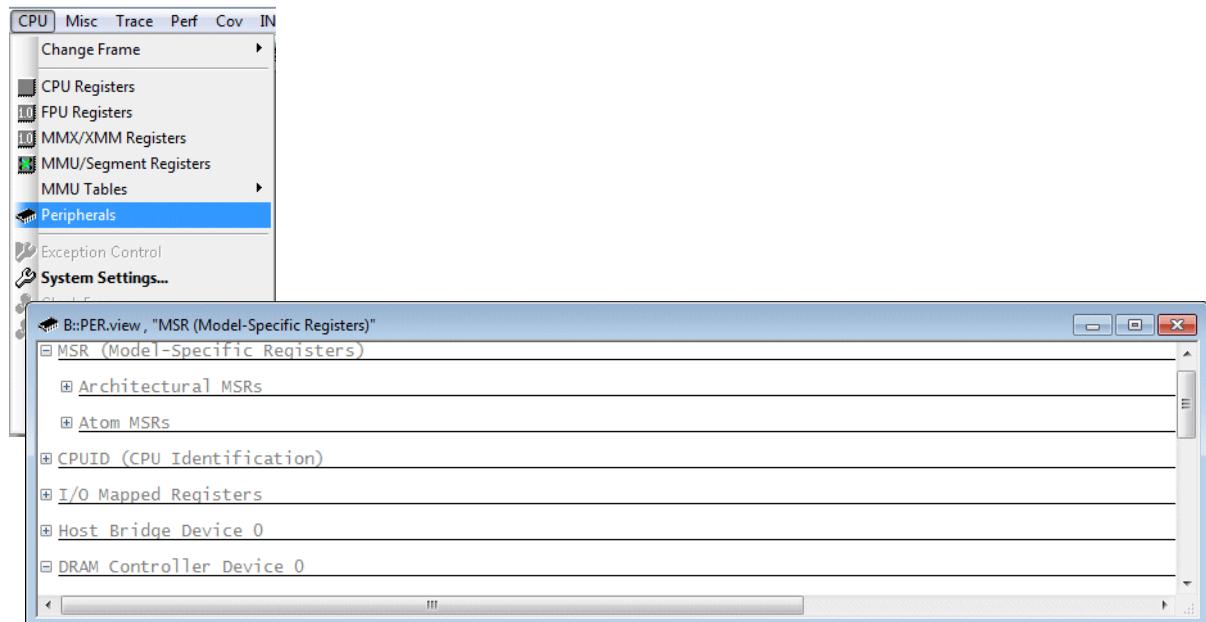
But not all configuration registers are common resources. Exceptions are the core-related registers e.g. CPUID registers, MSR registers ...



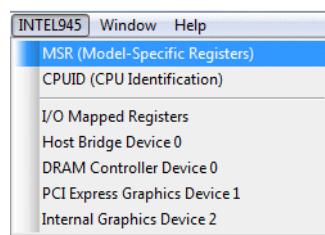
TRACE32 PowerView provides the **/CORE <n>** option in order to display details on core-related configuration registers:

```
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

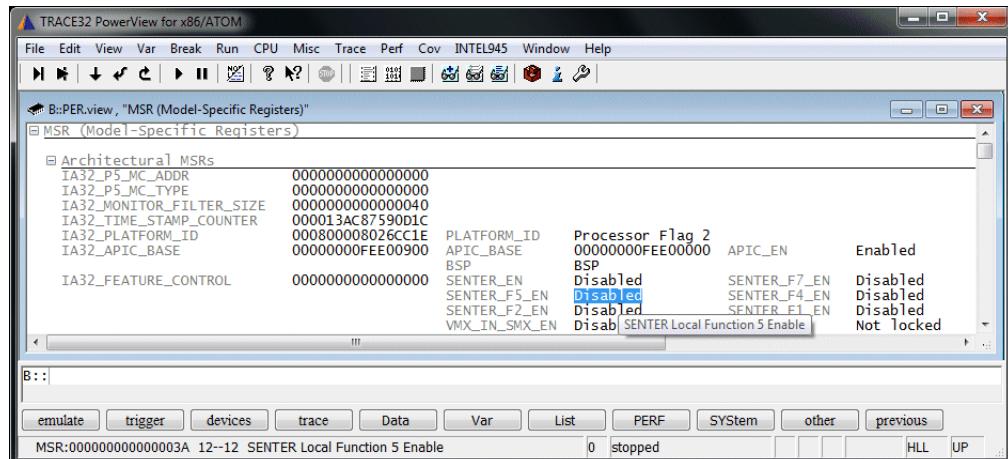
The screenshot shows the TRACE32 PowerView interface with a menu bar at the top. The 'INTEL945' tab is selected. The main window displays the 'B:PER.view, "MSR (Model-Specific Registers)"' content. The tree view under 'MSR (Model-Specific Registers)' includes the following nodes:

- Architectural MSRs

Below the tree view, a table lists the details of the Architectural MSRs:

IA32_P5_MC_ADDR	0000000000000000	PLATFORM_ID	Processor	Flag 2	
IA32_P5_MC_TYPE	0000000000000000	APIC_BASE	00000000FEE00000	APIC_EN	Enabled
IA32_MONITOR_FILTER_SIZE	0000000000000040	BSP			
IA32_TIME_STAMP_COUNTER	00000F578CB58358	SENTER_EN	Disabled	SENTER_F7_EN	Disabled
IA32_PLATFORM_ID	000800008026CC1E	SENTER_F5_EN	Disabled	SENTER_F4_EN	Disabled
IA32_APIC_BASE	00000000FEE000900	SENTER_F2_EN	Disabled	SENTER_F1_EN	Disabled
IA32_FEATURE_CONTROL	0000000000000000	VMX_IN_SMX_EN	Disabled	LOCK	Not locked

## Details about a Single Configuration Register



### 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.

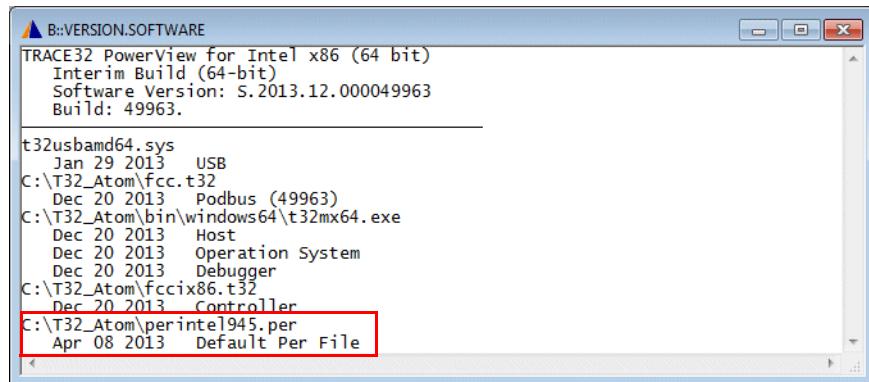
# The PER Definition File

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

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

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

## VERSION.SOFTWARE



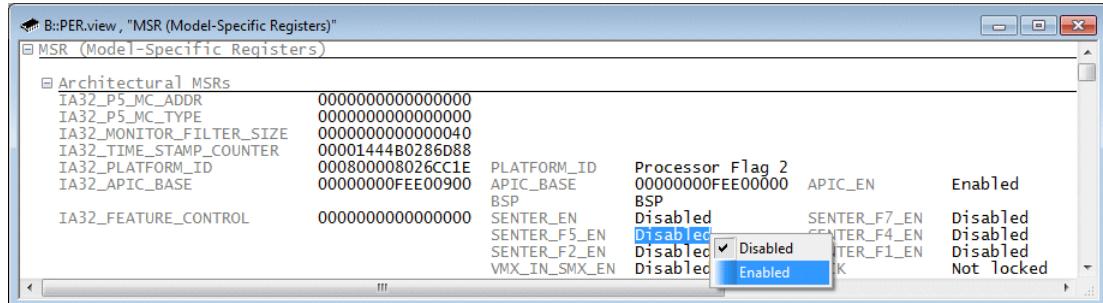
## PER.view [<filename>] [<tree-search-item>] Display the configuration registers/on-chip peripherals

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

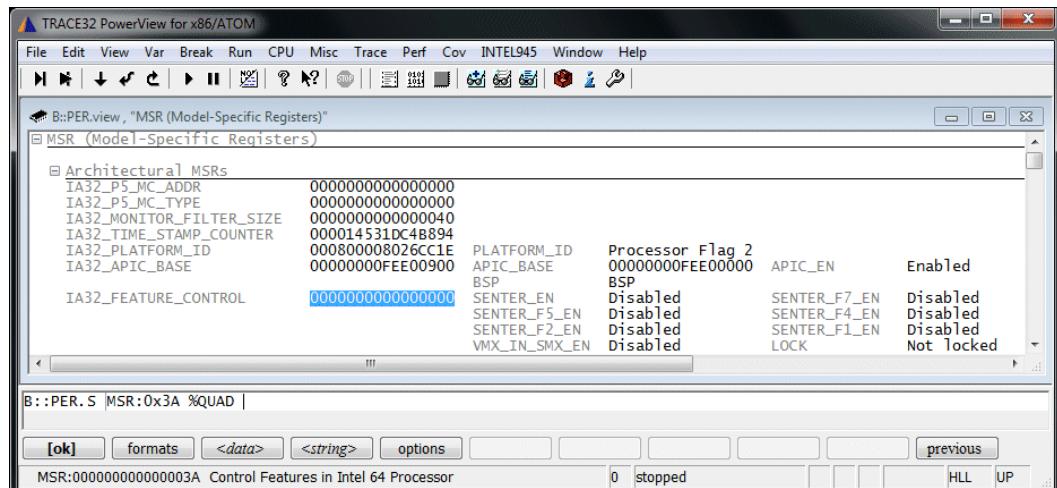
# Modify a Special Function Register

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

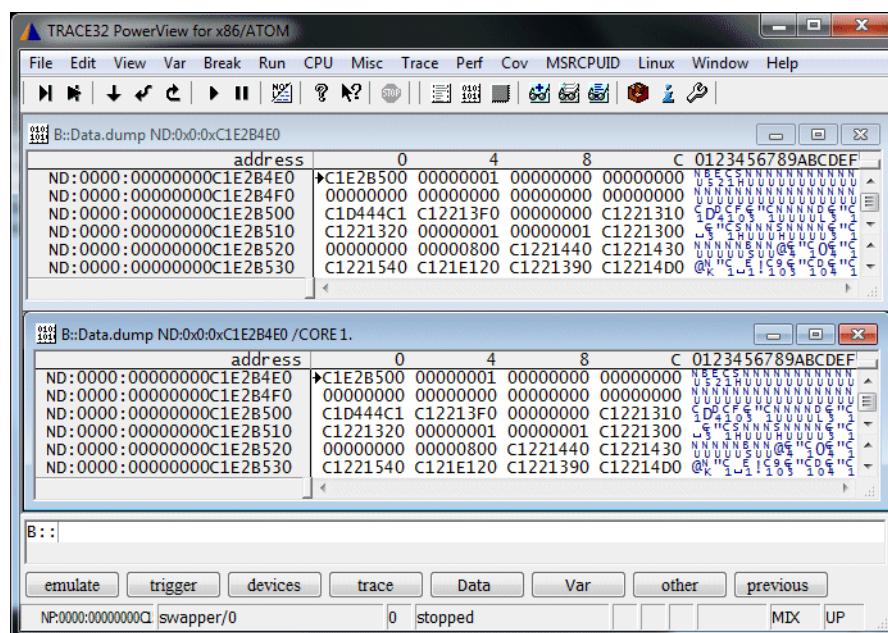
This training section introduces the most often used methods to display and modify memory:

- 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.

The screenshot shows the TRACE32 PowerView debugger interface. It has two main windows: **B::List** and **B::List / CORE1**. Both windows display assembly code and source code for the file **bitops.h**. The assembly code includes instructions like `mov eax, [edx+0x8]`, `test al, 0x8`, and `jne 0xC15565B5`. The source code includes conditional blocks like `#if defined(CONFIG_INTEL_ATOM_MDFLD_POWER) || \ defined(CONFIG_INTEL_ATOM_CLV_POWER)`. The bottom of the interface shows various buttons like **emulate**, **trigger**, **devices**, **trace**, **Data**, **Var**, **List**, **PERF**, **SYStem**, **other**, and **previous**. The bottom status bar shows the address **NP:0000:0000000C1556571** and the state **0 stopped**.

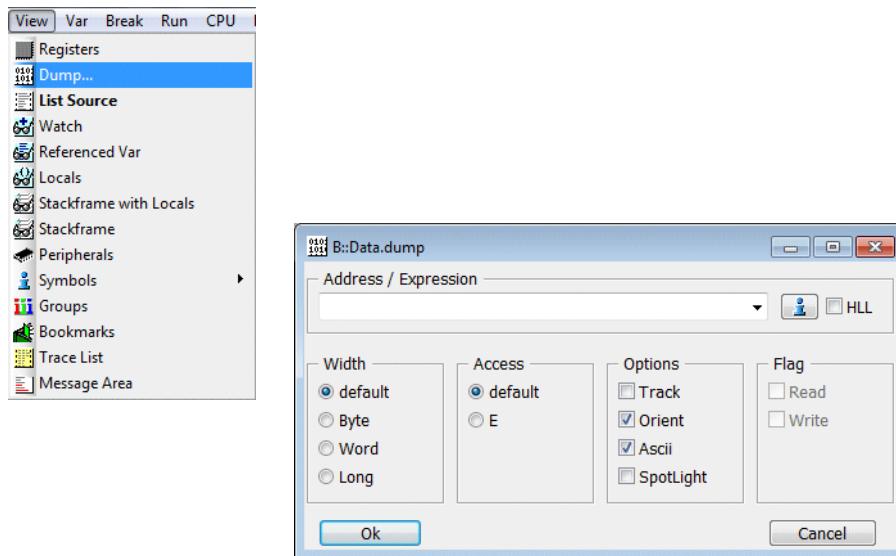
A so-called **access class** is always displayed together with a memory address. Examples:

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

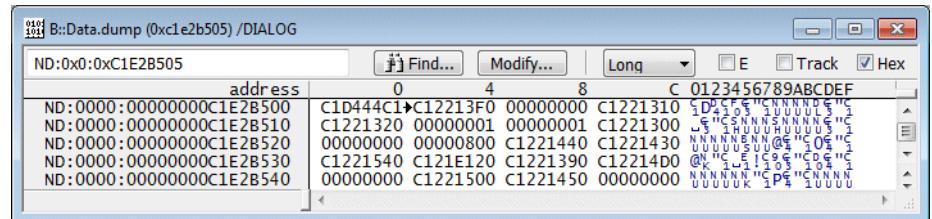
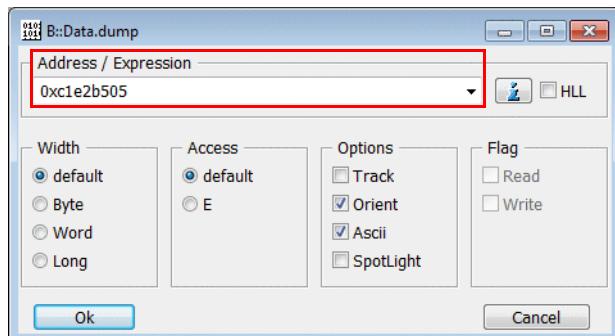
For a list of all access classes provided for the Intel® x86/x64 architecture refer to [“Intel® x86/x64 Debugger”](#) (debugger\_x86.pdf).

# The Data.dump Window

## Basics



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



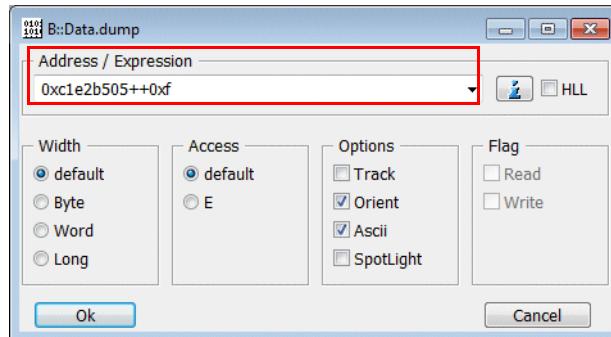
Please be aware, that TRACE32 permanently updates all windows. The default update rate is 10 times per second.

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

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

### Conventions for address ranges:

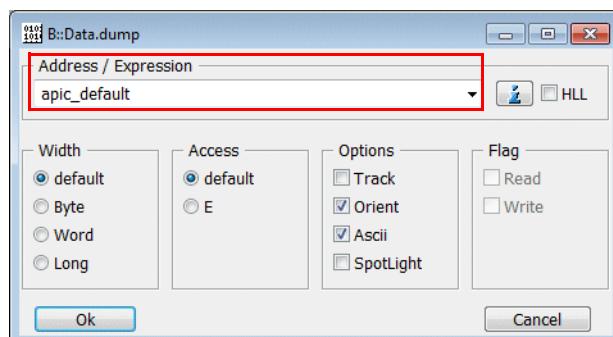
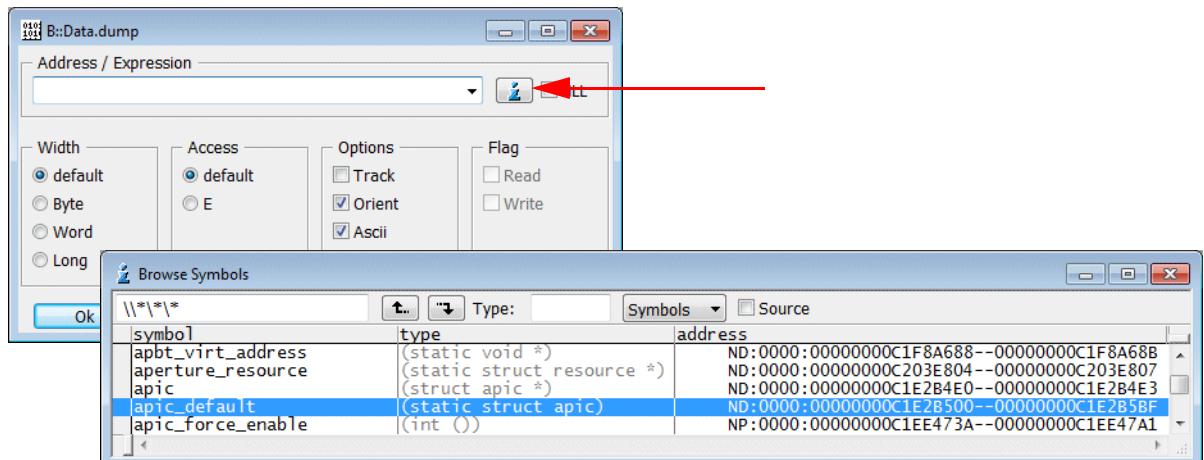
- <start address>--<end\_address>
- <start address>..<end\_address>
- <start address>++<offset\_in\_byte>



B::Data.dump (0xc1e2b505++0xf) /DIALOG	
address	0 4 8 C 0123456789ABCDEF
ND:0000:00000000C1E2B500	→C12213?? 00000000 C1221310
ND:0000:00000000C1E2B510	C1221320 ??????01
ND:0000:00000000C1E2B520	„C1H

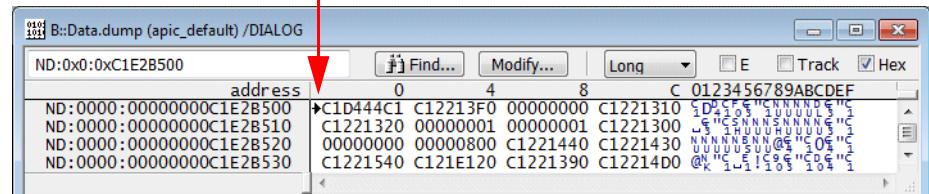
## Use a Symbol to Specify the Start Address for the Data.dump Window

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



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

A small arrow indicates  
the specified dump address.



```
Data.dump 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

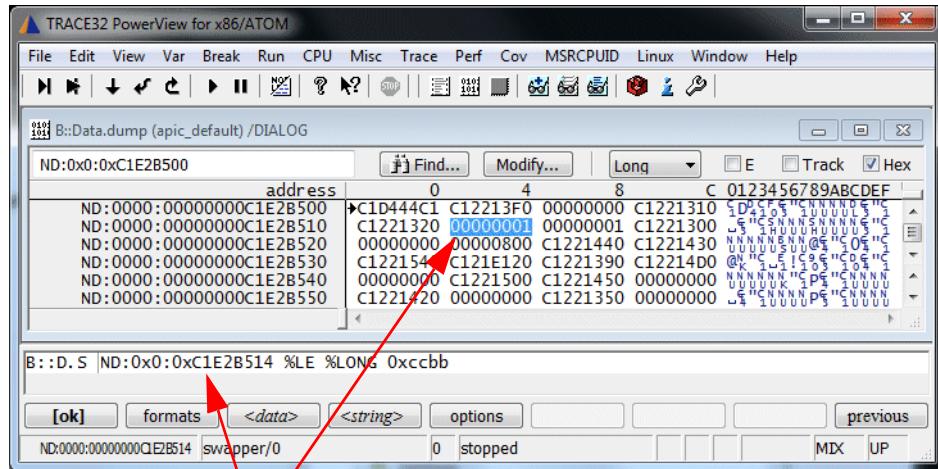
Data.dump 0xC1E2B4E0++0x7 ; Display a hex dump of the
; specified address range

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
```

# Modify the Memory Contents



By a left mouse double-click to the memory contents

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

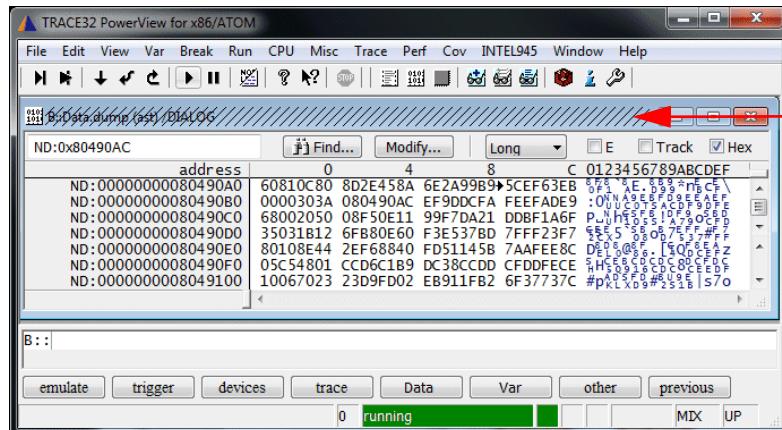
**Data.Set <address>|<range> [%<format>] [/<option>]**

Modify the memory contents

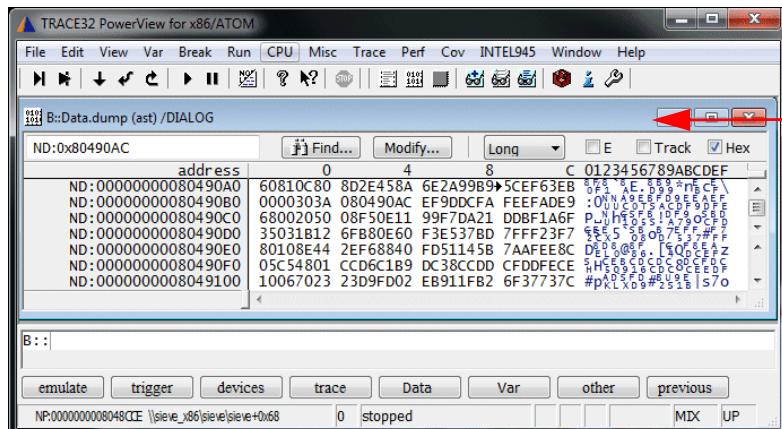
Data.Set 0xC1E2B4E0 0x0000aaaa	; Write 0x0000aaaa to the logical ; address 0xC1E2B4E0 in the current ; address space
Data.Set 0xC1E2B4E0 %Long 0xaaaa	; Write 0xaaaa as a 32 bit value to ; the address 0xC1E2B4E0, add the ; leading zeros automatically
Data.Set 0x6814 %LE %Long 0xaaaa	; Write 0xaaaa as 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

# Run-time Memory Access

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

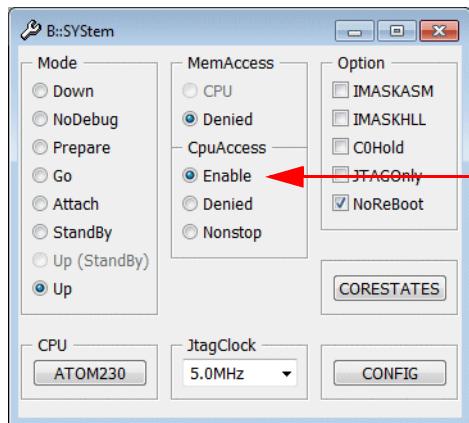


A hatched window frame indicates that the information display is frozen because the cores are executing the program.



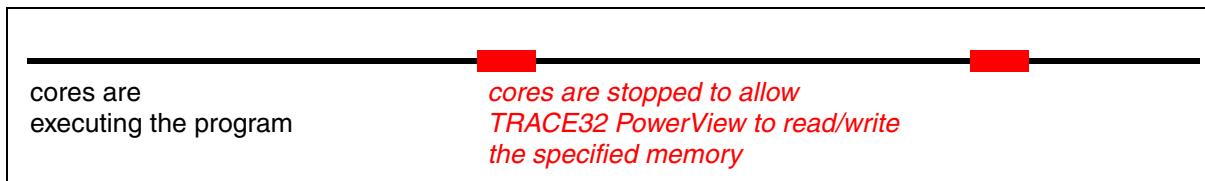
The plain window frame indicates that the information is updated, because the program execution is stopped.

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.

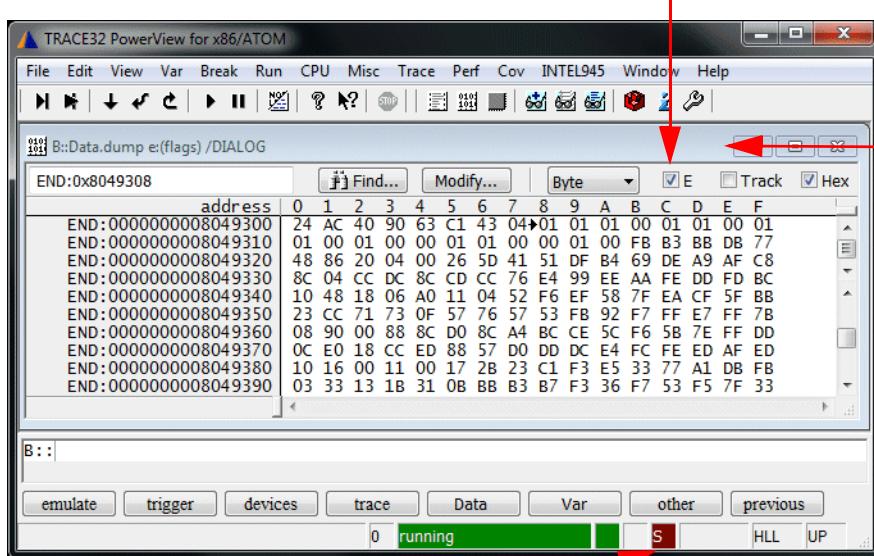


**CpuAccess Enable** allows an intrusive run-time memory access

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.



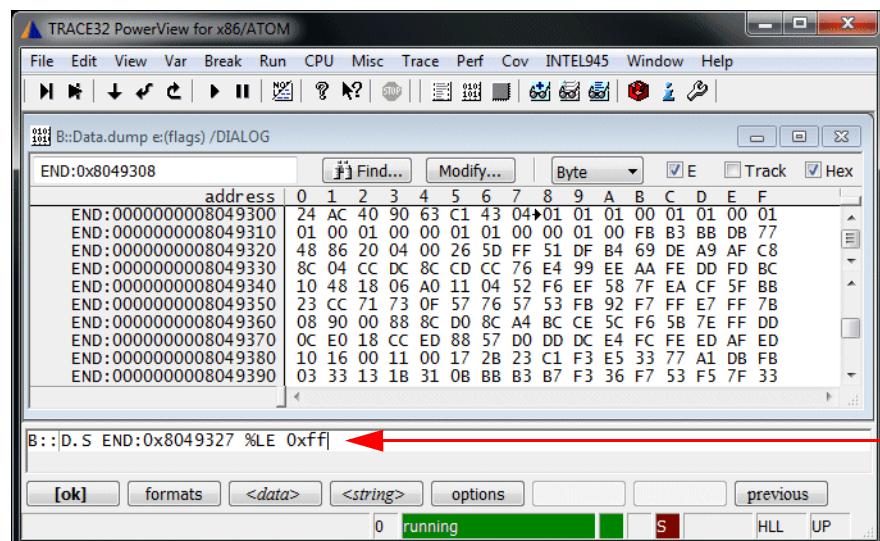
Enable the **E** check box to switch the run-time memory access to ON



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

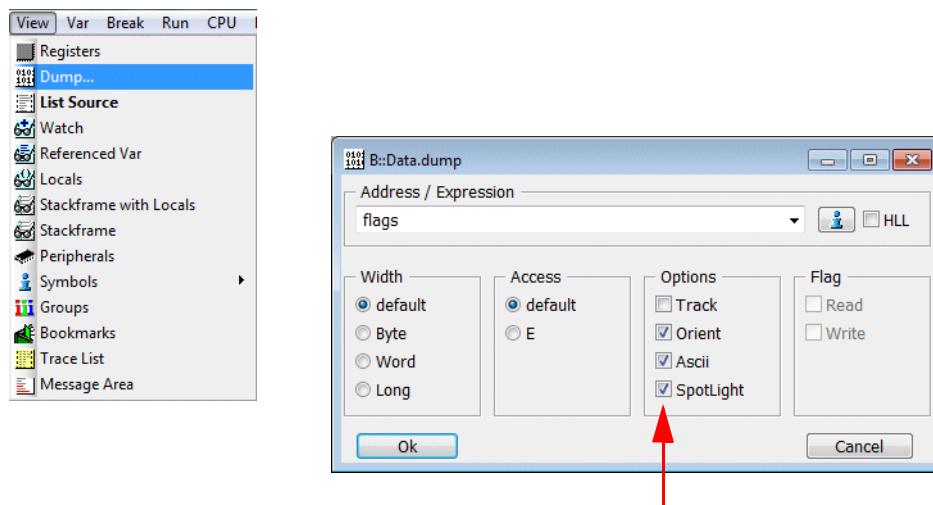
; ...

Go ; Start program execution

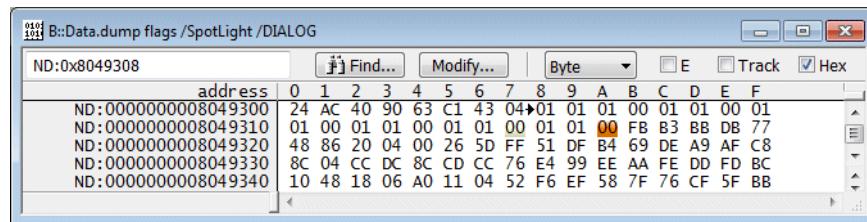
Data.dump E:0x6814 ; Display a hex dump starting at
; address 0x6814 via an intrusive
; run-time memory access

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.

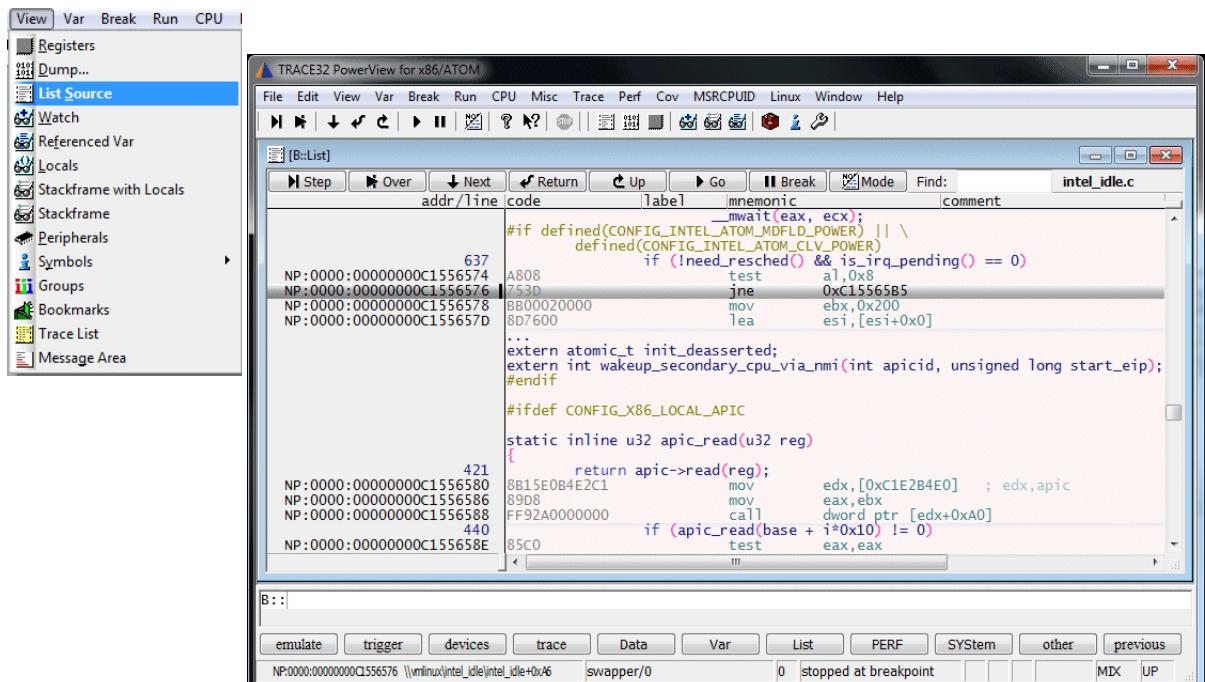


```
Data.dump flags /SpotLight
```

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

# The List Window

## Displays the Source Listing Around the PC



TRACE32 PowerView for x86/ATOM

File Edit 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:List

Step Over Next Return Up Go Break Mode Find: intel\_idle.c

addr/line code label mnemonic comment

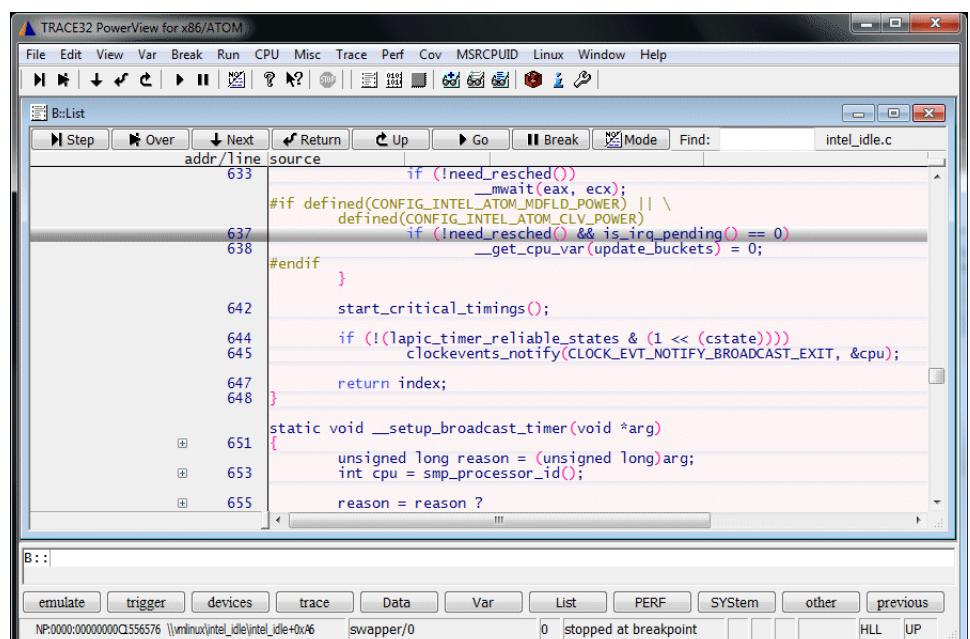
637 NP:0000:0000000C1556574 \_mwait(eax, ecx);  
#if defined(CONFIG\_INTEL\_ATOM\_MDFLD\_POWER) || \  
defined(CONFIG\_INTEL\_ATOM\_CLV\_POWER)  
if (!need\_resched() && is\_irq\_pending() == 0)  
8A08 test al,0x8  
753D jne 0x15565B5  
NP:0000:0000000C1556576 BB00020000 mov ebx,0x200  
NP:0000:0000000C1556578 8D7600 lea esi,[esi+0x0]  
...  
extern atomic\_t init\_deasserted;  
extern int wakeup\_secondary\_cpu\_via\_nmi(int apicid, unsigned long start\_eip);  
#endif  
#ifdef CONFIG\_X86\_LOCAL\_APIC  
static inline u32 apic\_read(u32 reg)  
{  
 return apic->read(reg);  
8B15E0B4E2C1 mov edx,[0xC1E2B4E0] ; edx,apic  
89D8 mov eax,ebx  
FF92A0000000 call dword ptr [edx+0xA0]  
440 NP:0000:0000000C155658E 85C0 if (apic\_read(base + i\*0x10) != 0)  
test eax,eax  
!!!

B::

emulate trigger devices trace Data Var List PERF SYSTEM other previous

NP:0000:0000000C1556576 \\\linux\intel\_idle\intel\_idle+0x46 swapper/0 0 stopped at breakpoint MIX UP

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



TRACE32 PowerView for x86/ATOM

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

B:List

Step Over Next Return Up Go Break Mode Find: intel\_idle.c

addr/line source

633 if (!need\_resched())  
#if defined(CONFIG\_INTEL\_ATOM\_MDFLD\_POWER) || \  
defined(CONFIG\_INTEL\_ATOM\_CLV\_POWER)  
if (!need\_resched() && is\_irq\_pending() == 0)  
 \_\_get\_cpu\_var(update\_buckets) = 0;  
#endif  
637  
638  
642 start\_critical\_timings();  
644 if (!(apic\_timer\_reliable\_states & (1 << (cstate))))  
 clockevents\_notify(CLOCK\_EVT\_NOTIFY\_BROADCAST\_EXIT, &cpu);  
645  
647 return index;  
648  
651 static void \_\_setup\_broadcast\_timer(void \*arg)  
652 {  
653 unsigned long reason = (unsigned long)arg;  
654 int cpu = smp\_processor\_id();  
655 reason = reason ?

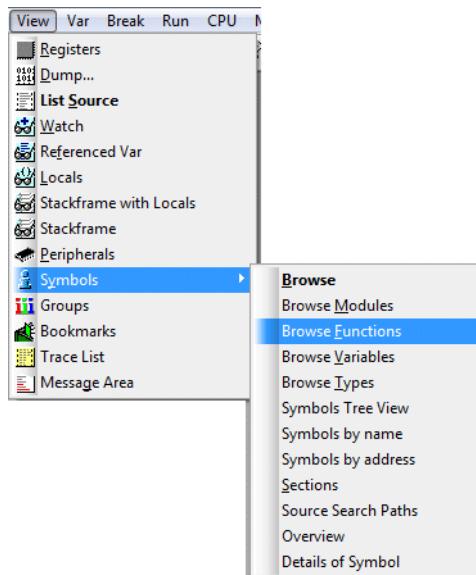
B::

emulate trigger devices trace Data Var List PERF SYSTEM other previous

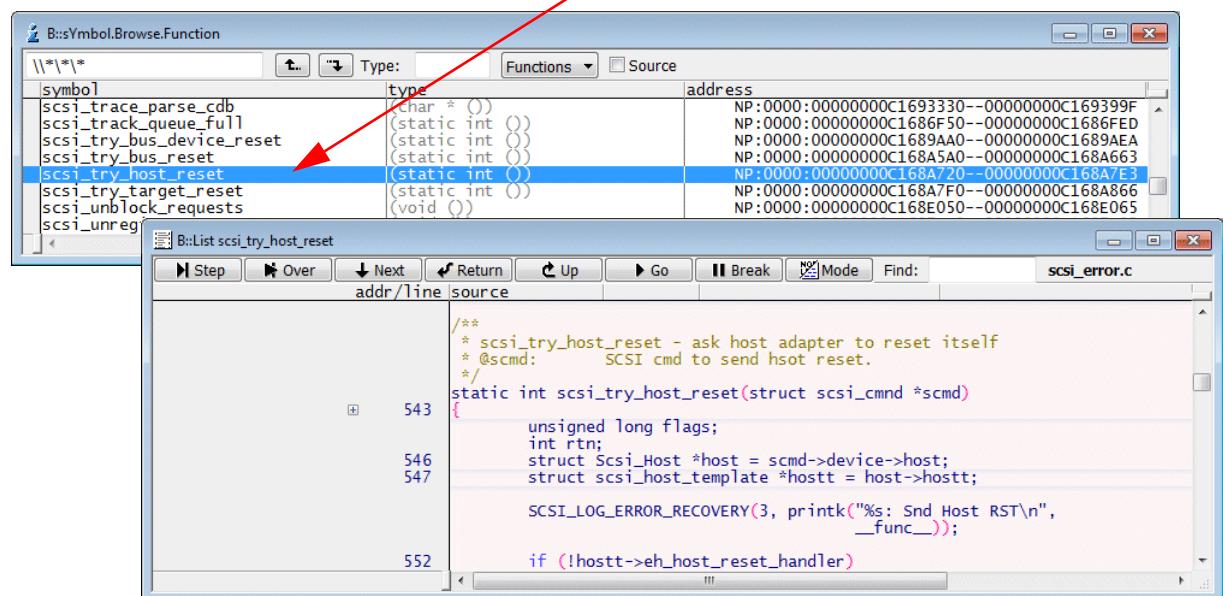
NP:0000:0000000C1556576 \\\linux\intel\_idle\intel\_idle+0x46 swapper/0 0 stopped at breakpoint HLL UP

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

# Displays the Source Listing of a Selected Function



Select the function you want to display



**List** [<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
```

# Breakpoints

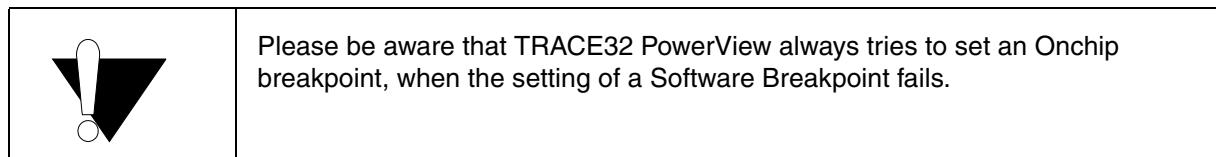
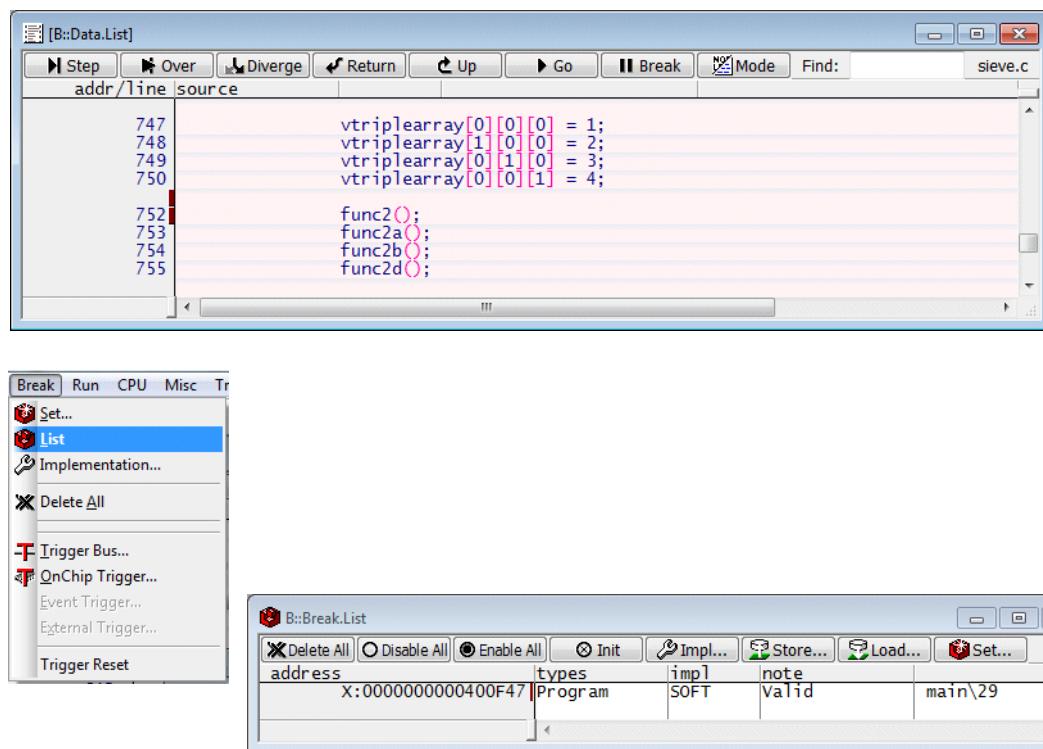
## Breakpoint Implementations

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.

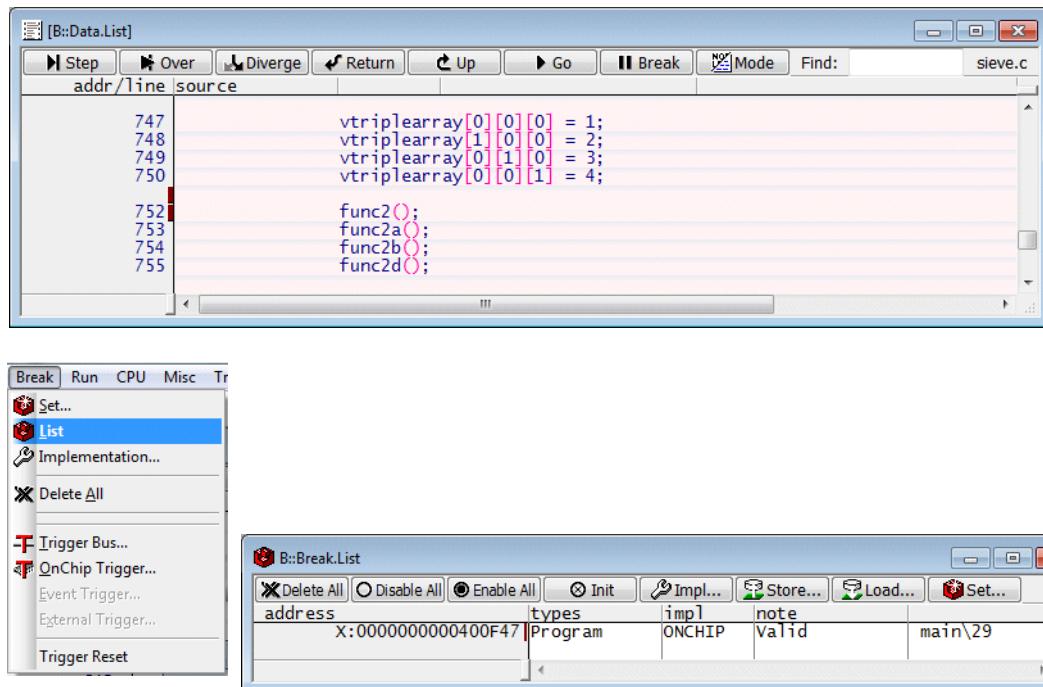


# Onchip Breakpoints in NOR Flash (Program)

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
```

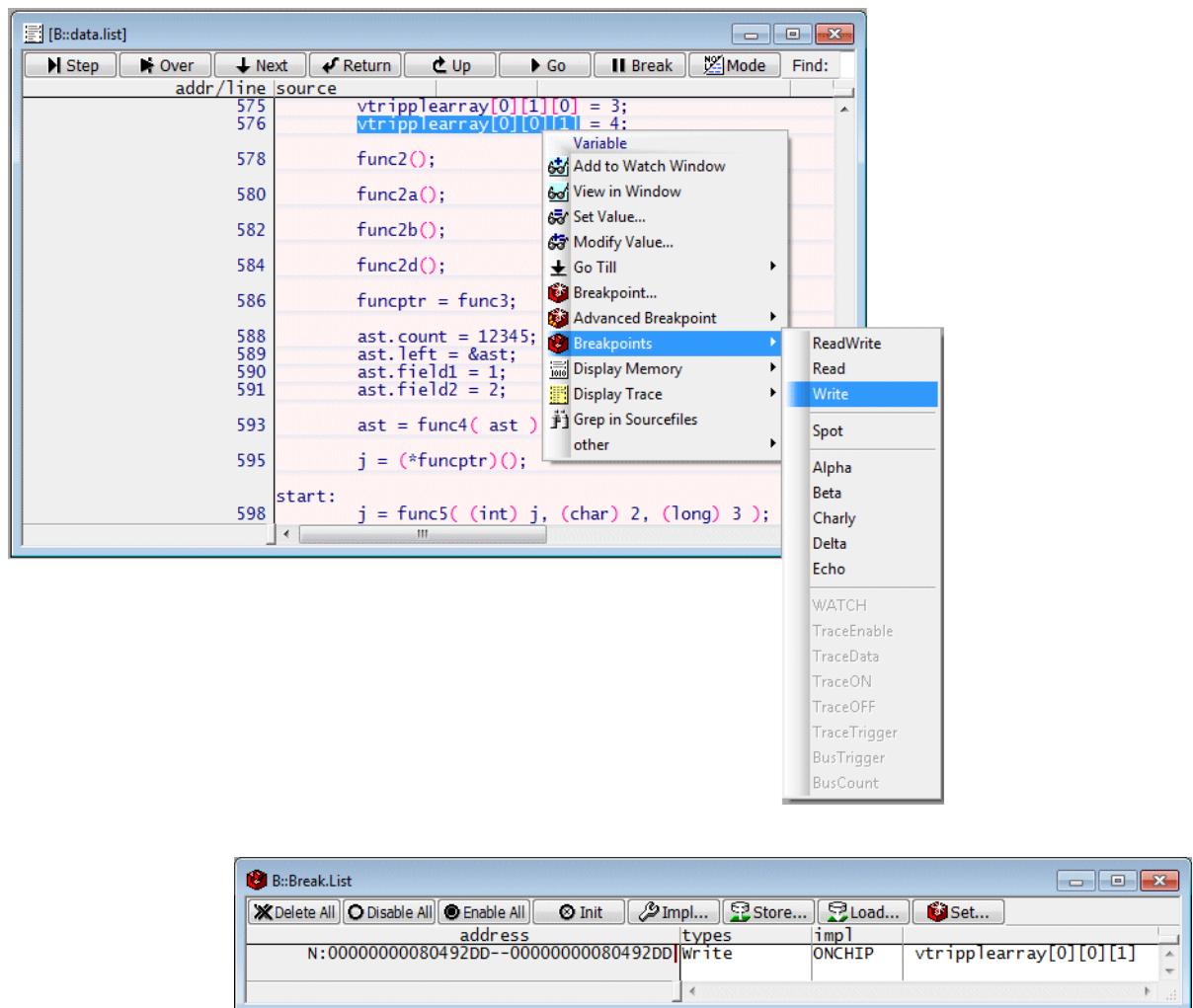
The onchip breakpoints of Intel® x86/x64 cores store only the logical address, but not the memory space ID. As a result an identical logical address within another process can also result in a breakpoint hit.

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 (Read/Write)

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

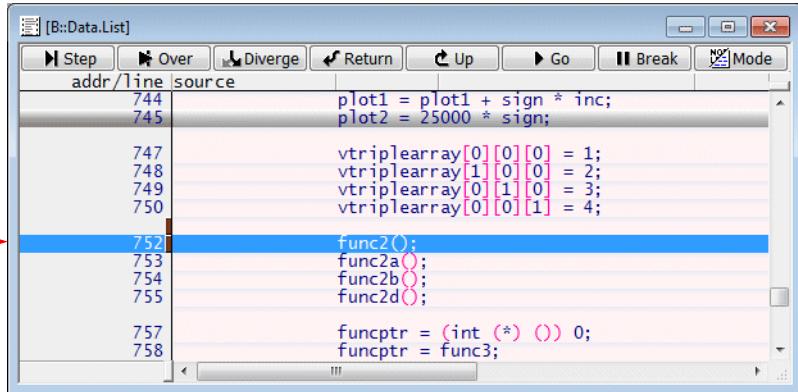
---

TRACE32 PowerView provides the following breakpoint types for standard debugging.

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

# Program Breakpoints

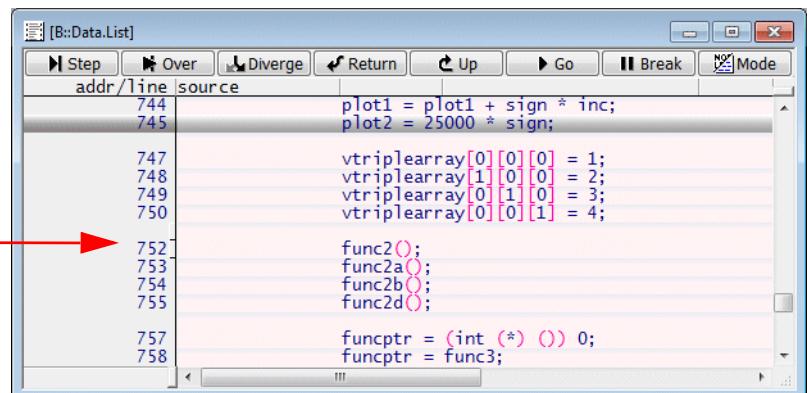
Set a Program breakpoint by a left mouse double-click to the instruction



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

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

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



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

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

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

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

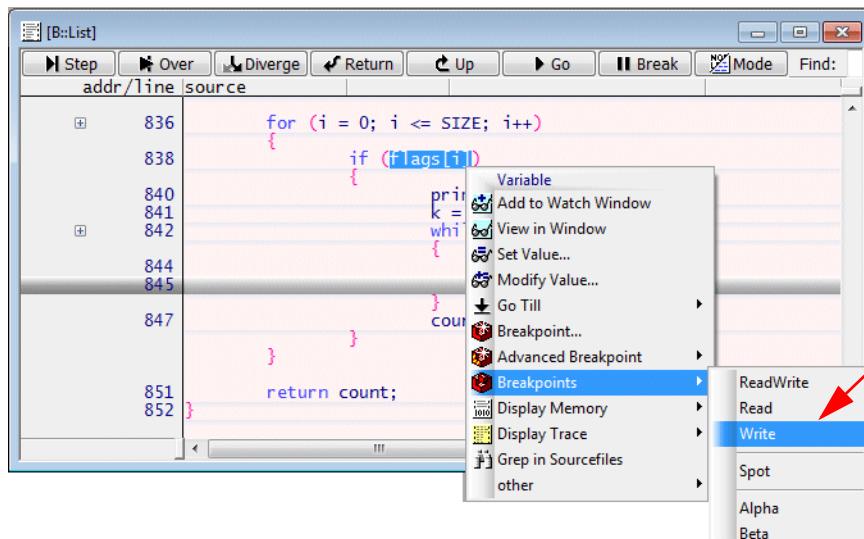
Break.Set func1+0x1c /Program ; set a Program breakpoint to the
                           ; instruction at address
                           ; func1 plus 28 bytes

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

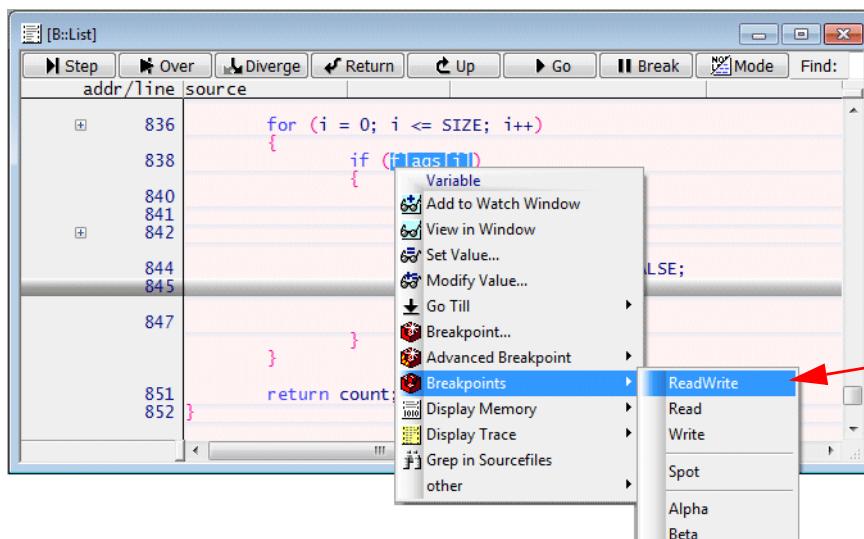
Break.Set func17 /Program /DISable ; set a Program breakpoint to the
                                    ; entry of function func17
                                    ; disable Program breakpoint

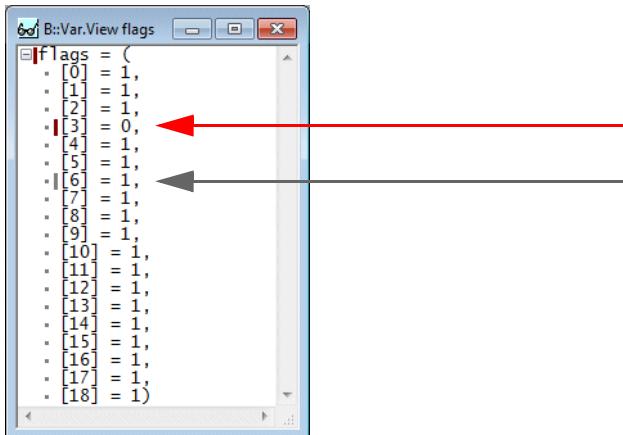
Break.List ; list all breakpoints
```

## Read/Write Breakpoints



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.





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.

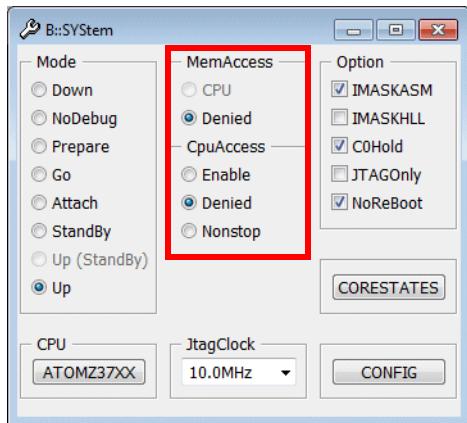
**Break.Set <address> | <range> /Write | /ReadWrite [/DISable]**

; 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.

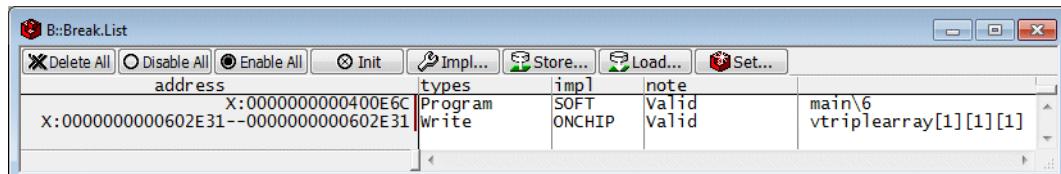
**TRACE32 has the following standard behavior:** The breakpoint list in TRACE32 PowerView is not deleted, when TRACE32 detects a reset/power cycle. Thus TRACE32 can set all listed breakpoints again when the program execution is stopped and restarted.

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.

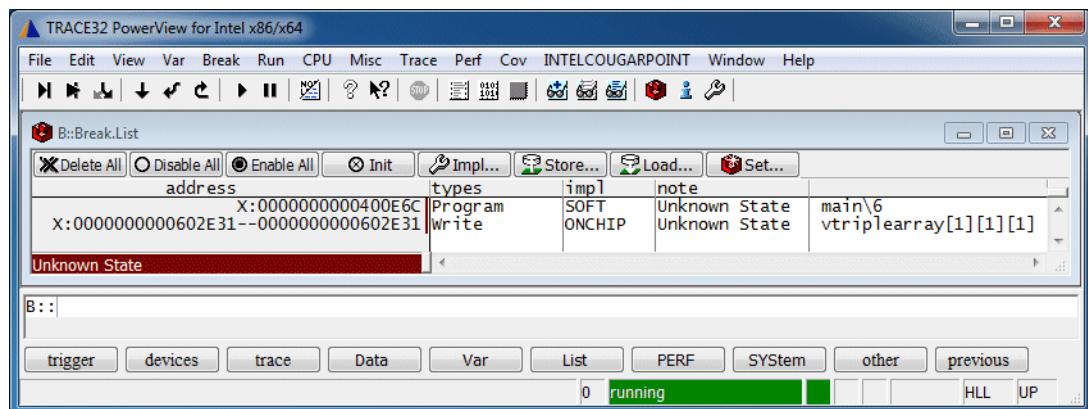
To indicate this loss, TRACE32 changes the state of the breakpoints from **Valid** to **Unknown State**. The state of the breakpoints is displayed in the **note** field of the breakpoint listing.

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

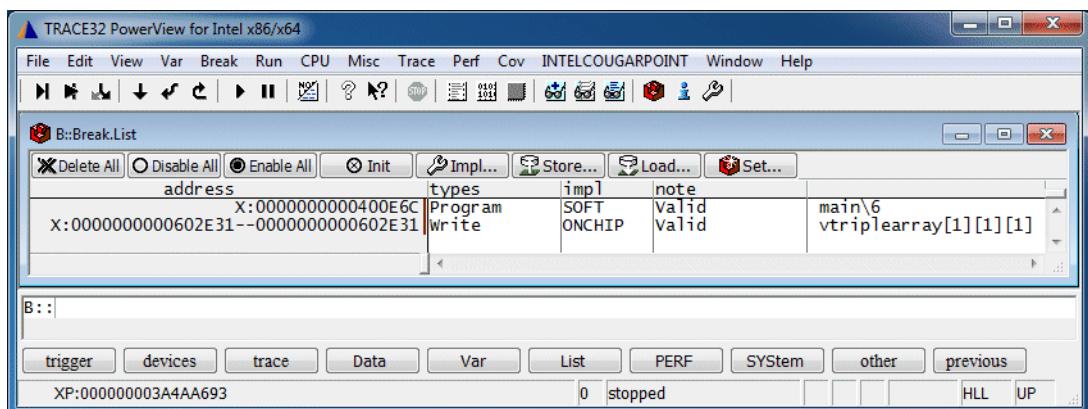
## Example:



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.

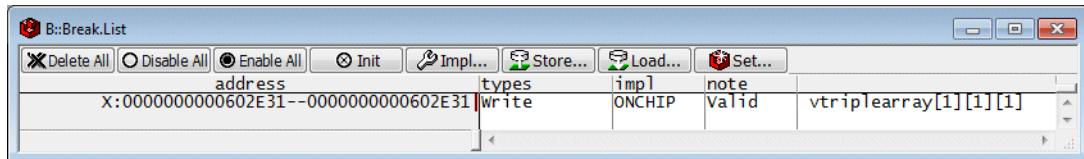


As soon as the program execution is stopped, all listed breakpoints become **Valid** again, because they will be set again when the program execution is restarted.



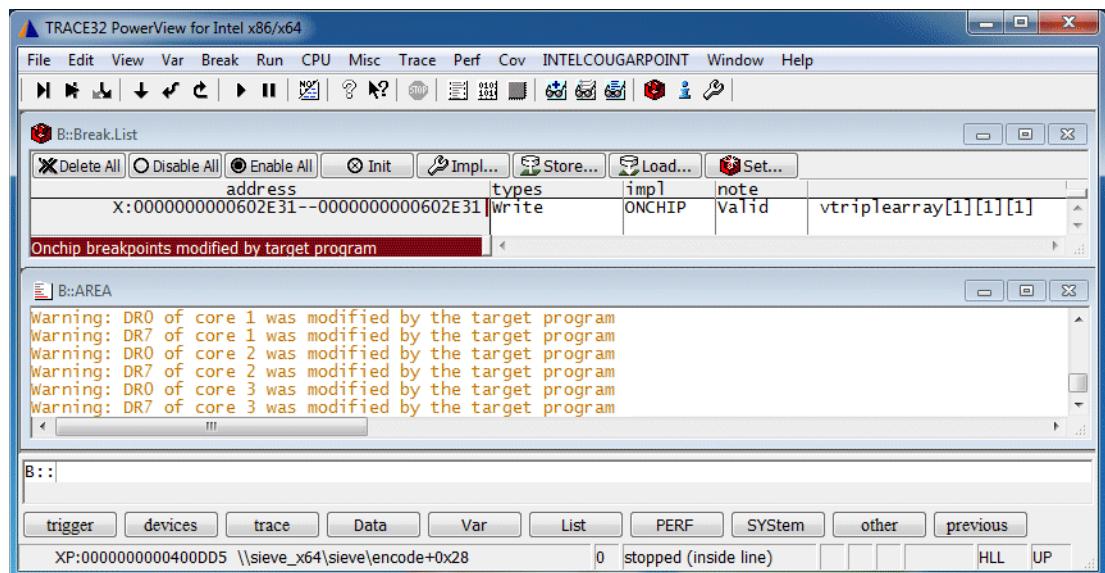
# Onchip Breakpoints Changed by Target Program

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

---

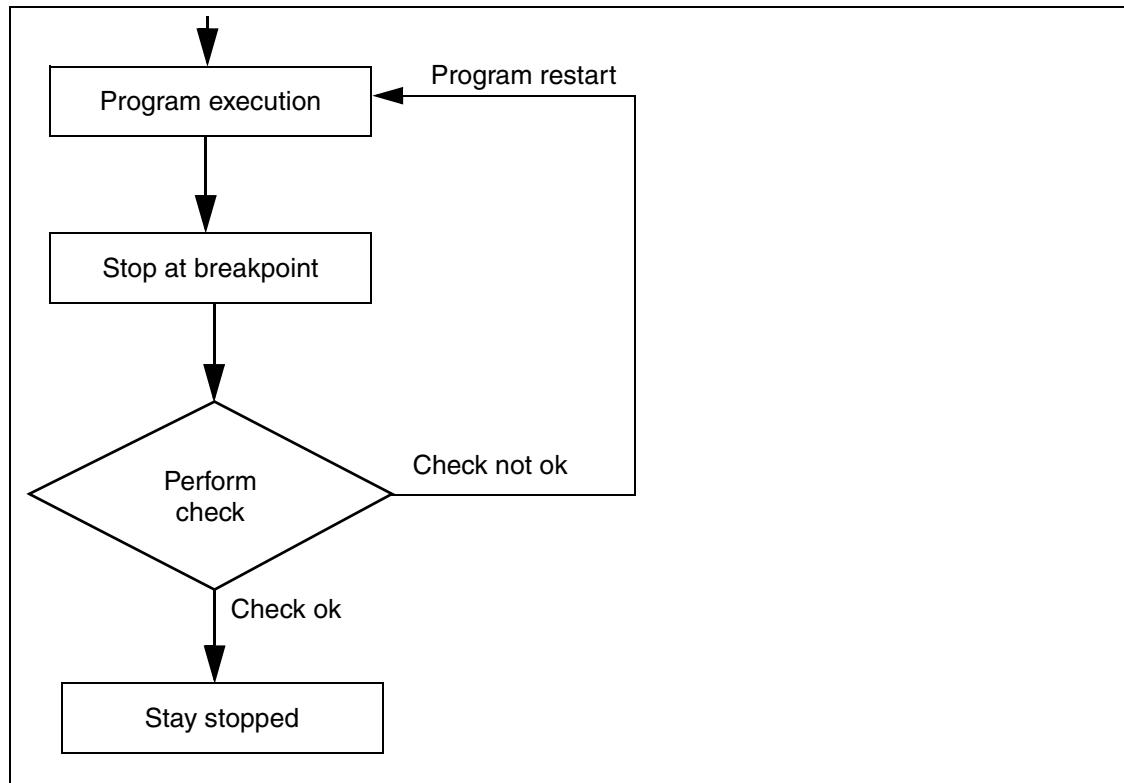
TRACE32 PowerView offers in addition to the basic breakpoints (Program/Read/Write) also complex breakpoints. Whenever possible these breakpoints are implemented as real-time breakpoints.

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

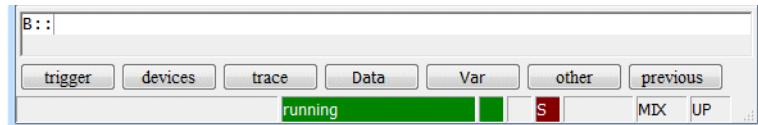
If the onchip breakpoint logic of a core does not provide the required features or if Software breakpoints are used, TRACE32 has to implement an intrusive breakpoint.

## 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.

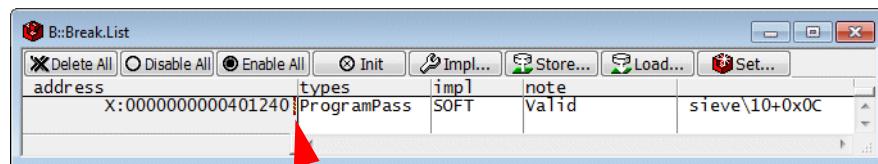
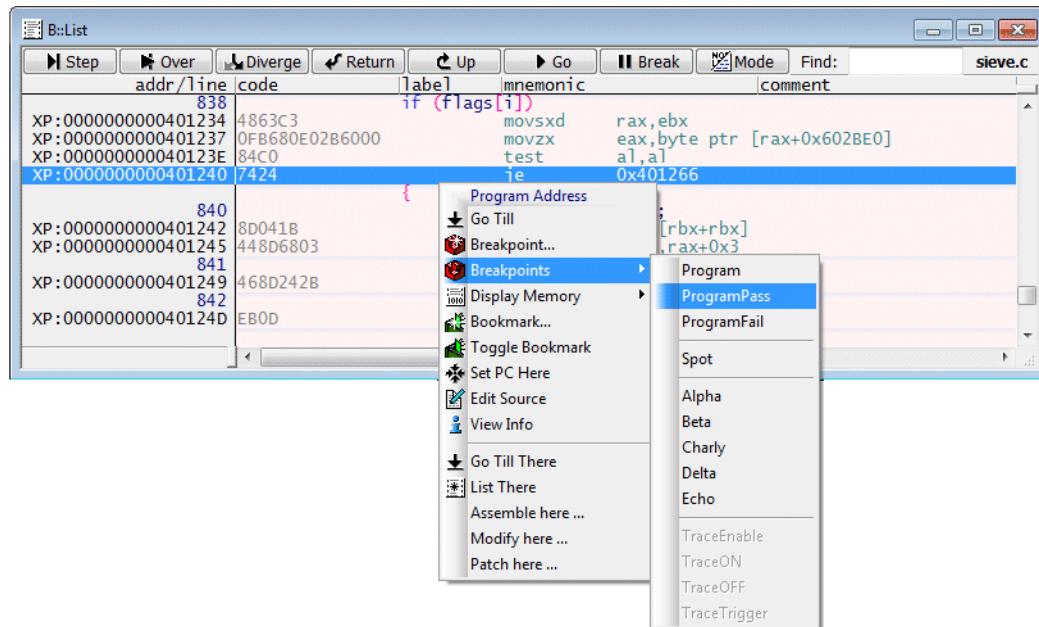
Intrusive breakpoints are marked by a special breakpoint indicator:



## ProgramPass/ProgramFail Breakpoints

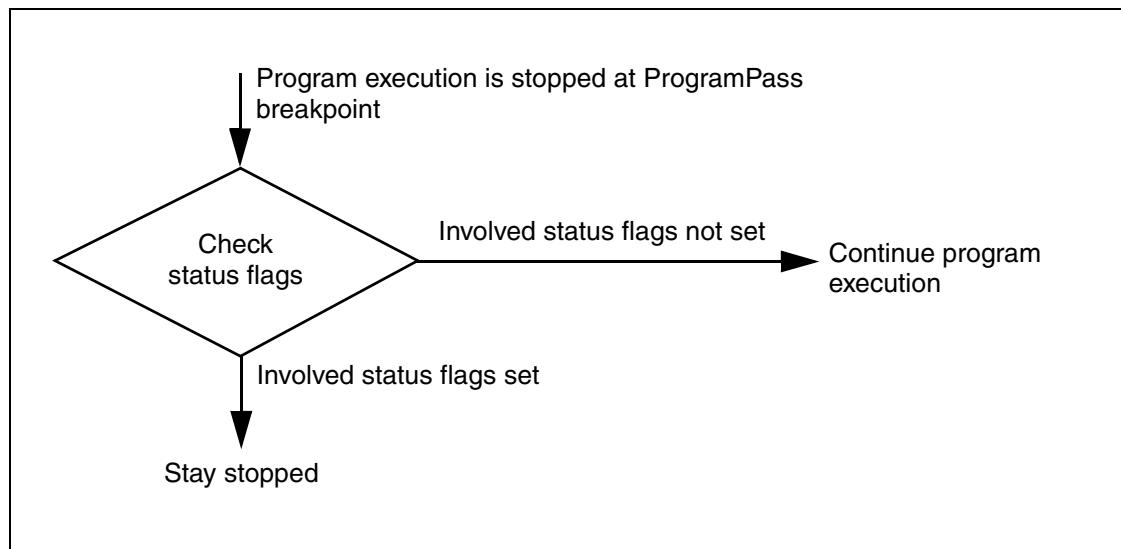
<b>ProgramPass</b>	If a breakpoint is set to a conditional instruction, the program execution is only stopped, if the condition is satisfied (pass).
<b>ProgramFail</b>	If a breakpoint is set to a conditional instruction, the program execution is only stopped, if the condition fails.

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

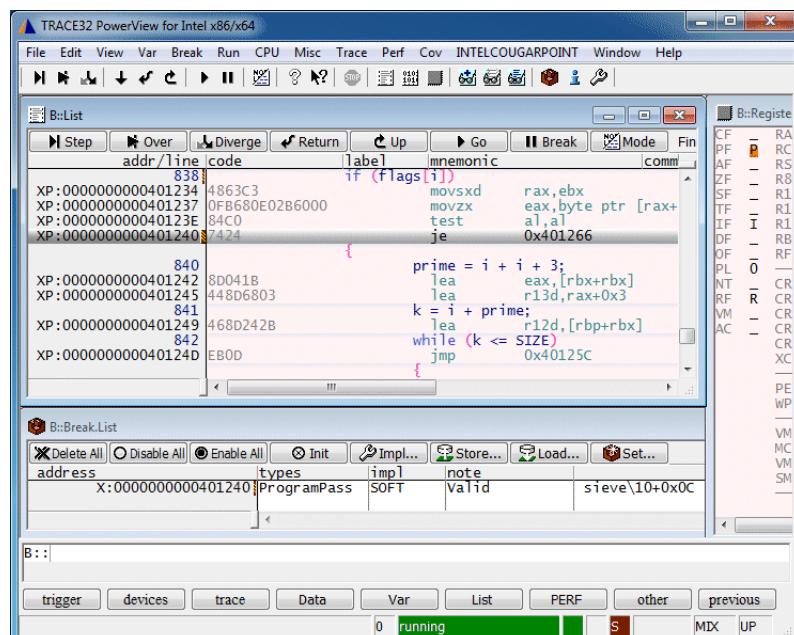


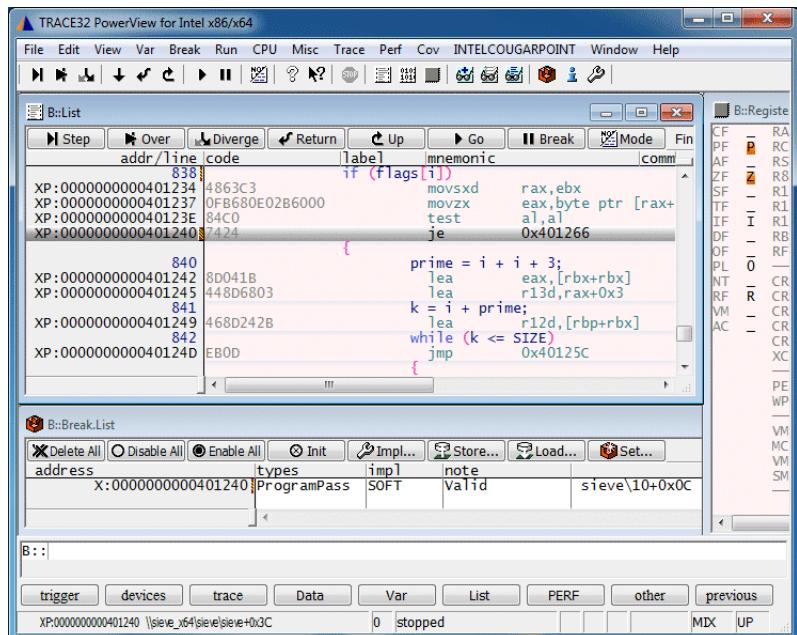
Intrusive breakpoints are marked with an intrusive breakpoint indicator

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.





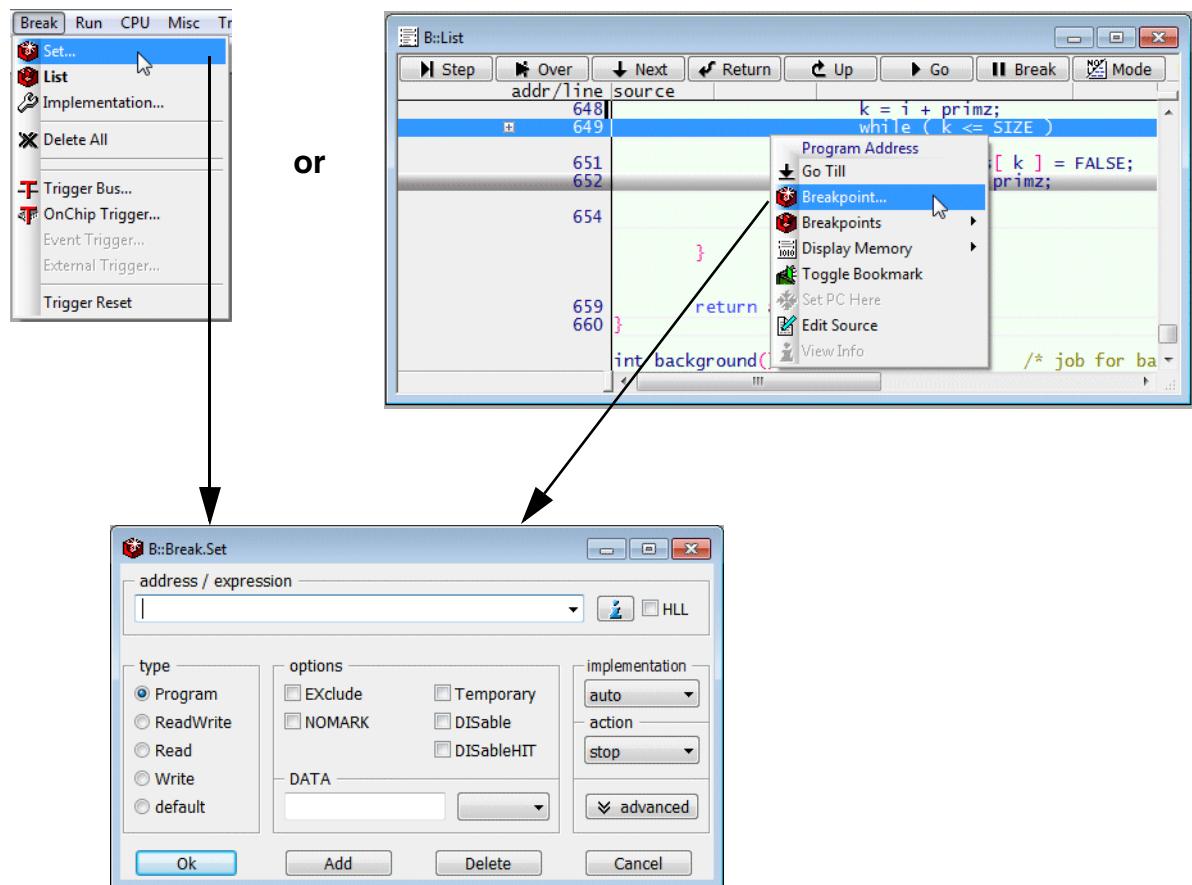
**Break.Set <address>|<range> /ProgramPass**

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

Break.Set 0x401240 /ProgramPass

# Break.Set Dialog Box

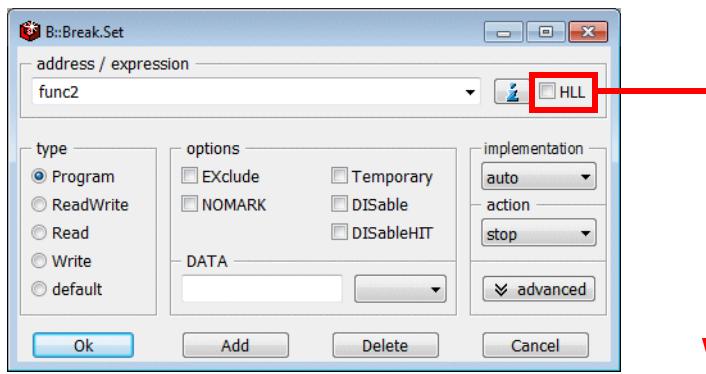
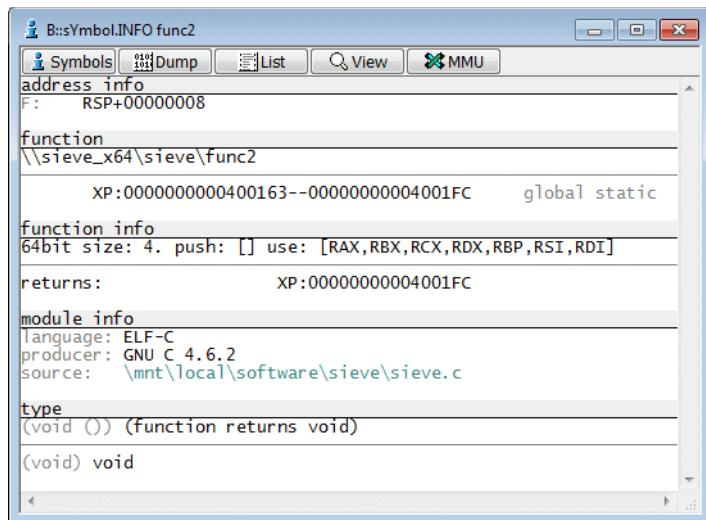
There are two standard ways to open a **Break.Set** dialog.



# The HLL Check Box

## Function Name/HLL Check Box OFF

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

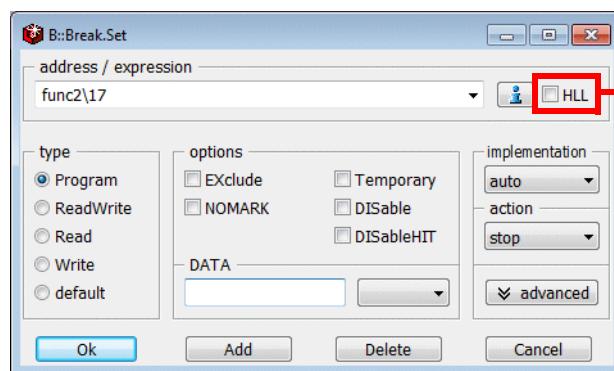
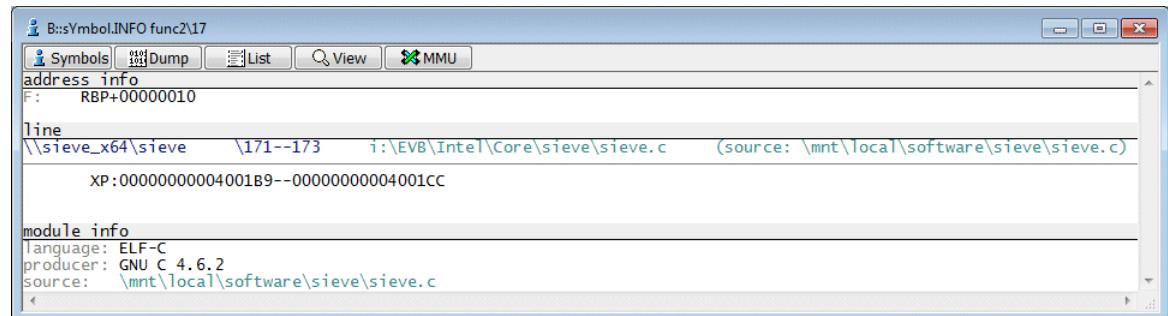


Program breakpoint is set to the function entry (first address of the function)

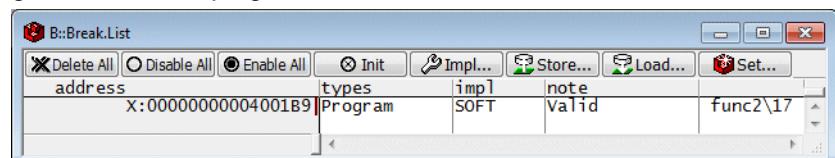
B::Break.List				
<input type="checkbox"/> Delete All	<input type="radio"/> Disable All	<input checked="" type="radio"/> Enable All	<input type="checkbox"/> Init	<input type="checkbox"/> Impl...
address	types	impl	note	
X:0000000000400163	Program	SOFT	Valid	func2

```
Break.Set func2
```

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

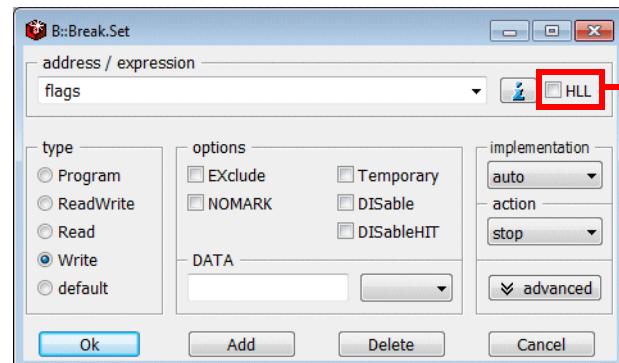
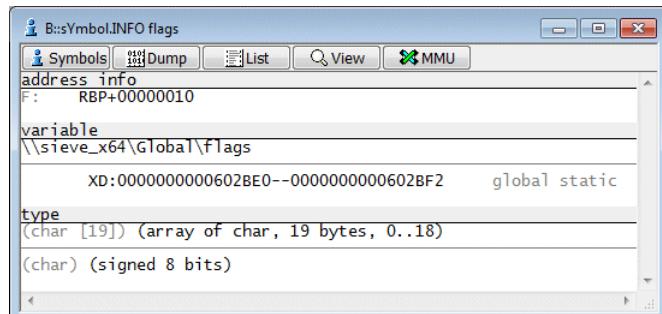


Program breakpoint is set to the first assembler instruction generated for the program line number

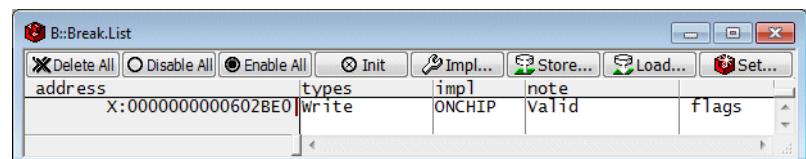


```
Break.Set func2\17
```

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

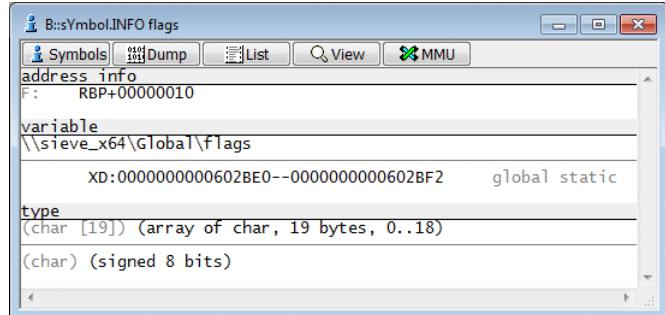


Selected breakpoint is set to the start address of the variable

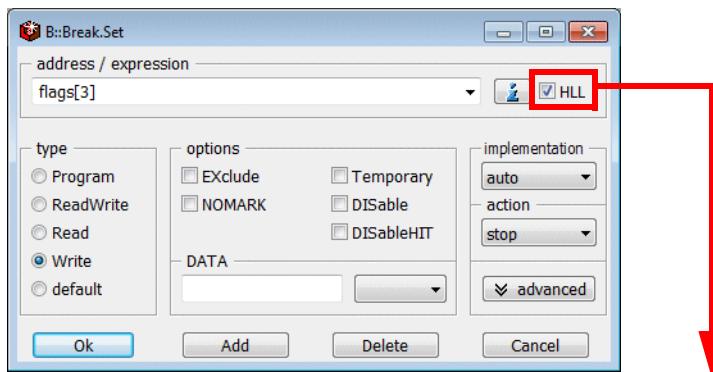


```
Break.Set flags
```

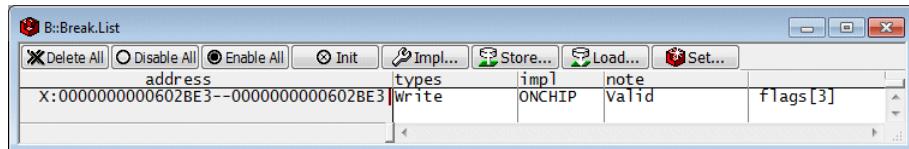
```
sYmbol.INFO flags ; display symbol information
; for variable flags
```



- 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.



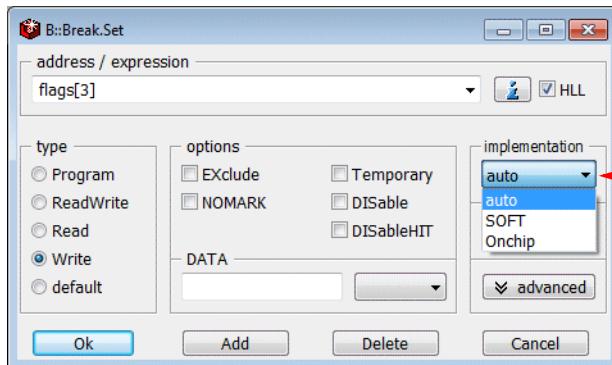
Selected breakpoint is set to the address range used by the HLL-expression



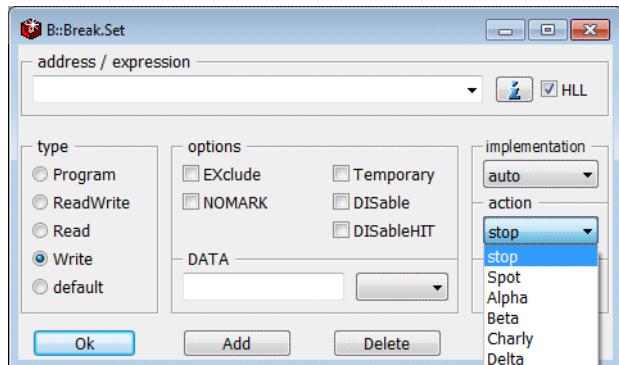
```
Var.Break.Set flags[3]
```

- otherwise the breakpoint is rejected with an error message.





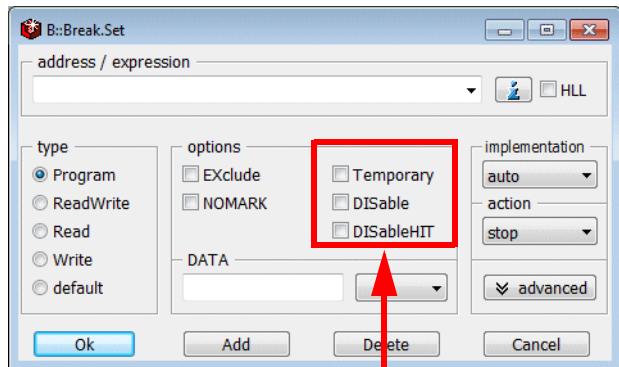
Implementation	
<b>auto</b>	Use breakpoint implementation predefined in TRACE32 PowerView.
<b>SOFT</b>	Implement breakpoint as Software breakpoint.
<b>Onchip</b>	Implement breakpoint as Onchip breakpoint.



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

Action (debugger)	
<b>Spot</b>	The program execution is stopped shortly (50..100ms) at a breakpoint hit to update the screen. As soon as the screen is updated, the program execution continues.
<b>Alpha</b>	Set an Alpha breakpoint.
<b>Beta</b>	Set a Beta breakpoint.
<b>Charly</b>	Set a Charly breakpoint.
<b>Delta</b>	Set a Delta breakpoint.
<b>Echo</b>	Set an Echo breakpoint.

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



Options

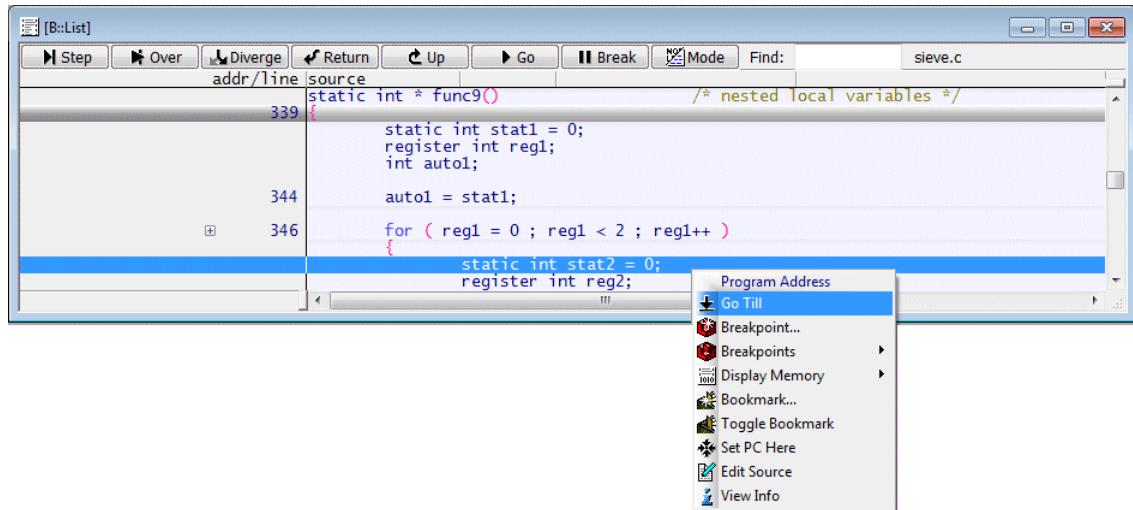
<b>Temporary</b>	<b>OFF:</b> Set a permanent breakpoint (default). <b>ON:</b> Set a temporary breakpoint. All temporary breakpoints are deleted the next time the core(s) stops the program execution.
<b>DISable</b>	<b>OFF:</b> Breakpoint is enabled (default). <b>ON:</b> Set breakpoint, but disabled.
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

## Example for the Option Temporary

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

### Examples:

- **Go Till**

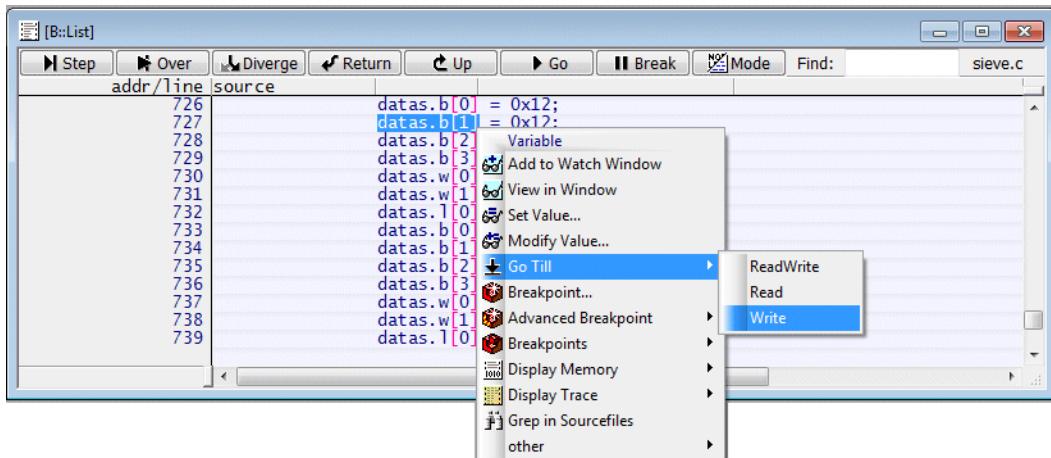


**Go <address> [ address> ...]**

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

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

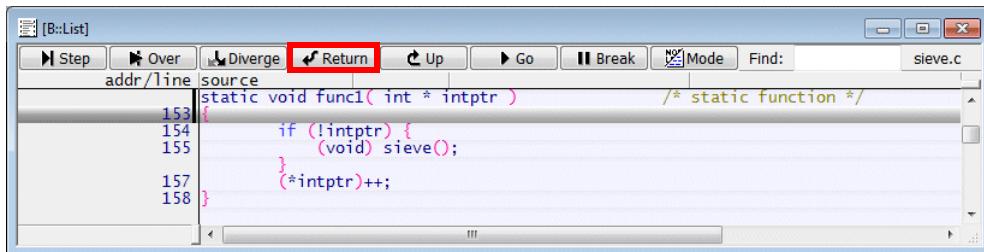
- Go Till -> Write



**Var.Go <hi\_l\_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**



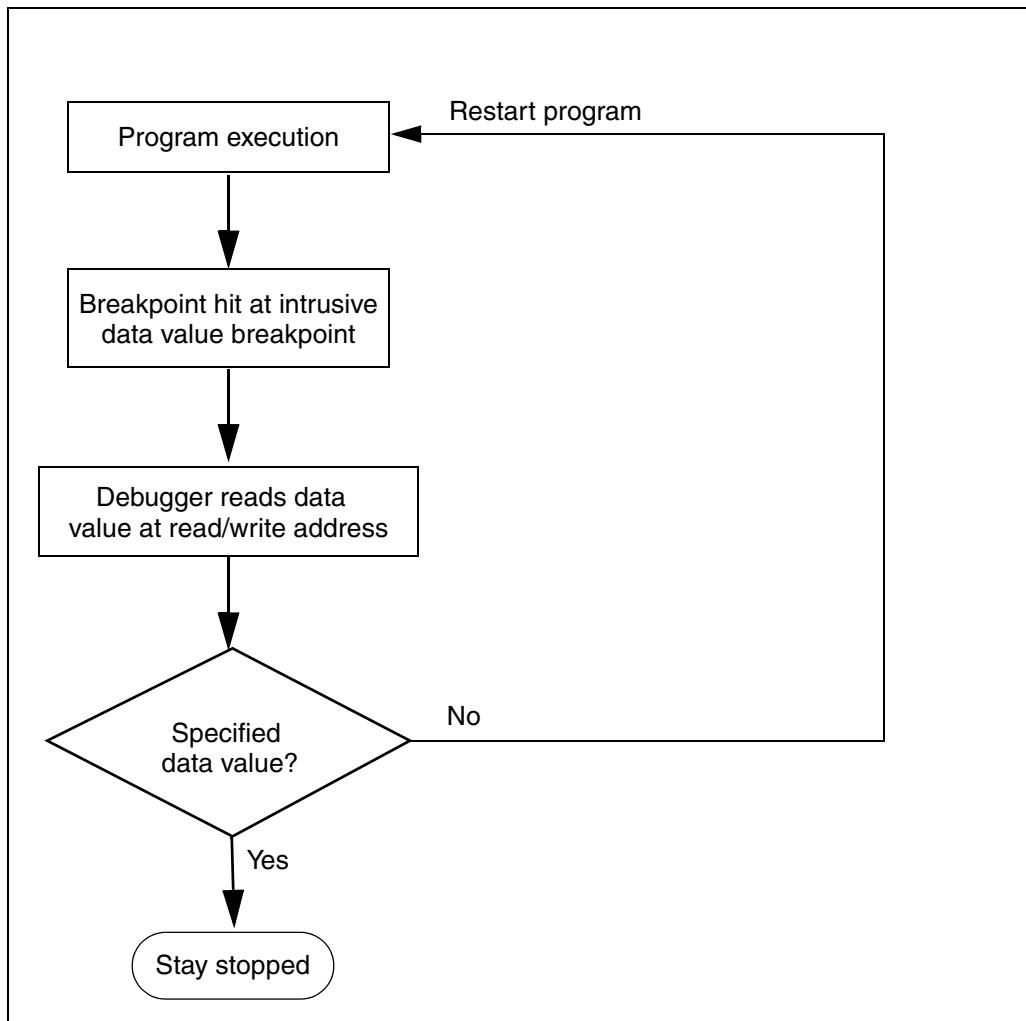
### Go.Return

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

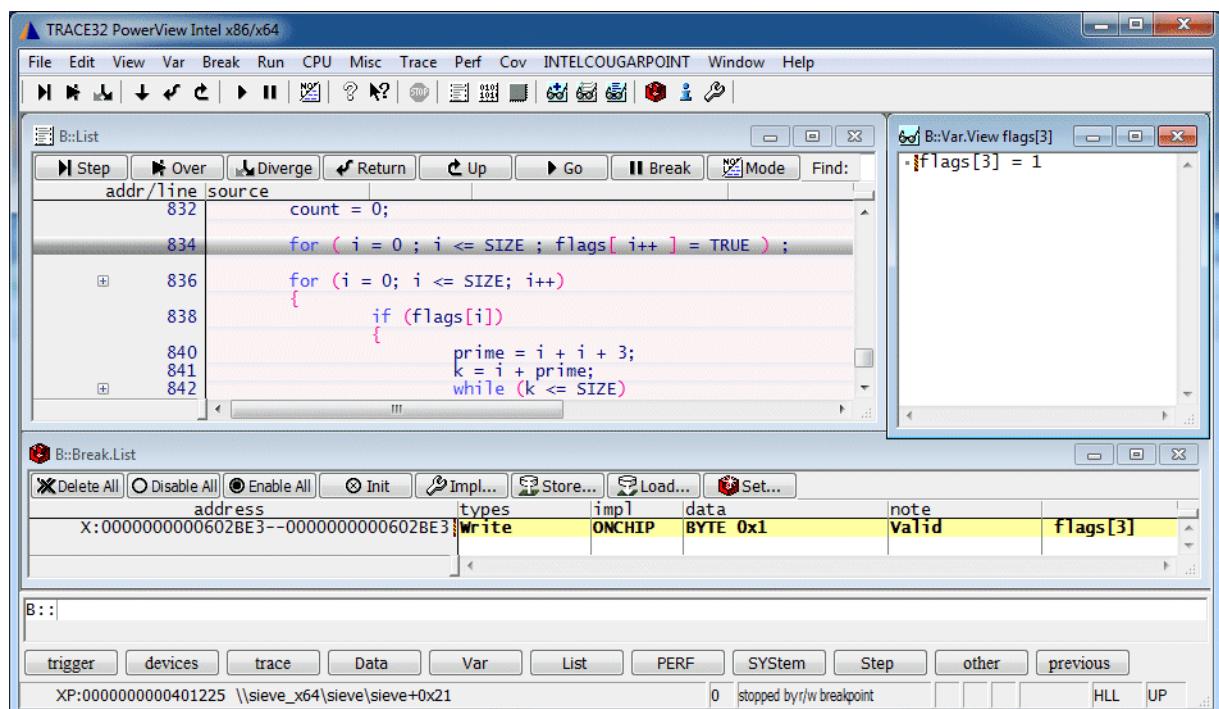
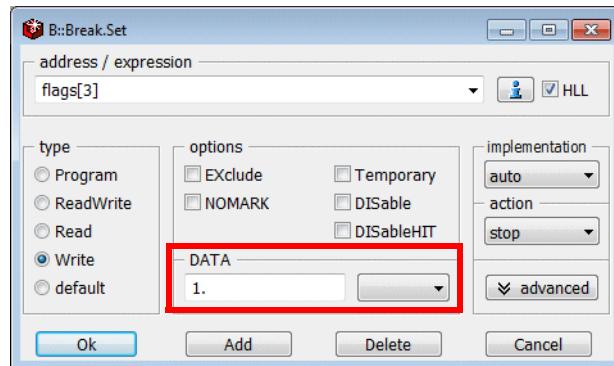
## DATA Breakpoints

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:

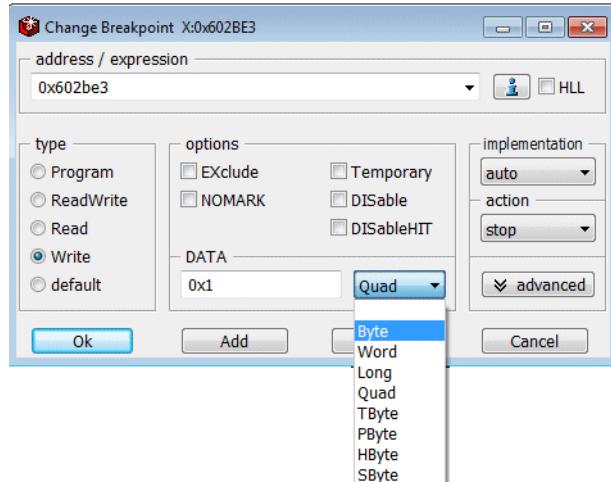


## Example 1: Stop the program execution if a 1 is written to flags[3].



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

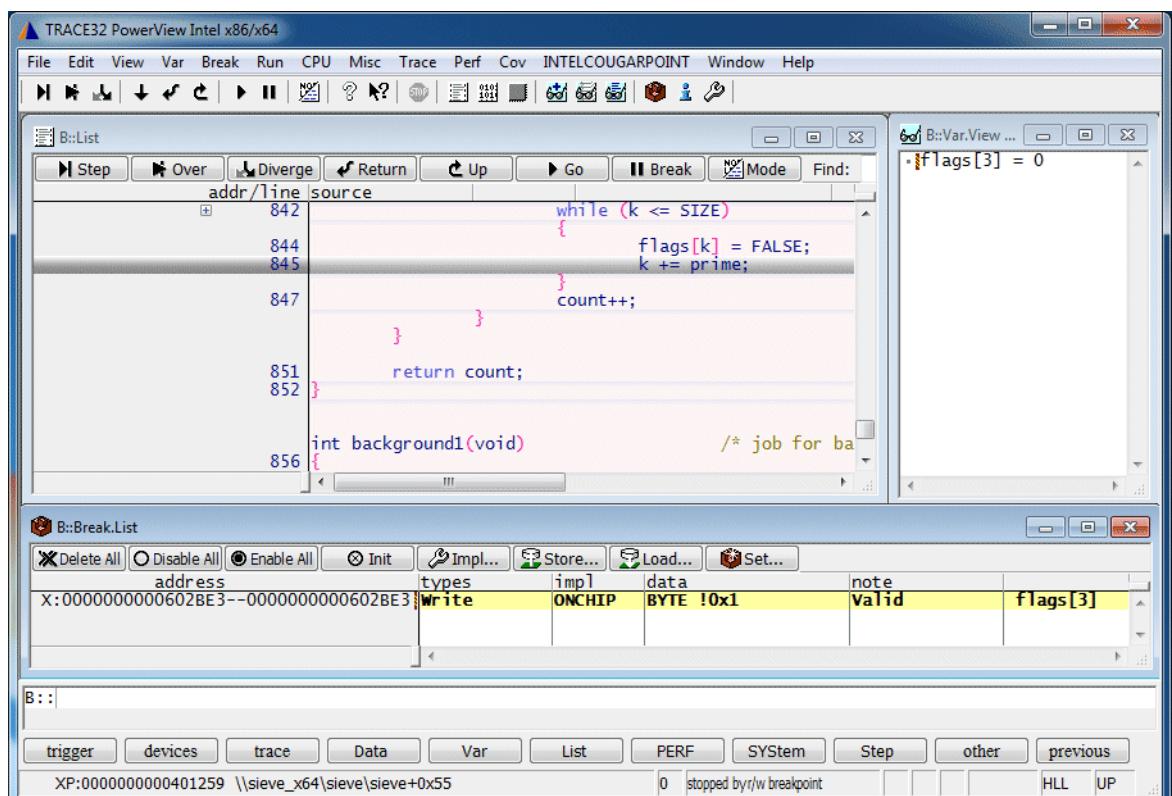
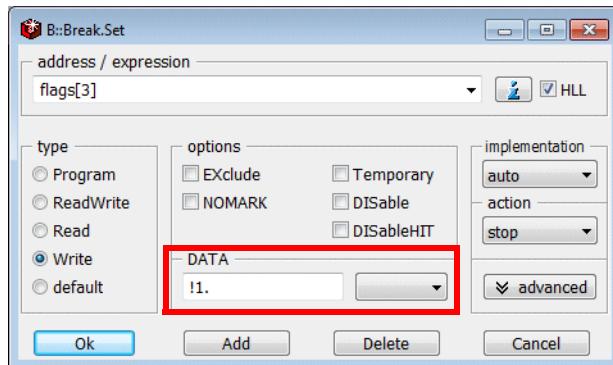
**Var.Break.Set <hll\_expression> /[Write | ReadWrite] /DATA.auto <value>**

**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].



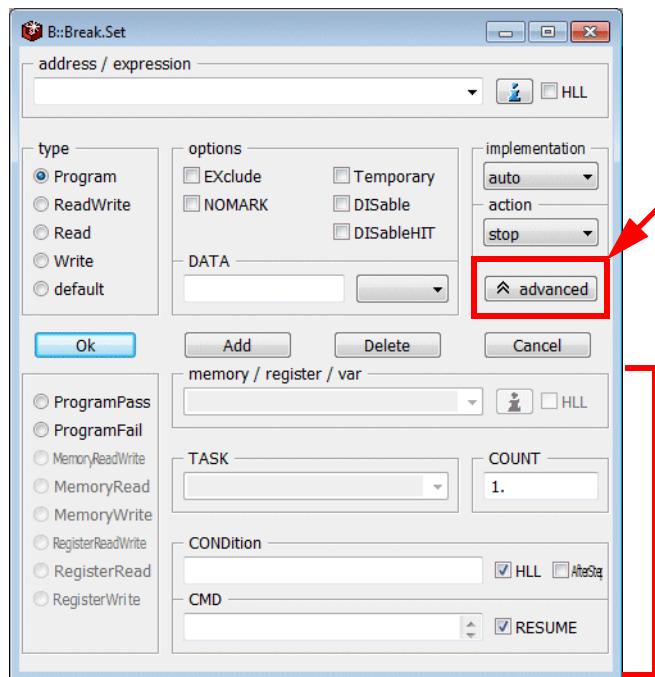
**Var.Break.Set** <hll\_expression> /[Write | ReadWrite] /DATA.auto !<value>

**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

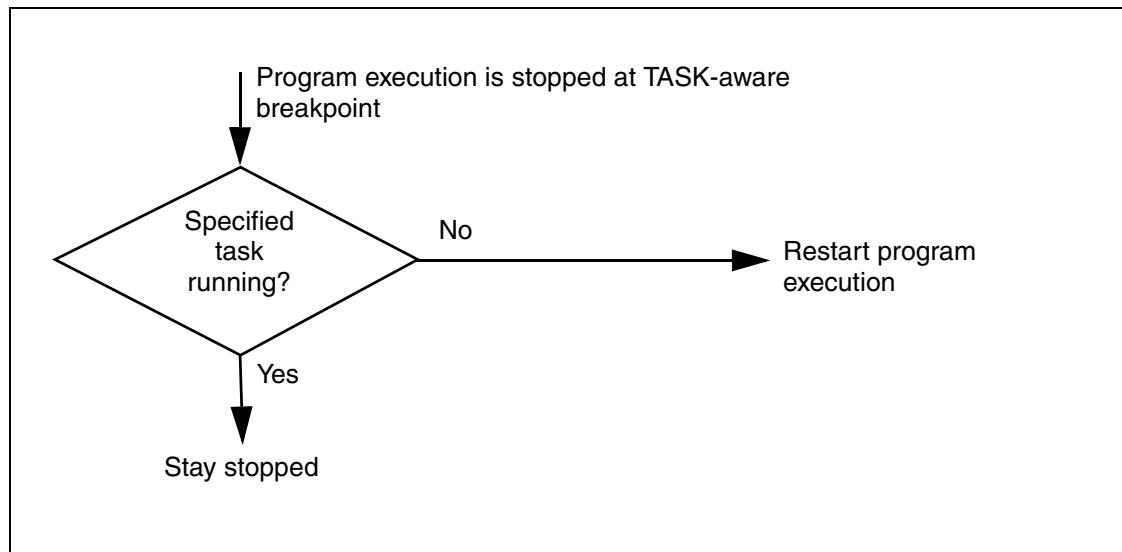
Advanced breakpoint input fields

## TASK-aware Breakpoints

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

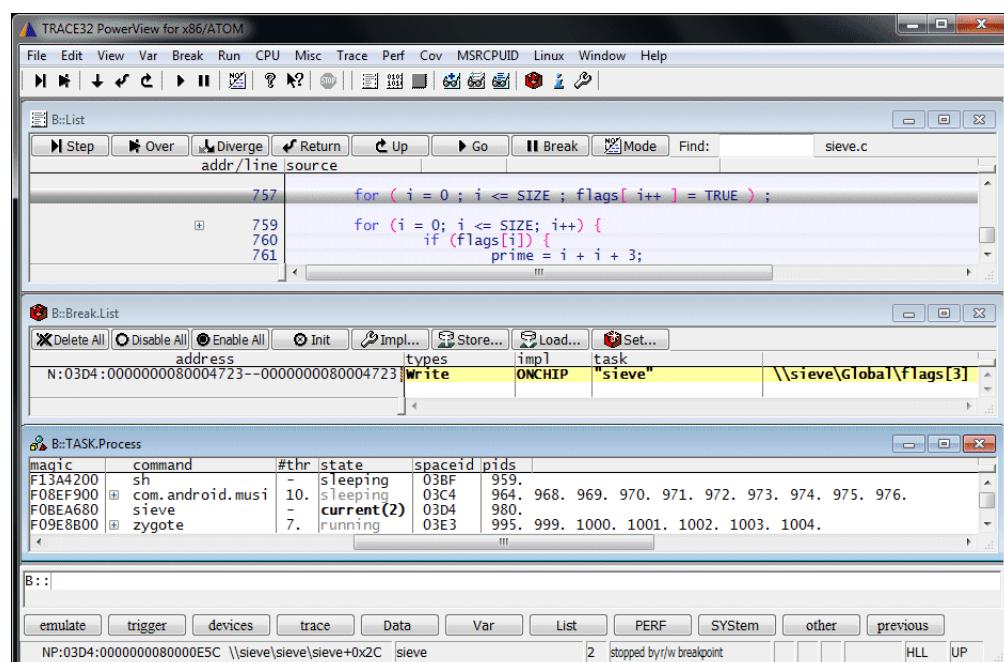
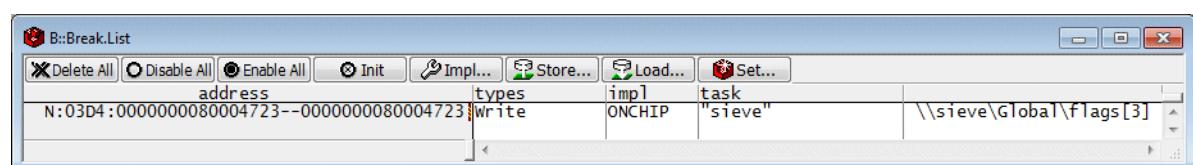
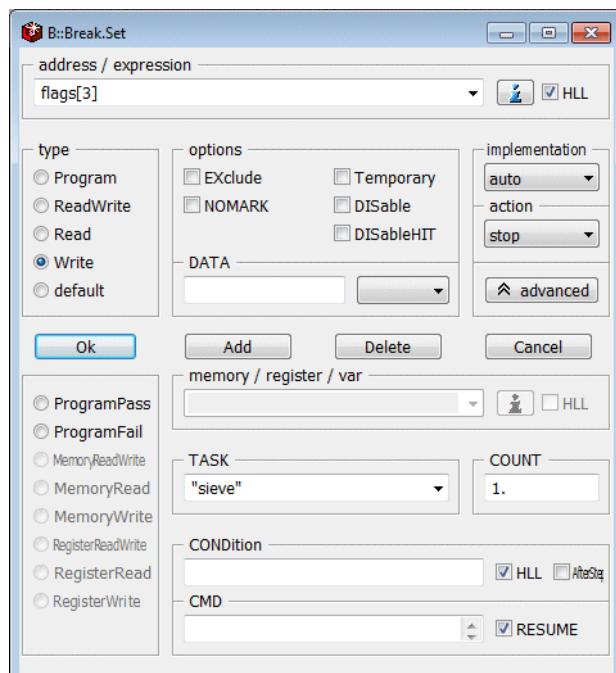
TASK-aware breakpoints are implemented as intrusive breakpoints.

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>
```

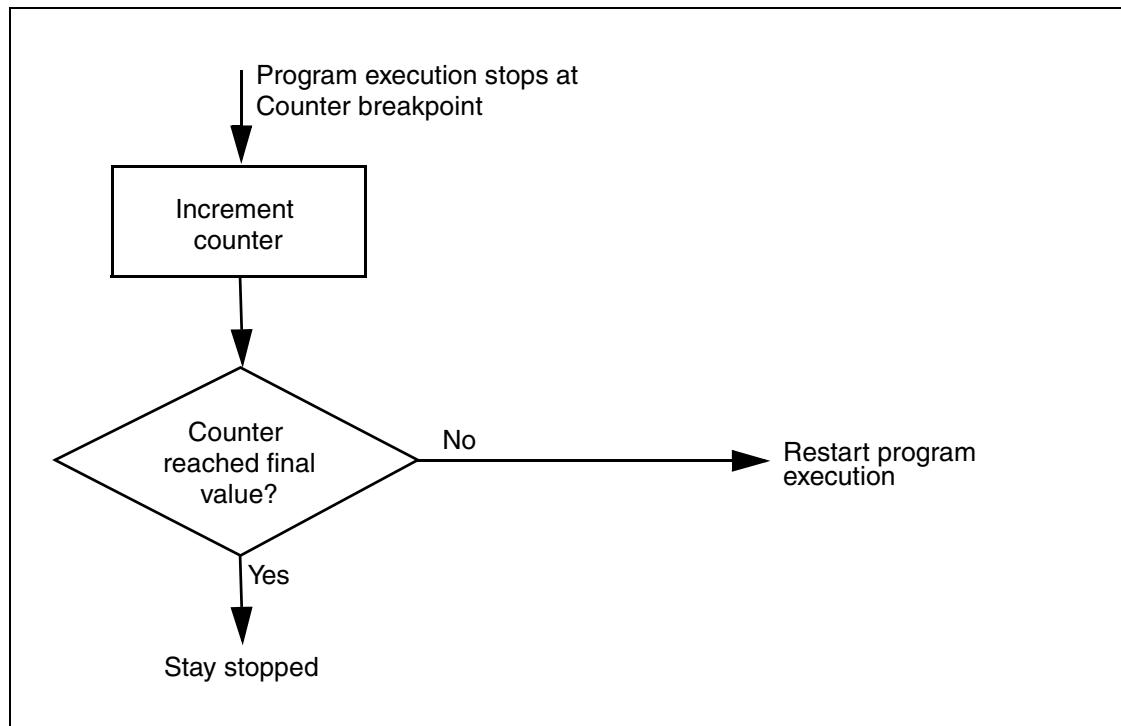
```
; use task ID to specify task  
Break.Set 0x602be3 /Program /TASK 223.  
  
; use task name to specify task  
Var.Break.Set flags[3] /Write /TASK "sieve"
```

## Counter

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

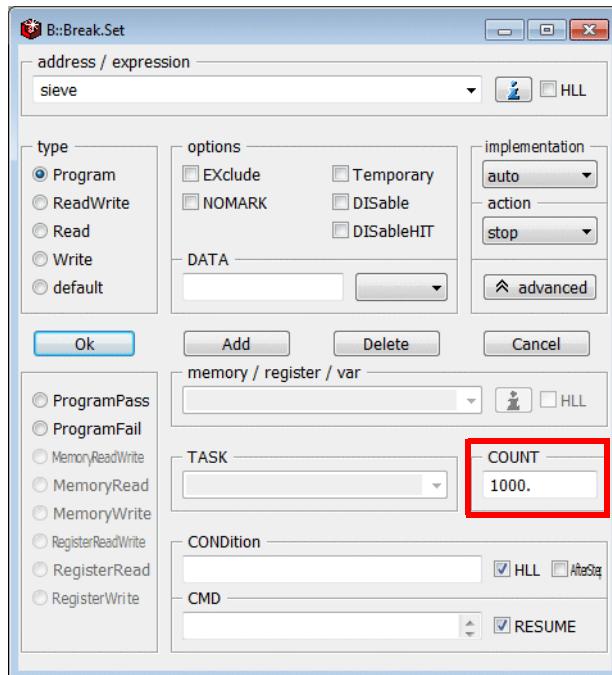
The onchip breakpoint logic of Intel® x86/x64 does not provide counters, so counters are implemented as software counters.

Processing:

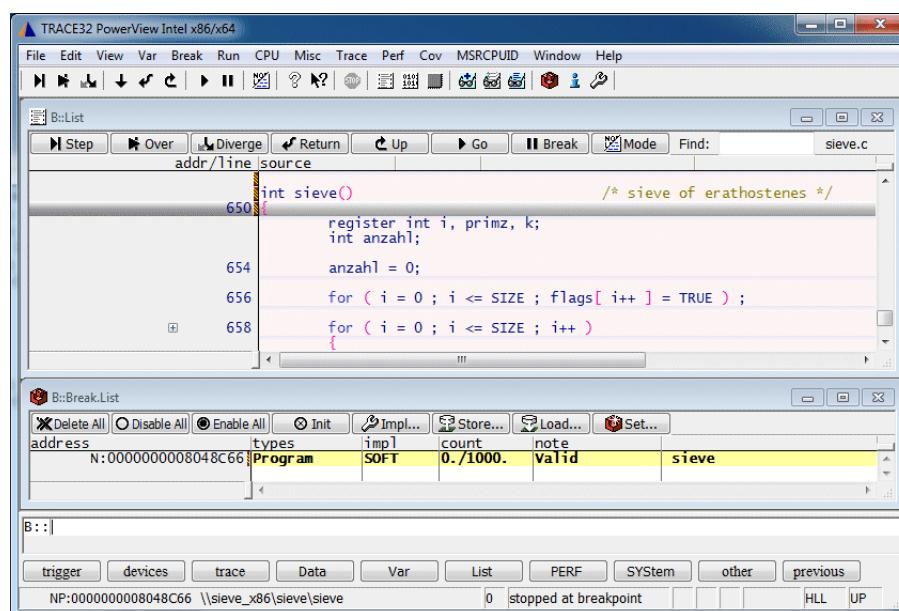
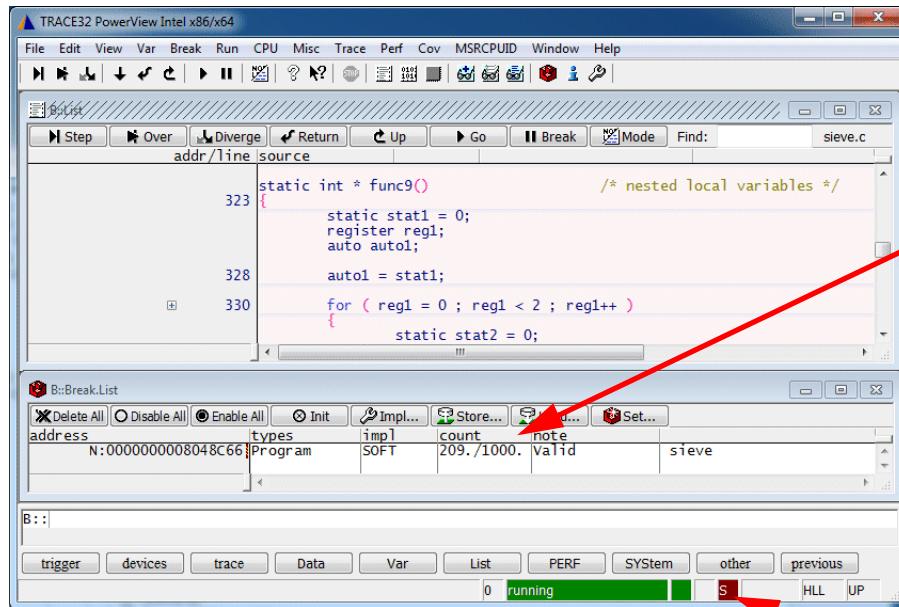


Each stop at a Counter breakpoint takes at least 1.ms. This is why the red S is displayed in the Debugger Activity field of the TRACE32 PowerView state line whenever the breakpoint is hit.

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



B::Break.List					
address	types	impl	count	note	
N:0000000008048C66	Program	SOFT	0./1000.	Valid	sieve

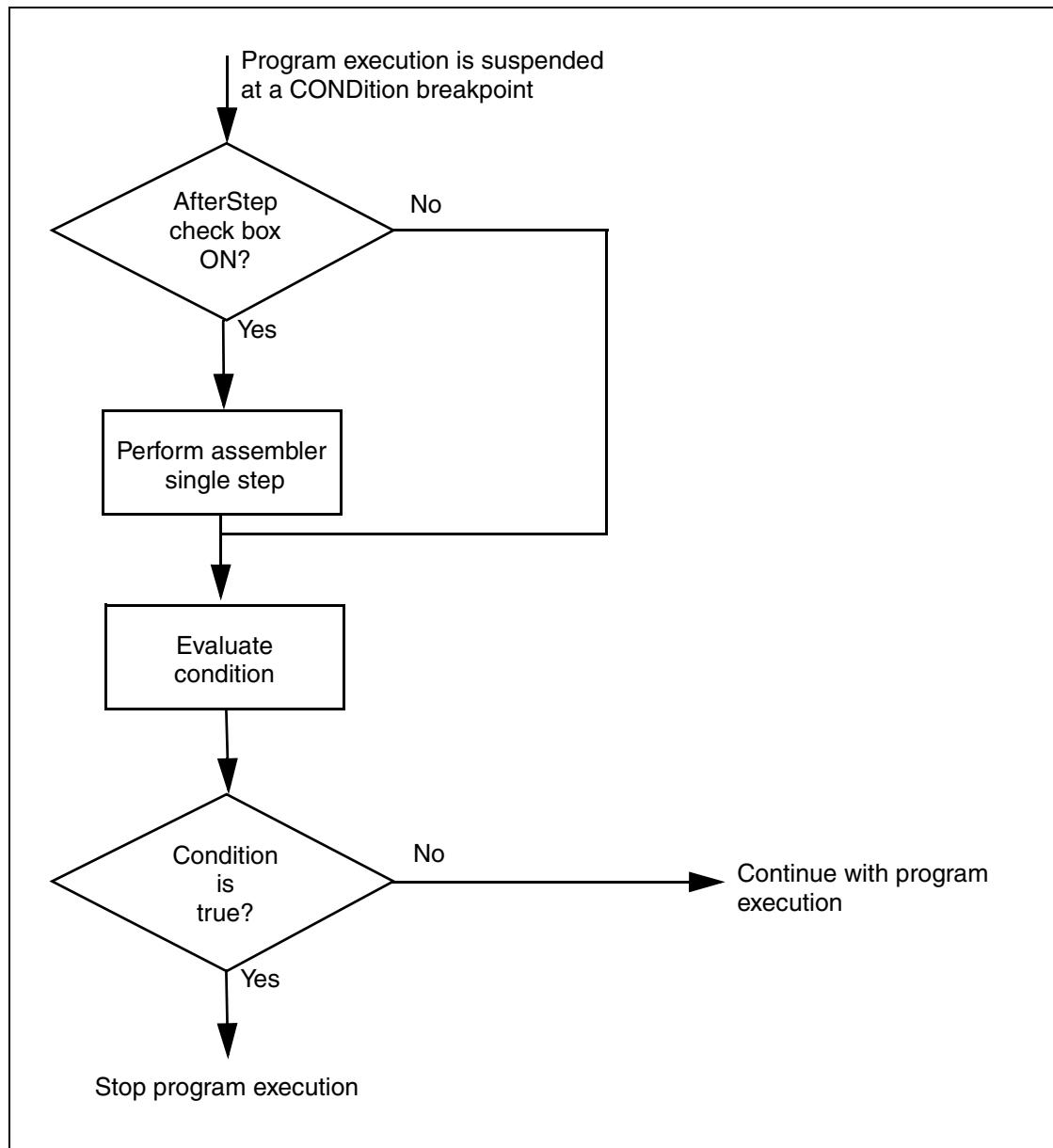


```
Break.Set sieve /COUNT 1000.
```

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

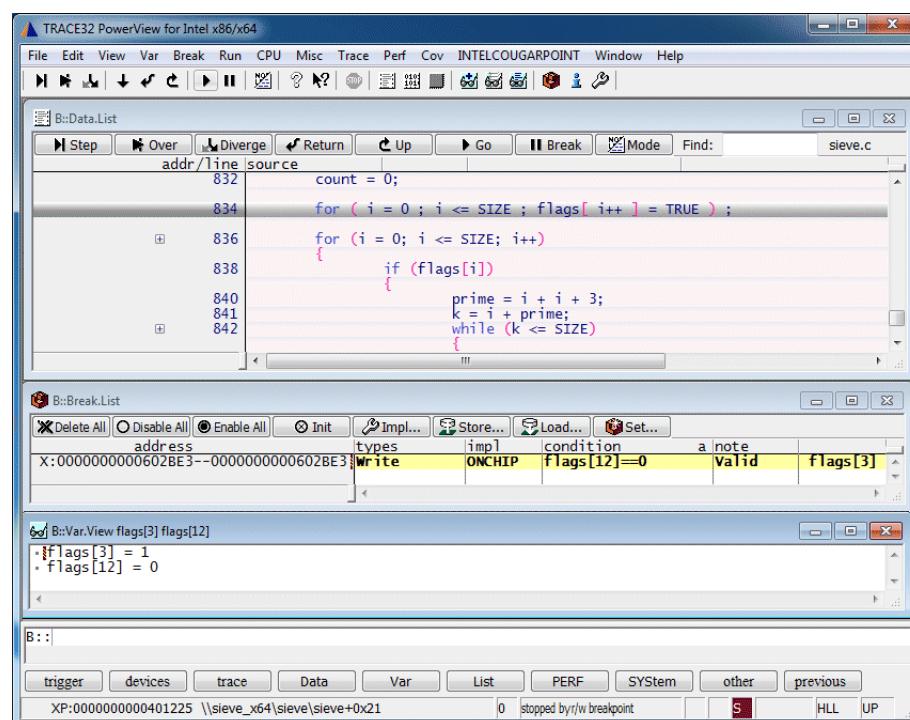
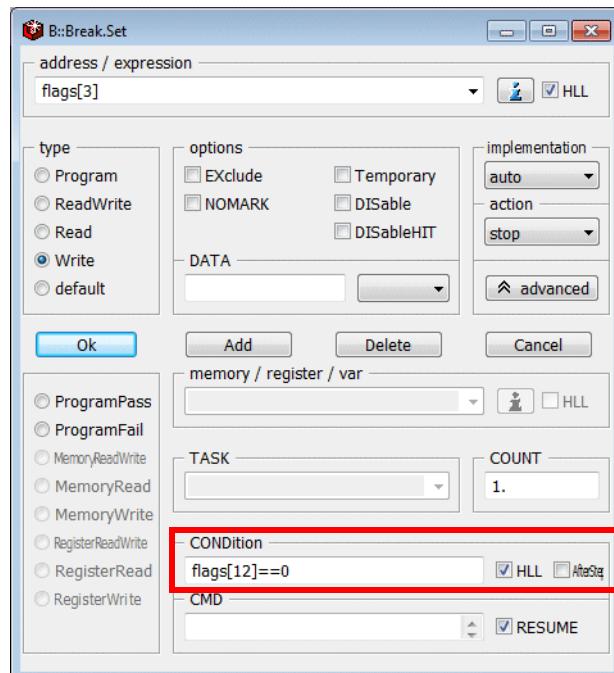
CONDition breakpoints are always intrusive.

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.



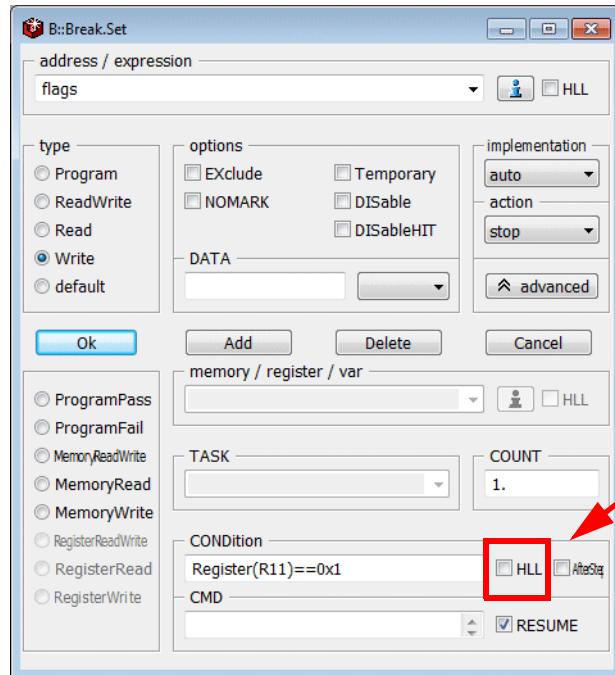
The red S indicates an intrusive breakpoint

**Var.Break.Set** <hl\_expression> /[Program | Write | ReadWrite] /VarCONDition <hl\_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>**

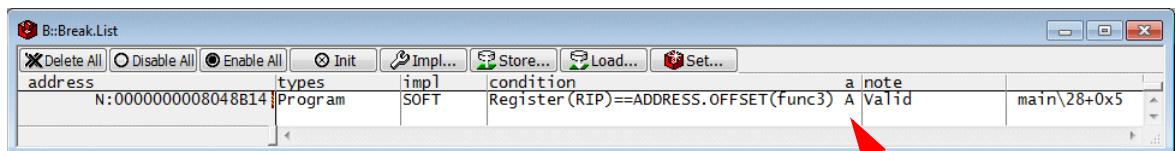
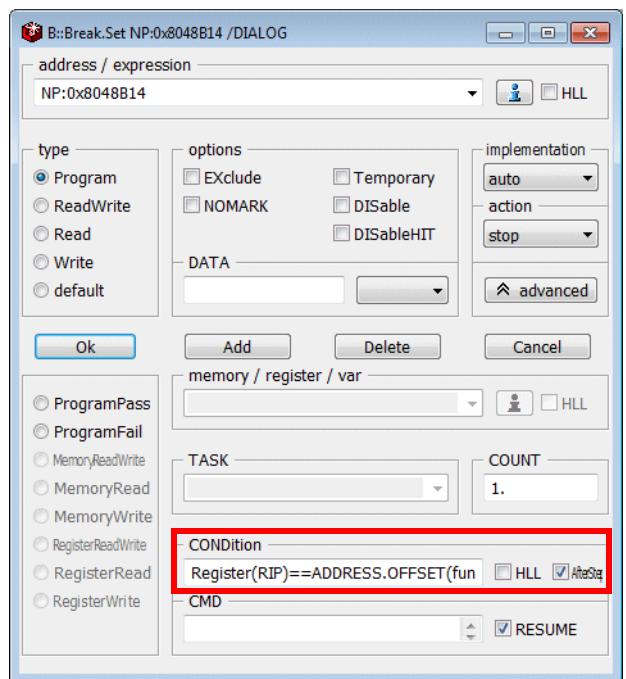
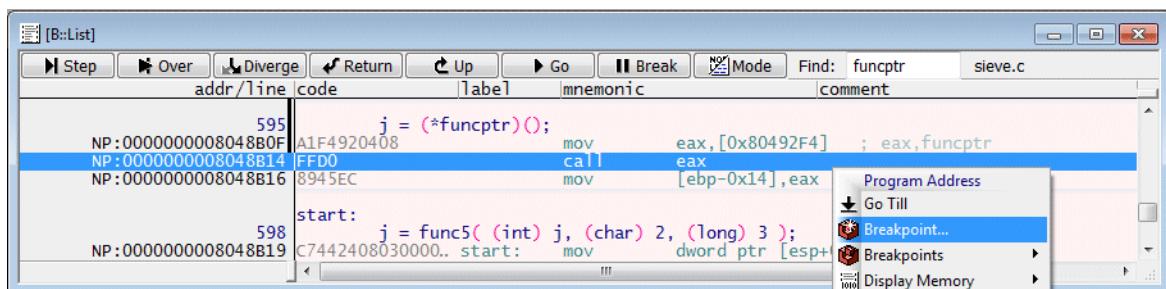
```
; stop the program execution at a write to the address flags if the
; register R11 is equal to 1
```

```
Break.Set flags /Write /CONDition Register(R11)==0x1
```

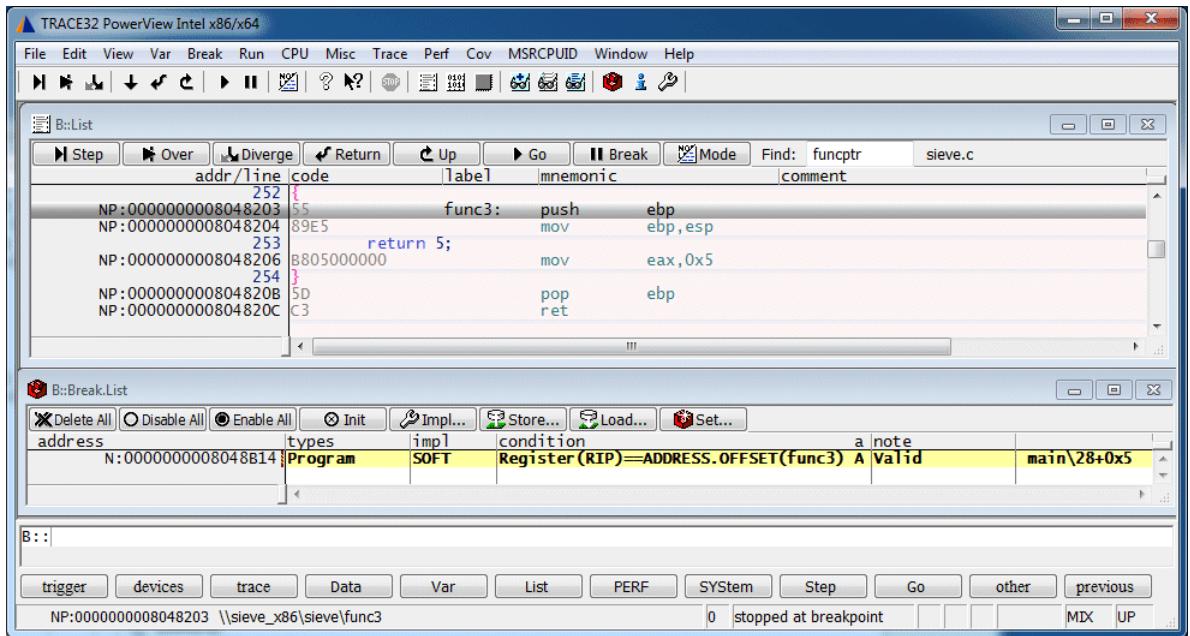
```
; stop program execution at a write to the address flags if the long
; at address ND:0x1000 is larger then 0x12345
```

```
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

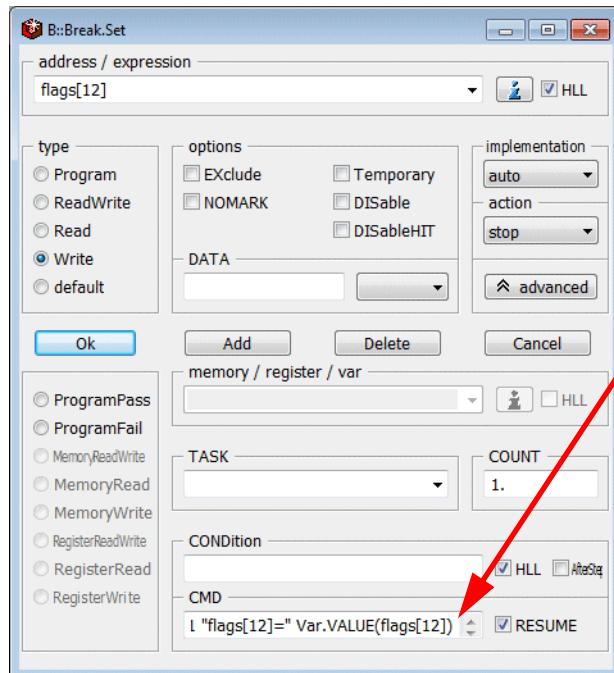
```
Break.Set main\44+0x7 /Program
/CONDition Register(RIP)==ADDRESS.OFFSET(func3) /AfterStep
```

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.

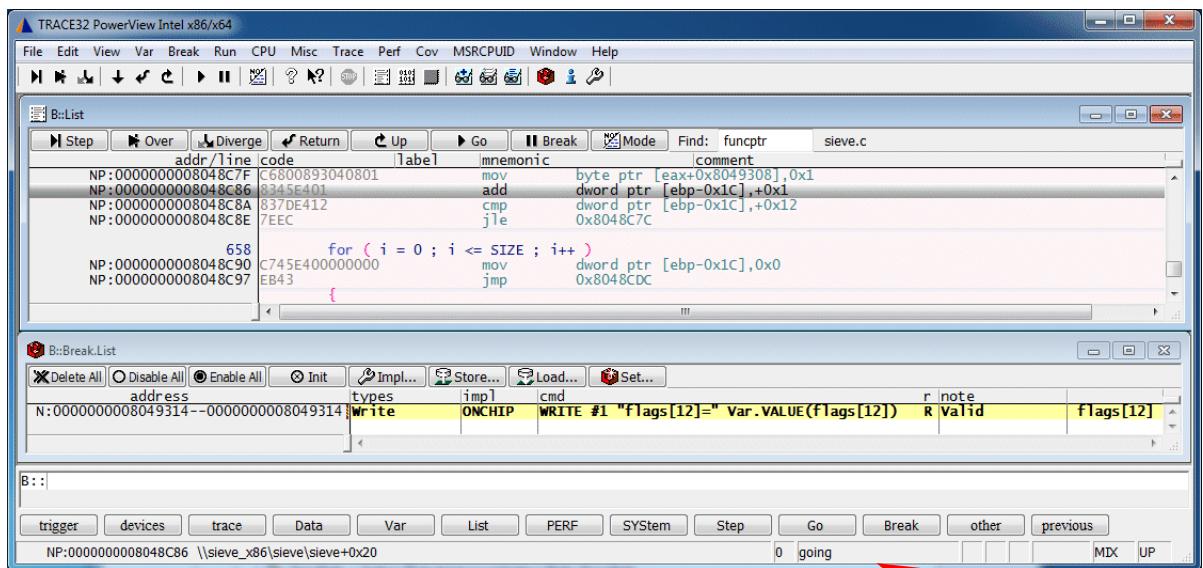
address	types	impl	cmd	r	note	f
N:0000000008049314	write	ONCHIP	WRITE #1 "flags[12] = " Var.VALUE(flags[12])	R	Valid	flags[12]



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.

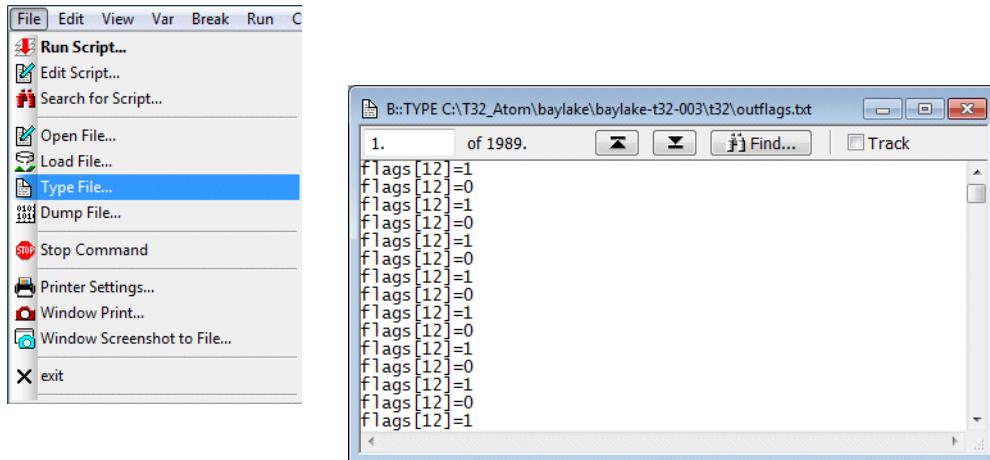


The state of the debugger toggles between **going** and **stopped**

**CLOSE #1**

; close the file when you are done

Display the result:

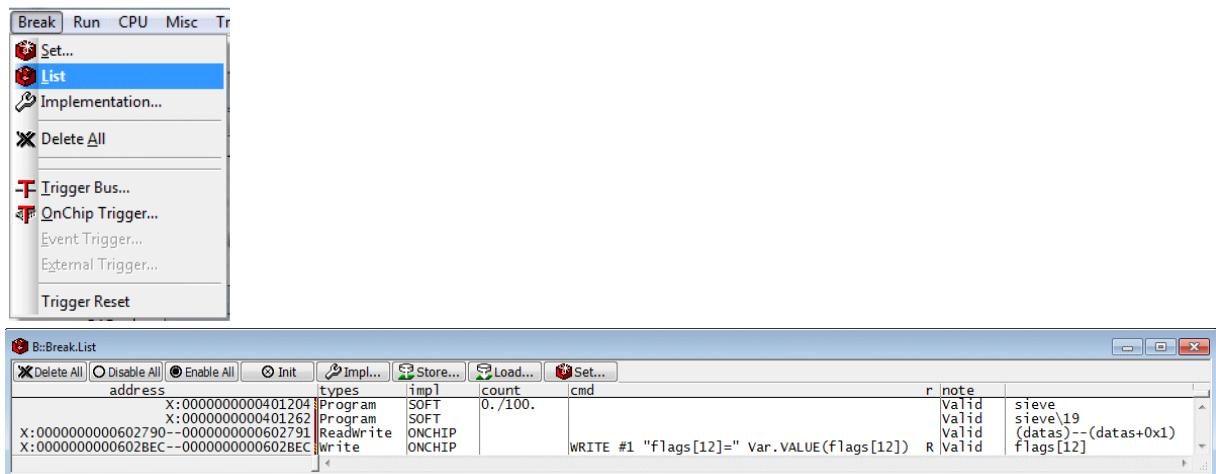


**Break.Set <address> | <range> [/Program | Write | ReadWrite] /CMD {<command>} [/RESUME]**

**Var.Break.Set <hl\_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



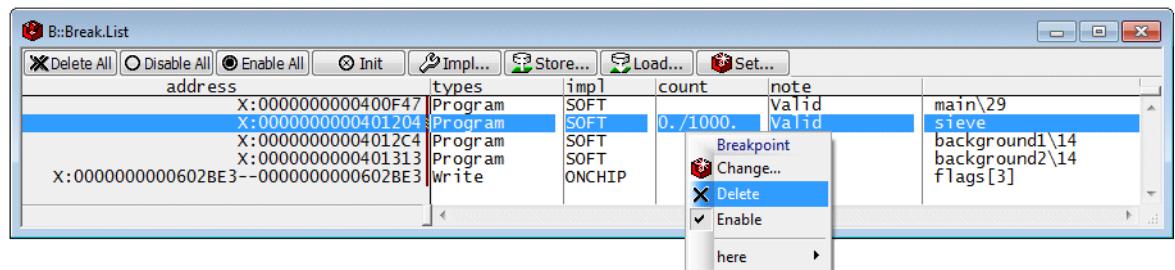
The image shows the 'Break' menu with the 'List' option highlighted. Below the menu is a window titled 'Break.List' showing a table of breakpoints. The table has columns: address, types, impl, count, cmd, r, note. The data is as follows:

address	types	impl	count	cmd	r	note
X:00000000000401204	Program	SOFT	0./100.		Valid	sieve
X:00000000000401262	Program	SOFT			Valid	sieve\19
X:00000000000602791	Readwrite	ONCHIP		WRITE #1 "flags[12] = " Var.VALUE(flags[12])	Valid	(data) -- (data+0x1)
X:00000000000602BEC	Write	ONCHIP			Valid	flags[12]

## Break.List

List all breakpoints

## Delete Breakpoints

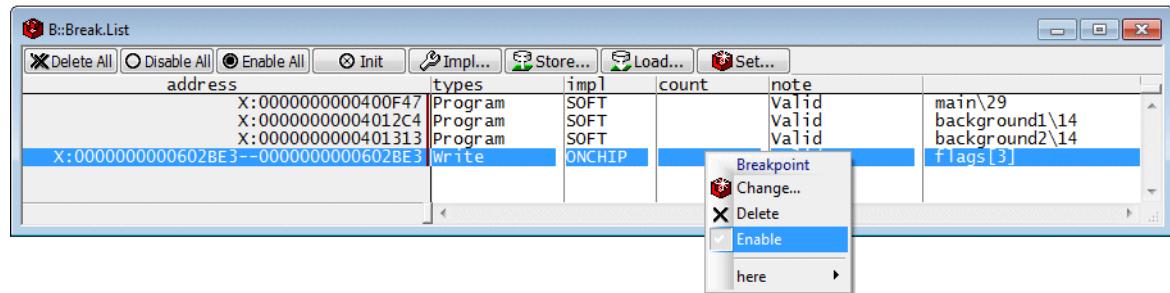


The image shows the 'Break.List' window with a context menu open over a selected breakpoint. The menu options are: Change..., Delete, Enable, and here. The selected breakpoint is X:00000000000401204, which is a Program type, SOFT impl, with a count of 0./1000. The note for this breakpoint is 'main\29 sieve'.

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

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

# Enable/Disable Breakpoints



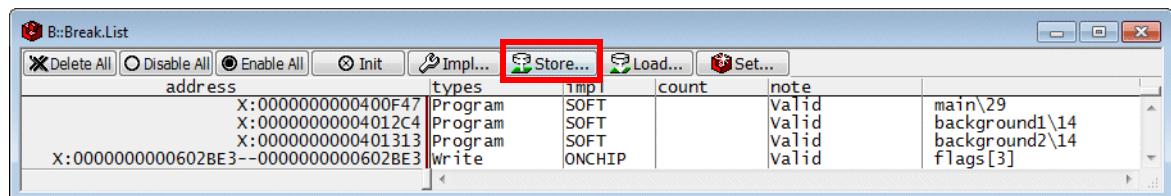
**Break.ENABLE** [<address>|<address\_range>] [/<option>]

Enable breakpoint

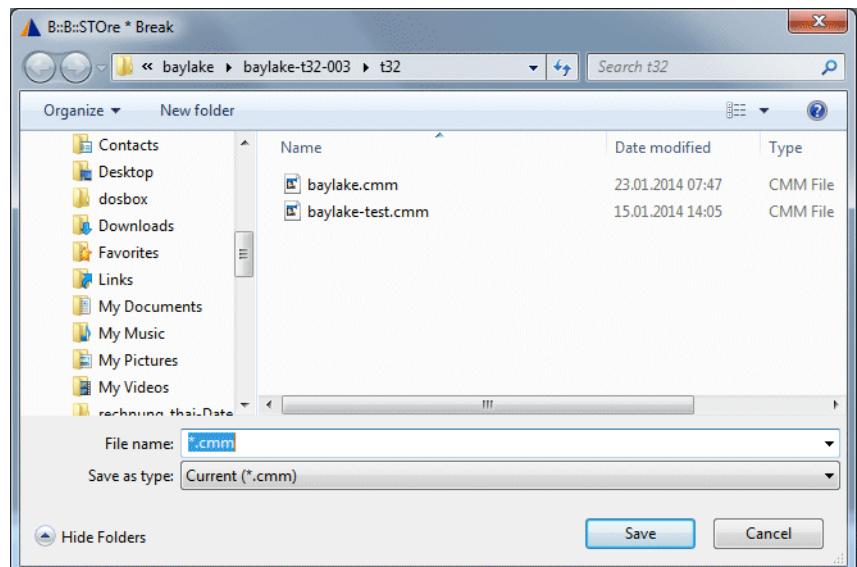
**Break.DISABLE** [<address>|<address\_range>] [/<option>]

Disable breakpoint

# Store Breakpoint Settings



address	types	tmp1	count	note
X:00000000000400F47	Program	SOFT		Valid
X:000000000004012C4	Program	SOFT		Valid
X:00000000000401313	Program	SOFT		Valid
X:0000000000602BE3--000000000602BE3	Write	ONCHIP		Valid



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

```
B:::
```

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

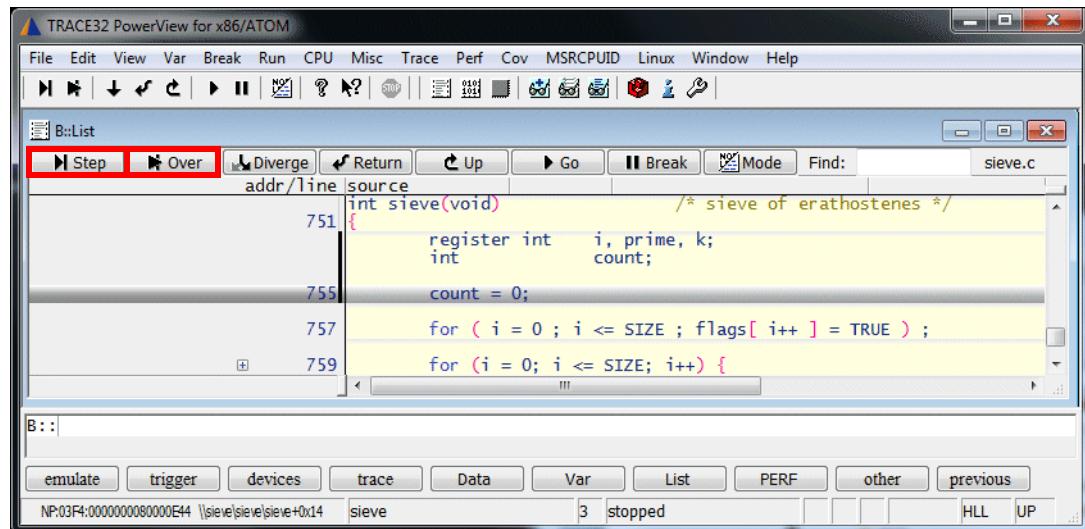
```
ENDDO
```

**STOre <filename> Break**

Generate a script for breakpoint settings

## Basic Debug Control

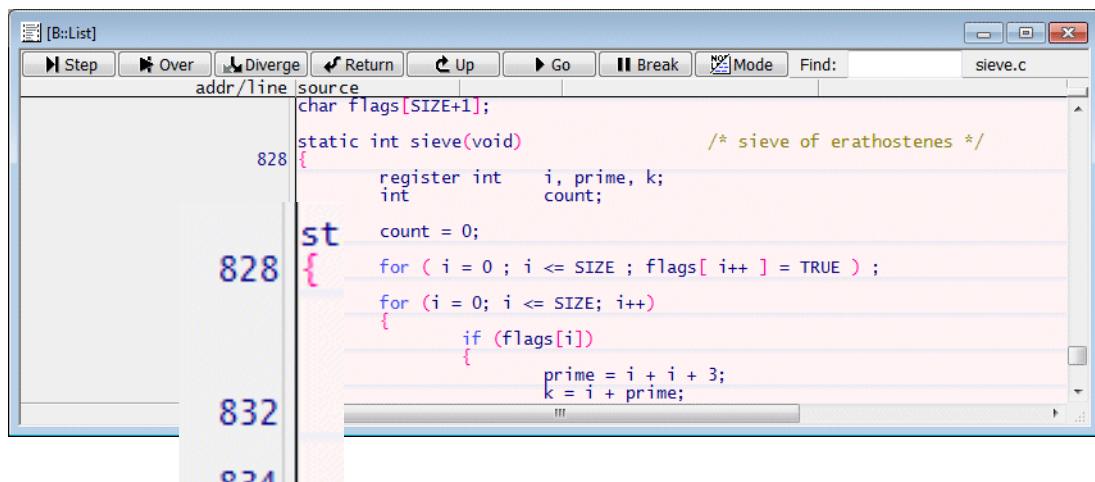
There are local buttons in the **List** window for all basic debug commands:



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

## More details on Step.Diverge

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



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

The following command allows you to get more details:

```
List.auto /DIVERGE
```

B::List /DIVERGE

s	state	i	addr/line	source
h	stop		828	register int i, prime, k; int count;
h	done		832	count = 0;
h	done		834	for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
h	done	target	836	for (i = 0; i <= SIZE; i++) for (i = 0; i <= SIZE; i++)
h	done		838	{ if (flags[i])
hit			840	{ prime = i + i + 3;
			841	k = i + prime;
			842	while (k <= SIZE)
			844	{ flags[k] = FALSE;
			845	k += prime;
			847	} } }



Drag this handle to see the DIVERGE details

B::List /DIVERGE

s	state	i	addr/line	code	label	mnemonic	comment
a	done		834	For ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;			
a	done		XP:0000000000401214	BB00000000		mov	ebx,0x0
a	done		XP:0000000000401219	E8D		jmp	0x401228
			XP:000000000040121B	4863C3		movsx	rax,ebx
			XP:000000000040121E	C680E02B600001		mov	byte ptr [rax+0x602BE0],0x1
			XP:0000000000401225	83C301		add	ebx,+0x1
a	done	hit	XP:0000000000401228	83FB12		cmp	ebx,+0x12
			XP:0000000000401228	7EE		jle	0x40121B
			836	BB00000000			
			XP:000000000040122D	E835		mov	ebx,0x0
			XP:0000000000401232			jmp	0x401269

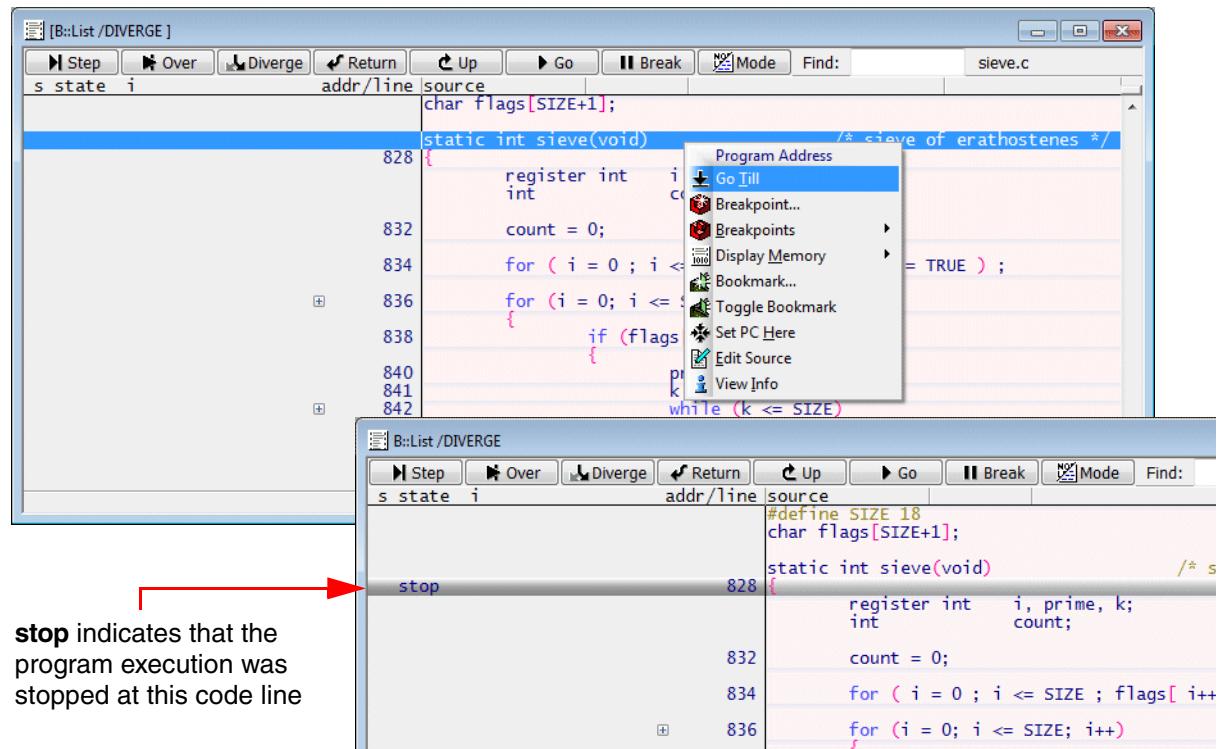


### Column layout

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

## Example 1: Diverge through function sieve.

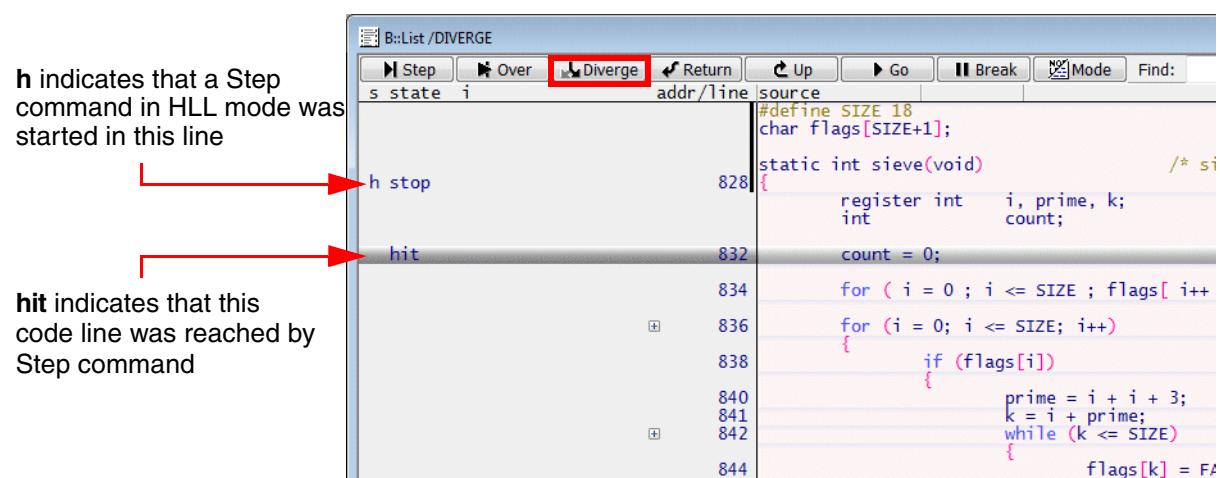
### 1. Run program execution until entry to function sieve.



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

```
#define SIZE 18
char flags[SIZE+1];
static int sieve(void) /* sieve of erathostenes */
{
    register int i, prime, k;
    int count = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i ] = TRUE ) ;
    for ( i = 0; i <= SIZE; i++ )
    {
        if (flags[i])
        {
            prime = i + i + 3;
            k = i + prime;
            while (k <= SIZE)
            {
                flags[k] = FALSE;
                k = k + prime;
            }
        }
    }
}
```

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



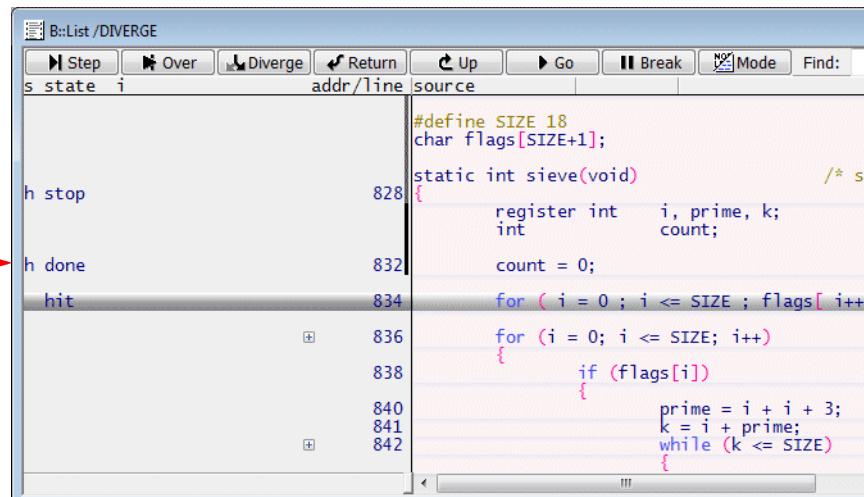
h indicates that a Step command in HLL mode was started in this line

hit indicates that this code line was reached by Step command

```
#define SIZE 18
char flags[SIZE+1];
static int sieve(void) /* sieve of erathostenes */
{
    register int i, prime, k;
    int count = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i ] = TRUE ) ;
    for ( i = 0; i <= SIZE; i++ )
    {
        if (flags[i])
        {
            prime = i + i + 3;
            k = i + prime;
            while (k <= SIZE)
            {
                flags[k] = FALSE;
                k = k + prime;
            }
        }
    }
}
```

### 3. Continue with Step.Diverge.

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



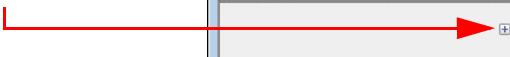
The screenshot shows a debugger's memory dump window titled "B::List /DIVERGE". The window has a toolbar with buttons for Step, Over, Diverge, Return, Up, Go, Break, and Mode. The columns are labeled "state", "addr/line", and "source". The "source" column displays the following C code:

```
#define SIZE 18
char flags[SIZE+1];
static int sieve(void)
{
    register int i, prime, k;
    int count;
    count = 0;
    for ( i = 0 ; i <= SIZE ; flags[ i++ ]
    for ( i = 0; i <= SIZE; i++)
    {
        if (flags[i])
        {
            prime = i + i + 3;
            k = i + prime;
            while (k <= SIZE)
            {

```

The line 834, which is highlighted, has a status of "done". A red arrow points to this "done" status. The line numbers 828, 832, 834, 836, 838, 840, 841, and 842 are also visible.

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



```

B::List / DIVERGE
Step Over Diverge Return Up Go Break Mode Find:
s state i addr/line source
h stop 828 #define SIZE 18
h done 832 static int sieve(void) /* siev
hit 834 register int i, prime, k;
     int count = 0;
     for (i = 0; i <= SIZE; i++)
     {
         if (flags[i])
             prime = i + i + 3;
     }

```

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



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

```

B::List / DIVERGE
Step Over Diverge Return Up Go Break Mode Find:
s state i addr/line source
h stop 828 #define SIZE 18
h done 832 static int sieve(void) /* siev
h done 834 register int i, prime, k;
hit 836 int count = 0;
     for (i = 0; i <= SIZE; i++)
     {
         for (j = 0; j <= SIZE; j++)
         {
             if (flags[i])
                 prime = i + i + 3;
             k = i + prime;
             while (k <= SIZE)
             {

```

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



Assembly code (B::List /DIVERGE):

```

addr/line source
828 h stop
832 h done
834 h done
836 hit
836

```

Source code (B::List /DIVERGE):

```

#define SIZE 18
char flags[SIZE+1];
static int sieve(void)
{
    register int i, prime, k;
    int count = 0;
    for (i = 0; i <= SIZE; flags[i] = 1)
        for (k = 0; k <= SIZE; i++)
            if (flags[i])
                prime = i + i + 3;
                k = i + prime;
                while (k <= SIZE)
                    flags[k] = 1;
                    k += prime;
    count++;
    return count;
}

```

Assembly code (B::List.Mix /Track):

```

addr/line code
836 for (i = 0; i <= SIZE; i++) mov ebx, 0x0
XP:000000000040122D BB00000000
XP:0000000000401232 EB35 { jmp 0x401269

```

## 5. Continue with Step.Diverge.

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



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



Assembly code (B::List /DIVERGE):

```

addr/line source
828 h stop
832 h done
834 h done
836 hit
836

```

Source code (B::List /DIVERGE):

```

#define SIZE 18
char flags[SIZE+1];
static int sieve(void)
{
    register int i, prime, k;
    int count = 0;
    for (i = 0; i <= SIZE; flags[i] = 1)
        for (k = 0; k <= SIZE; i++)
            if (flags[i])
                prime = i + i + 3;
                k = i + prime;
                while (k <= SIZE)
                    flags[k] = 1;
                    k += prime;
    count++;
    return count;
}

```

Assembly code (B::List.Mix /Track):

```

addr/line code
836 for (i = 0; i <= SIZE; i++) mov ebx, 0x0
840
841
842
844
845
847
851
852

```

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

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

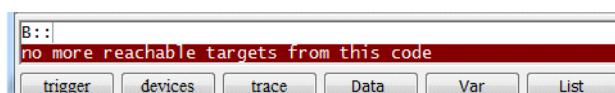
s	state	i	addr/line	source
	h stop		828	static int sieve(void)
	h done		832	register int i, prime, k;
	h done		834	int count = 0;
	h done	target	836	for (i = 0; i <= SIZE; i++)
	h done		836	for (i = 0; i <= SIZE; i++)
	h done		838	{
	h done		840	if (flags[i])
	h done		841	prime = i + i;
	h done		842	k = i + prime;
	h done	hit	844	while (k <= SIZE)
	target		845	{
	target		847	flags[k] = 1;
	target		851	k += prime;
			852	}
				count++;
				}
				return count;

## 7. Continue with Step.Diverge.

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

s	state	i	addr/line	source
	h stop		828	static int sieve(void)
	h done		832	register int i, prime, k;
	h done		834	int count = 0;
	h done	target	836	for (i = 0; i <= SIZE; i++)
	h done		836	for (i = 0; i <= SIZE; i++)
	h done		838	{
	h done		840	if (flags[i])
	h done		841	prime = i + i;
	h done		842	k = i + prime;
	hit		844	while (k <= SIZE)
	hit		845	{
	target		847	flags[k] = 1;
	target		851	k += prime;
			852	}
				count++;
				}
				return count;

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

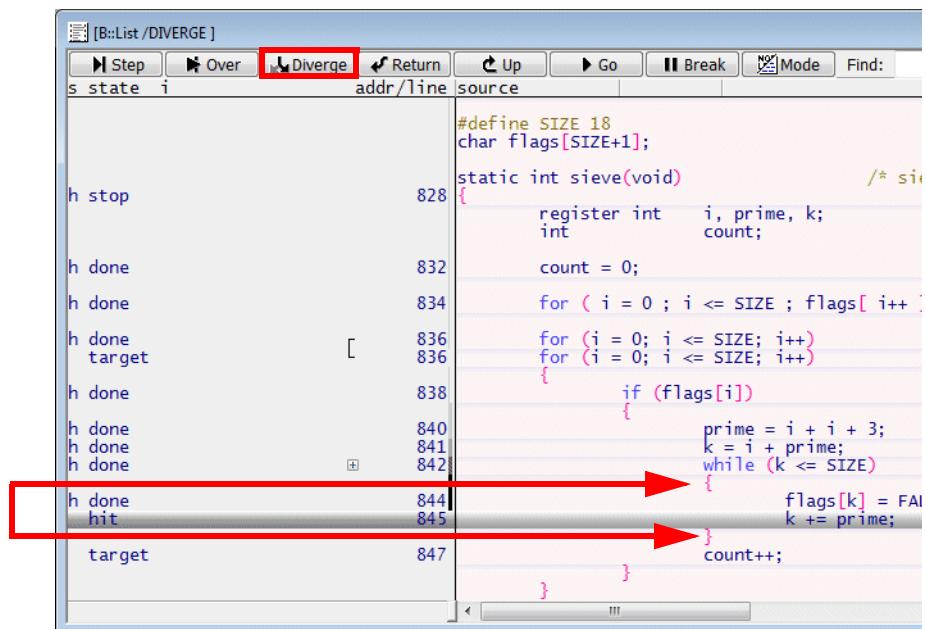


The **DIVERGE** marking is cleared when you use the **Go.direct** command without address or the **Break** command while the program execution is stopped.

## Example 2: Exit a loop.

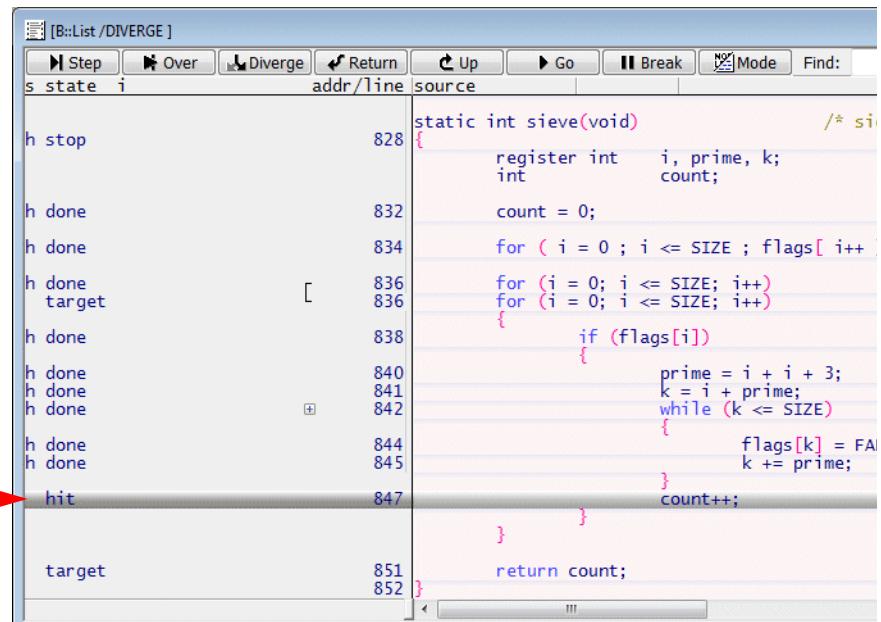
DIVERGE marking is done whenever you single step.

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



[B::List /DIVERGE ]

s	state	i	addr/line	source
				#define SIZE 18
				char flags[SIZE+1];
			828	static int sieve(void)
				/* si
	h stop			register int i, prime, k;
				int count;
			832	count = 0;
	h done		834	for (i = 0; i <= SIZE; i++)
	h done		836	for (i = 0; i <= SIZE; i++)
	h done target		836	{
	h done		838	if (flags[i])
	h done		840	prime = i + i + 3;
	h done		841	k = i + prime;
	h done		842	while (k <= SIZE)
	h done hit		844	{
			845	flags[k] = FAI
				k += prime;
	target		847	}
				count++;
				}

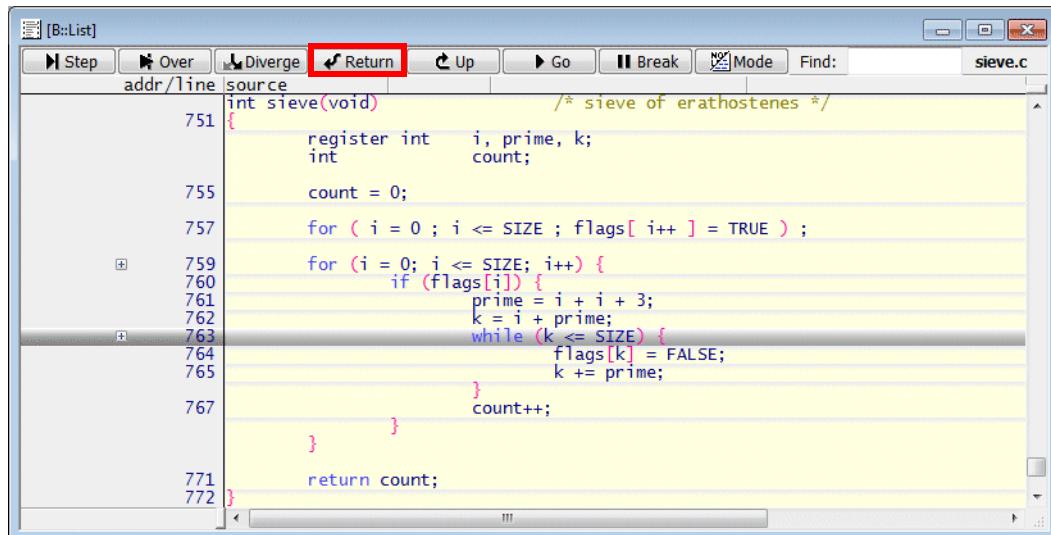


[B::List /DIVERGE ]

s	state	i	addr/line	source
				static int sieve(void)
				/* si
	h stop		828	register int i, prime, k;
				int count;
	h done		832	count = 0;
	h done		834	for (i = 0; i <= SIZE; i++)
	h done target		836	for (i = 0; i <= SIZE; i++)
	h done		836	{
	h done		838	if (flags[i])
	h done		840	prime = i + i + 3;
	h done		841	k = i + prime;
	h done		842	while (k <= SIZE)
	h done		844	{
	h done		845	flags[k] = FAI
	hit		847	k += prime;
				}
				count++;
	target		851	}
			852	return count;

**Return**

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



The screenshot shows a debugger interface with a menu bar at the top. The 'Return' button is highlighted with a red box. The main window displays a source code file named 'sieve.c' with the following content:

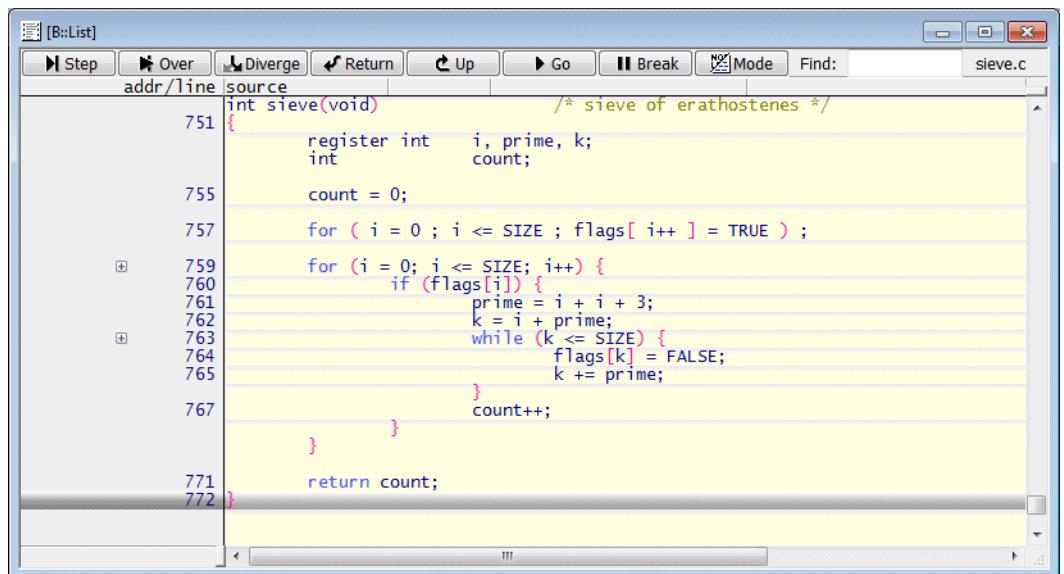
```
int sieve(void) /* sieve of erathostenes */
{
    register int i, prime, k;
    int count;

    count = 0;

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

    for ( i = 0; i <= SIZE; i++ ) {
        if (flags[i]) {
            prime = i + i + 3;
            k = i + prime;
            while (k <= SIZE) {
                flags[k] = FALSE;
                k += prime;
            }
            count++;
        }
    }

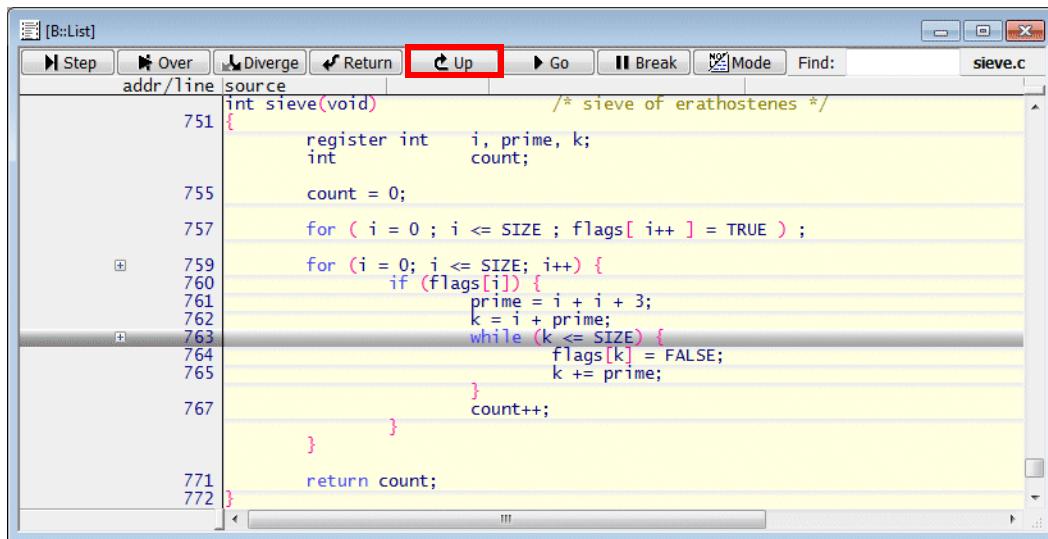
    return count;
}
```



This screenshot is identical to the one above, showing the same debugger interface and source code for 'sieve.c'. The only difference is that the 'Return' button in the menu bar is not highlighted with a red box.

**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**)

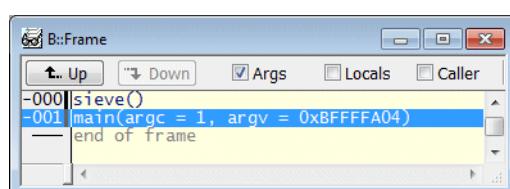
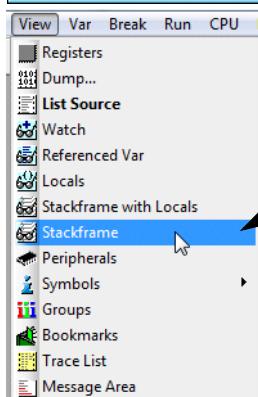


Screenshot of the B::List debugger showing the source code for sieve.c. The 'Up' button in the toolbar is highlighted with a red box. The code implements the Sieve of Eratosthenes algorithm.

```
int sieve(void) /* sieve of erathostenes */
{
    register int i, prime, k;
    int count;

    count = 0;

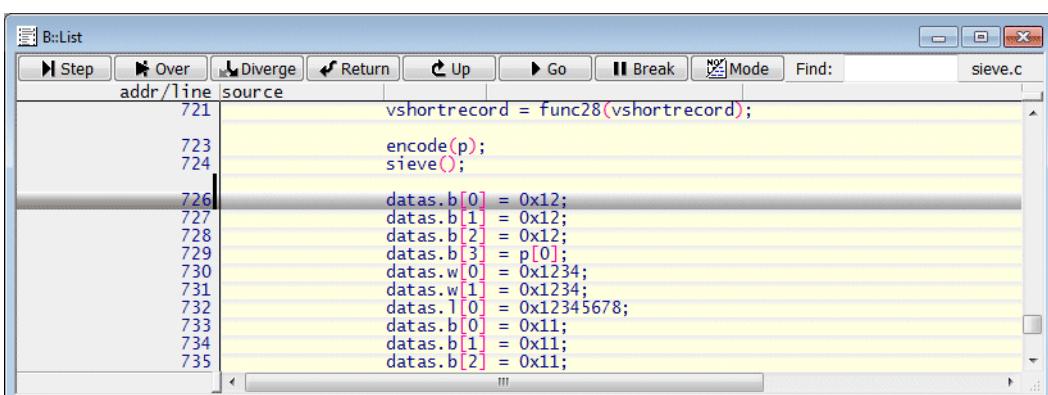
    for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
        for ( i = 0; i <= SIZE; i++ ) {
            if (flags[i]) {
                prime = i + i + 3;
                k = i + prime;
                while (k <= SIZE) {
                    flags[k] = FALSE;
                    k += prime;
          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



Display the HLL stack to see the function nesting

Screenshot of the B::Frame debugger showing the stack frame list. The 'sieve()' frame is selected.

Frame	Address	Function
-000		sieve()
-001		main(argc = 1, argv = 0xBFFFFA04)
		end of frame



Screenshot of the B::List debugger showing the source code for a different function. The 'Up' button in the toolbar is visible.

```
vshortrecord = func28(vshortrecord);
encode(p);
sieve();

datas.b[0] = 0x12;
datas.b[1] = 0x12;
datas.b[2] = 0x12;
datas.b[3] = p[0];
datas.w[0] = 0x1234;
datas.w[1] = 0x1234;
datas.l[0] = 0x12345678;
datas.b[0] = 0x11;
datas.b[1] = 0x11;
datas.b[2] = 0x11;
```

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

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

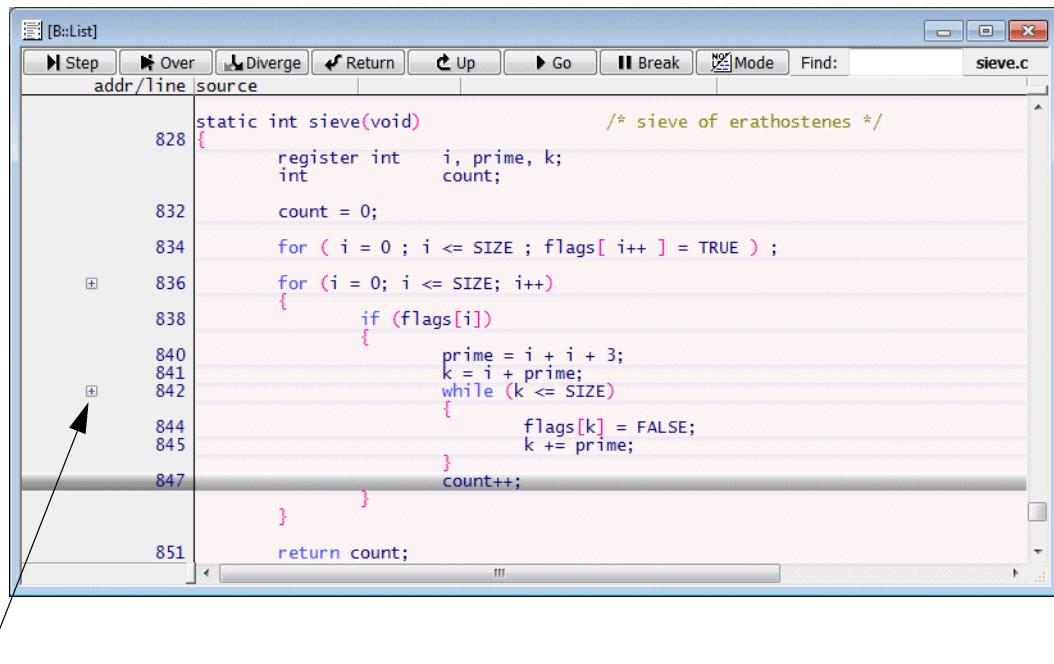
; step until the variable mstatic2 is larger the 3 and
; the variable flags[3] is unequal 1
Var.Step.Till (mstatic2>3)&&(flags[3]!=1)
```

# 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.



```
static int sieve(void)          /* sieve of erathostenes */
{
    register int i, prime, k;
    int count;

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

tree button

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

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

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

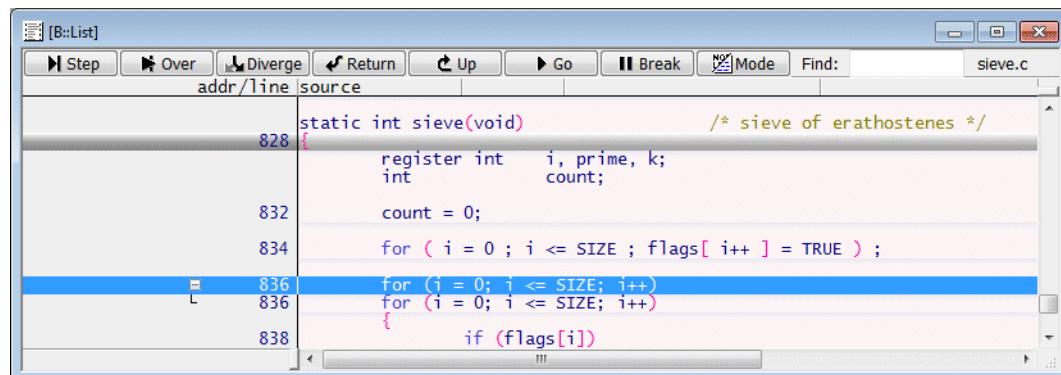
#### List.HLL

Display source listing, display HLL code lines only.

#### List.Mix /Track

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

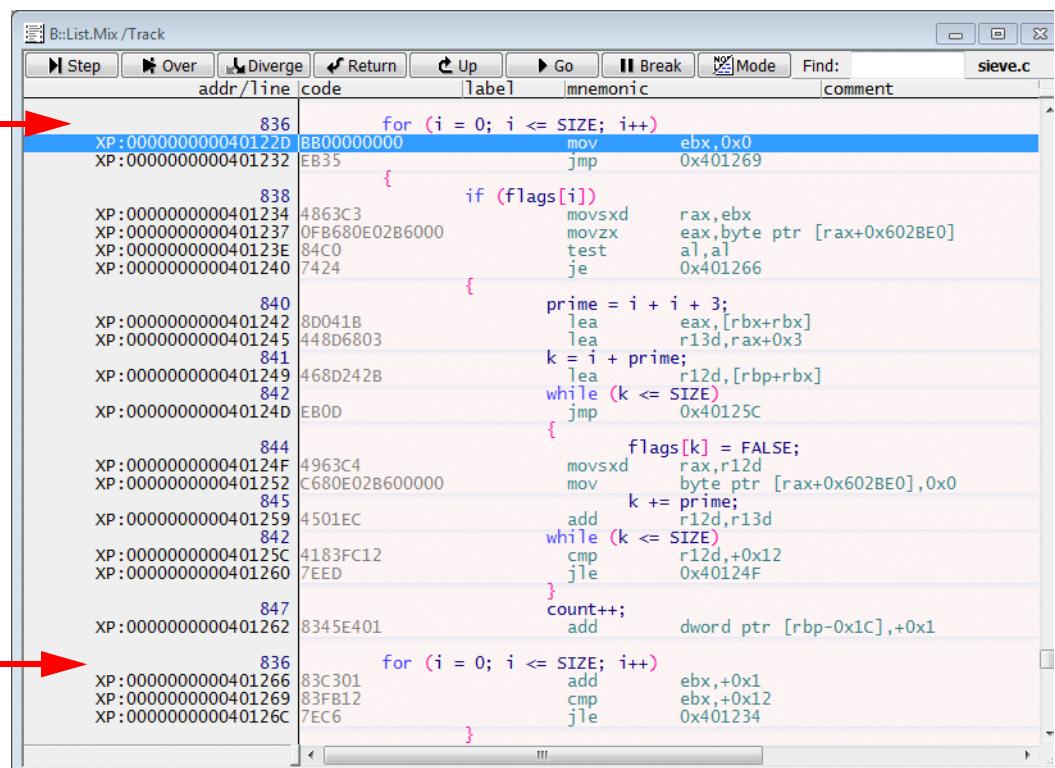
The blue cursor in the MIX mode display follows the cursor movement of the HLL mode display (Track option).



```
static int sieve(void)          /* sieve of erathostenes */
{
    register int i, prime, k;
    int count;

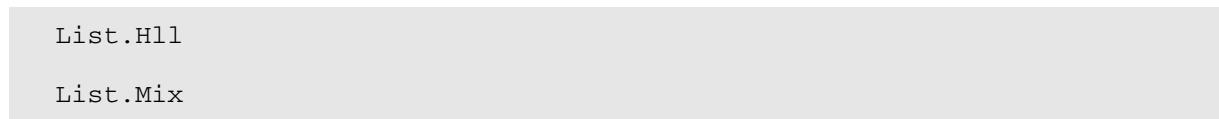
    count = 0;

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



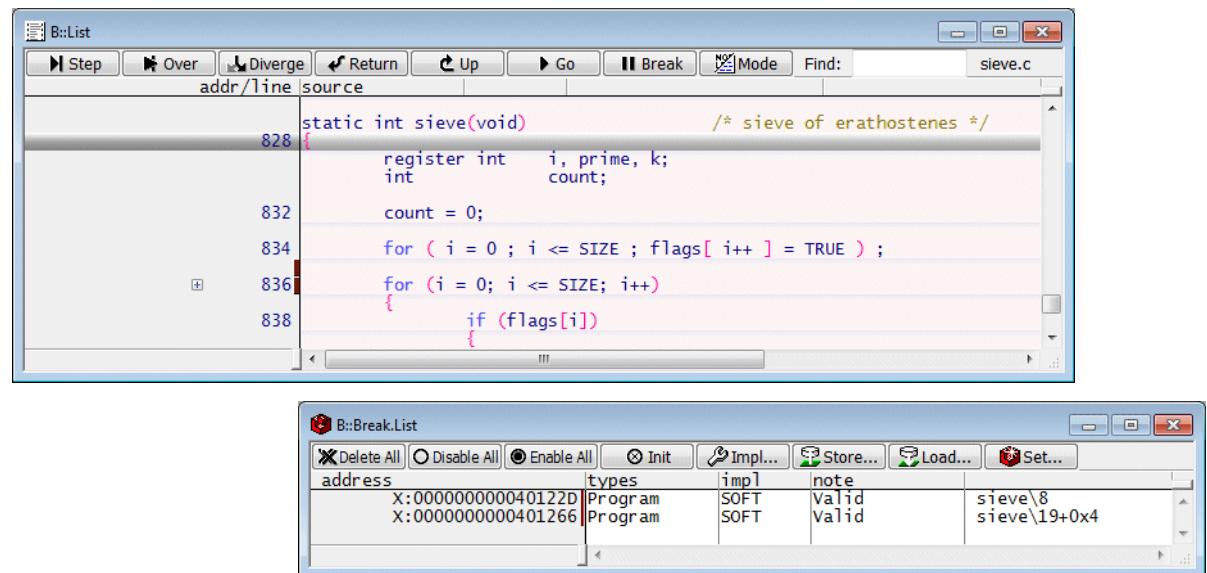
addr/line	code	label	mnemonic	comment
836	for (i = 0; i <= SIZE; i++)			
XP:000000000040122D	BB000000000		mov	ebx,0x0
XP:0000000000401232	EB35		jmp	0x401269
838	{			
XP:0000000000401234	4863C3		movsx	rax,ebx
XP:0000000000401237	0FB680E02B6000		movzx	eax,byte ptr [rax+0x602BE0]
XP:000000000040123E	84C0		test	al,al
XP:0000000000401240	7424		je	0x401266
840				
XP:0000000000401242	8D041B		lea	eax,[rbx+rbx]
XP:0000000000401245	448D6803		lea	r13d,rax+0x3
841				
XP:0000000000401249	468D242B		k = i + prime;	r12d,[rbp+rbx]
842				
XP:000000000040124D	EB0D		while (k <= SIZE)	jmp 0x40125C
844				
XP:000000000040124F	4963C4		flags[k] = FALSE;	movsx rax,r12d
XP:0000000000401252	C680E02B600000		mov	byte ptr [rax+0x602BE0],0x0
845				
XP:0000000000401259	4501EC		k += prime;	add r12d,r13d
842				
XP:000000000040125C	4183FC12		while (k <= SIZE)	cmp r12d,+0x12
XP:0000000000401260	7EED		jle	0x40124F
847				
XP:0000000000401262	8345E401		count++;	add dword ptr [rbp-0x1C],+0x1
836	for (i = 0; i <= SIZE; i++)			
XP:0000000000401266	83C301		add	ebx,+0x1
XP:0000000000401269	83FB12		cmp	ebx,+0x12
XP:000000000040126C	7EC6		jle	0x401234

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



Please be aware of the following:

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

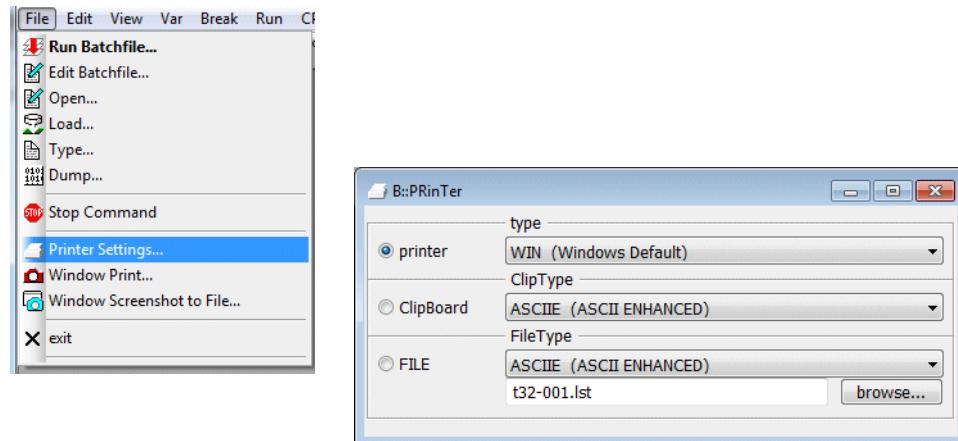


## Settings

TRACE32 PowerView supports the following ways to document your results:

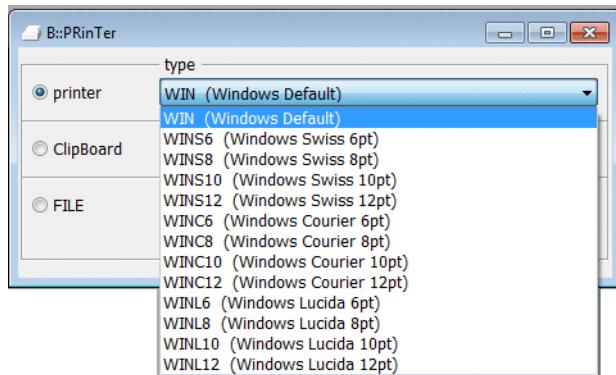
- Sending them to a printer
- Transferring them via the clipboard to other applications
- Saving them to a file

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



## Print

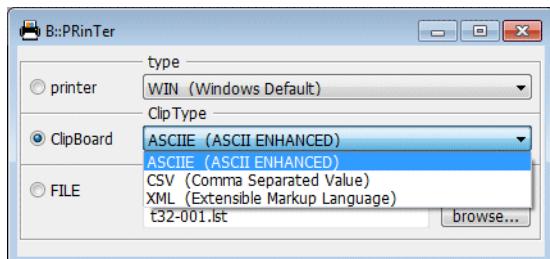
---



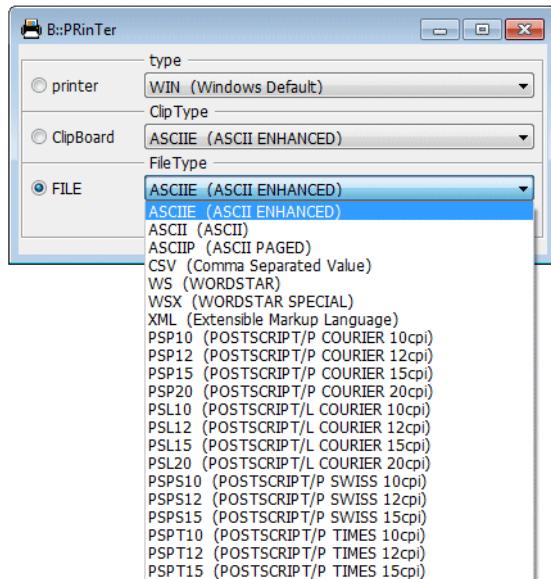
If you send your results to a printer, you can select Windows Default or other font sizes and families supported by your printer.

## Clipboard

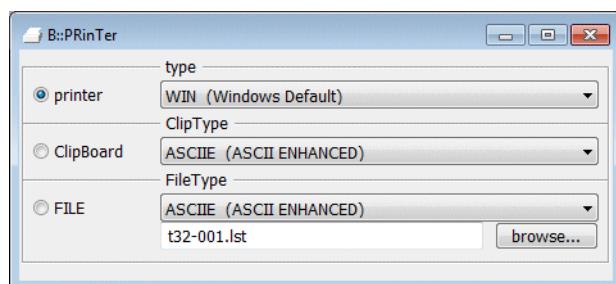
---



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



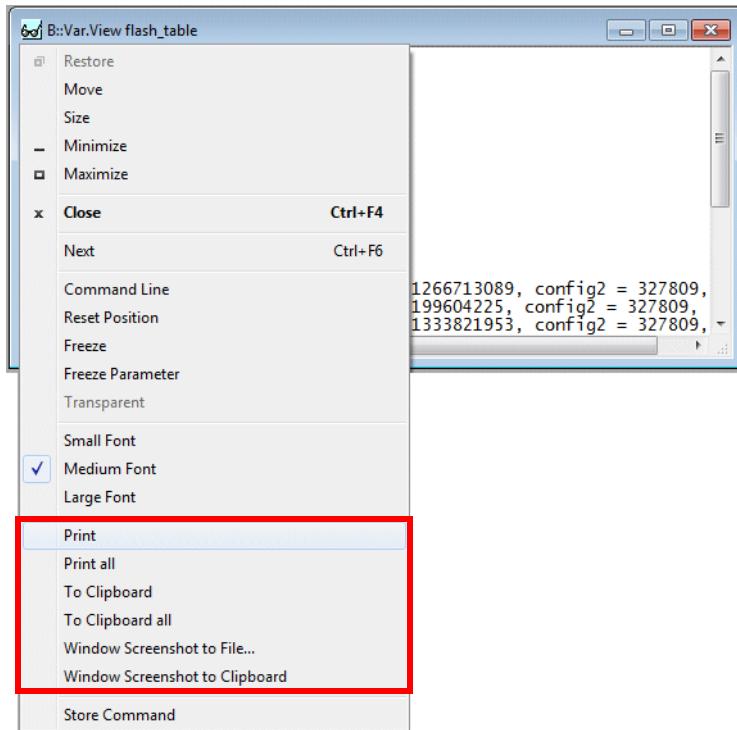
If you save your result to a file, you can choose between various ASCII formats, Comma-Separated Value, HTML and various POSTSCRIPT formats.



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.

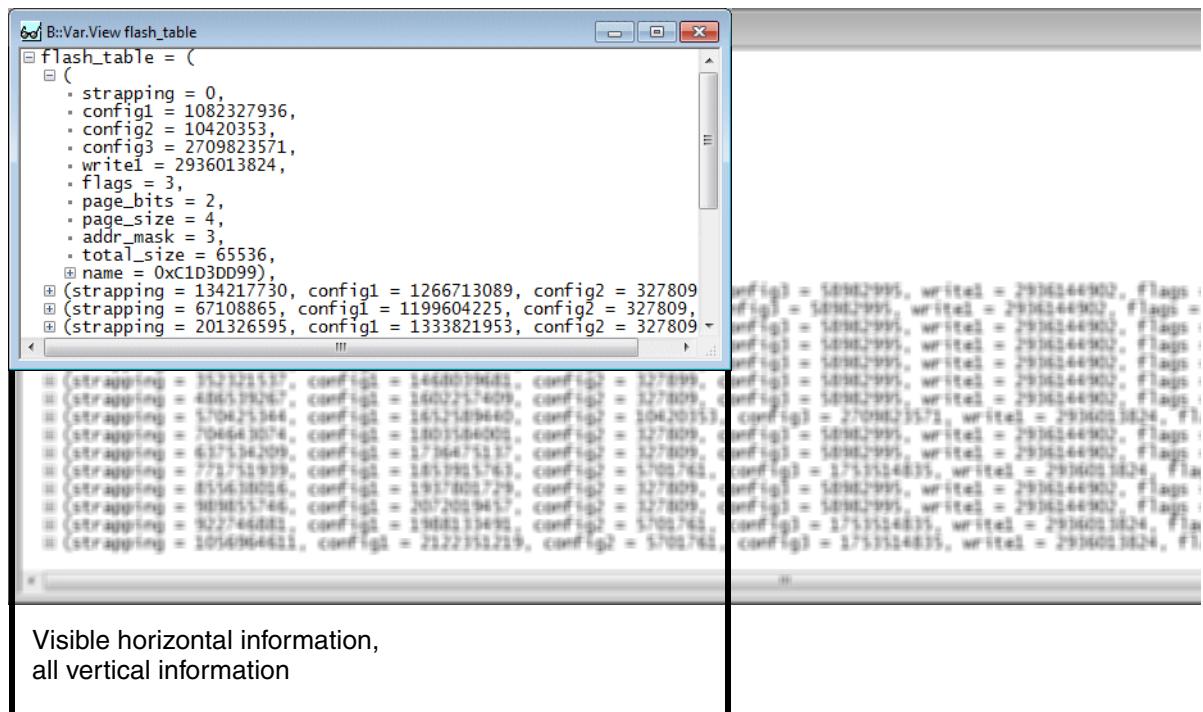
# Quick Output



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

# Advanced Output

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

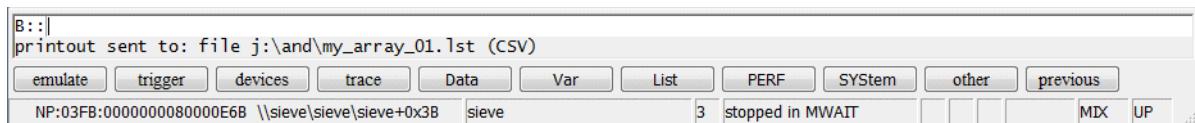
**WinPrint.<command>**

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
```