Training Basic Debugging

Training Basic Debugging

Training Basic Debugging

Version 06-Jun-2024

System Concept

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

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

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

On-chip Debug Interface

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

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

Debug Features

Depending on the processor architecture different debug features are available.

Debug features provided by all processor architectures:

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

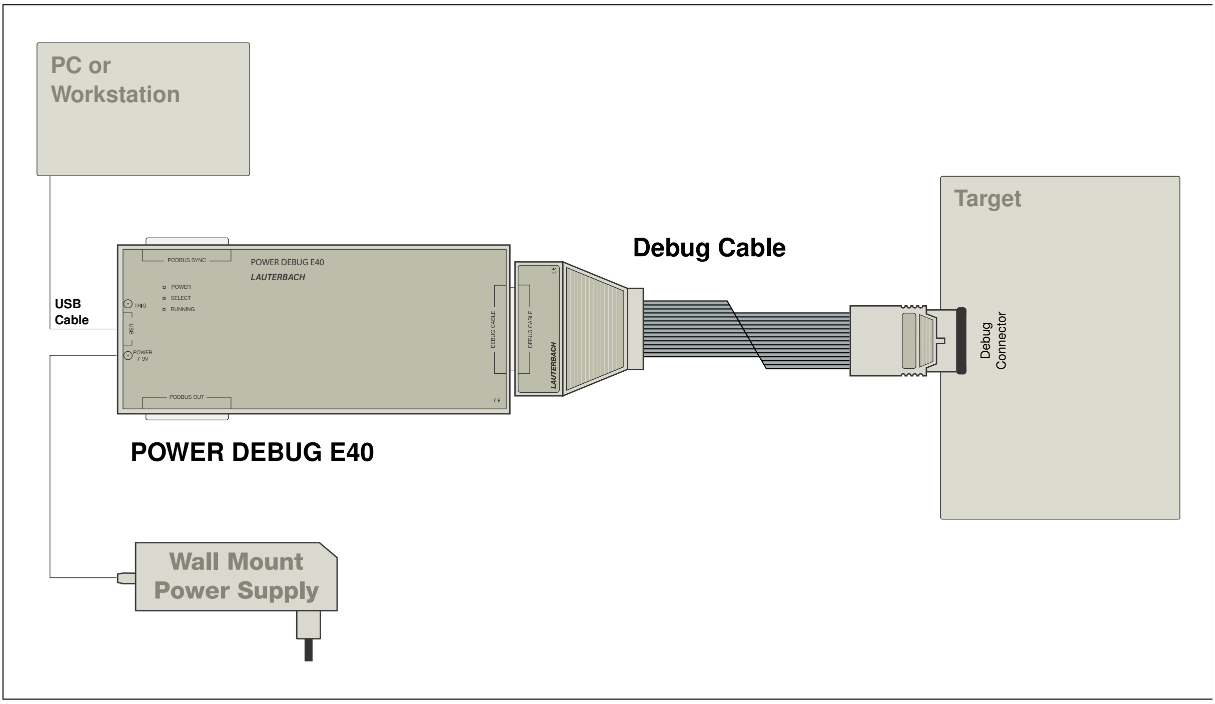
Debug features specific for a processor architecture:

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

The TRACE32 debugger hardware always consists of:

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

Debug Only Modules



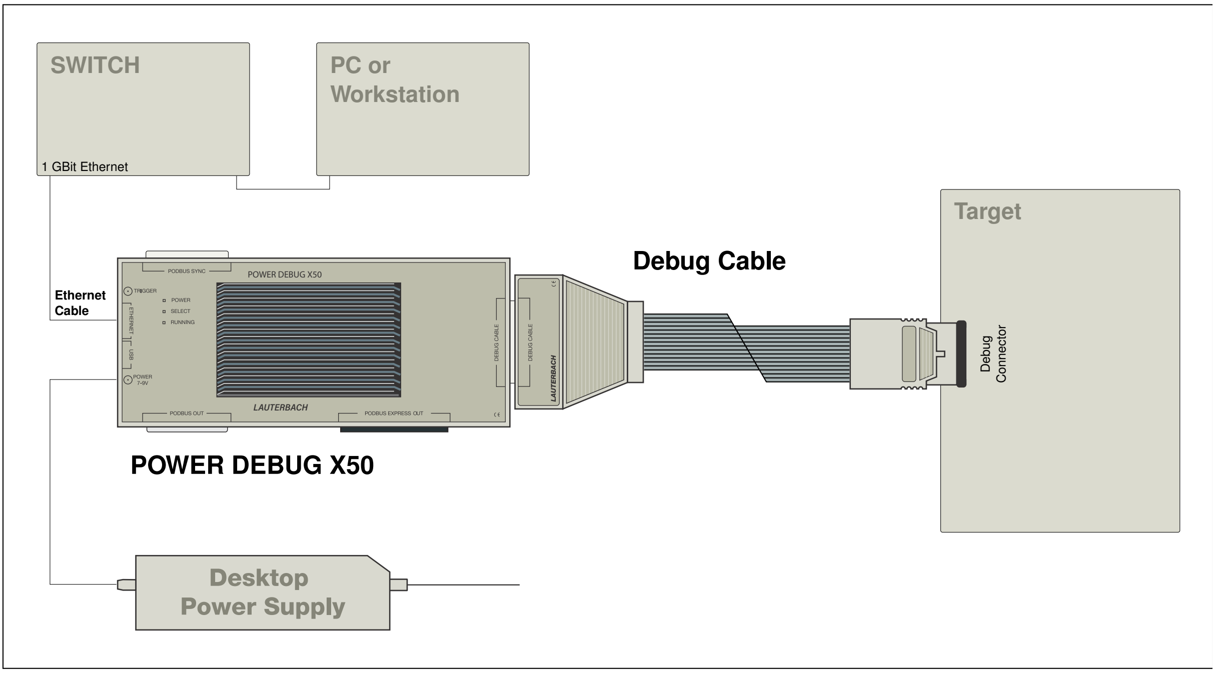
Current module:

- POWER DEBUG E40

Deprecated modules:

- POWER DEBUG INTERFACE / USB 3
- POWER DEBUG INTERFACE / USB 2

Debug Modules with Option for Off-chip Trace Extension



Current module:

- POWER DEBUG X50

Deprecated modules:

- POWER DEBUG PRO (USB 3 and 1 GBit Ethernet)
- POWER DEBUG II (USB 2 and 1 GBit Ethernet)
- POWER DEBUG / ETHERNET (USB 2 and 100 MBit Ethernet)

On-chip Debug Interface plus On-chip Trace Buffer

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

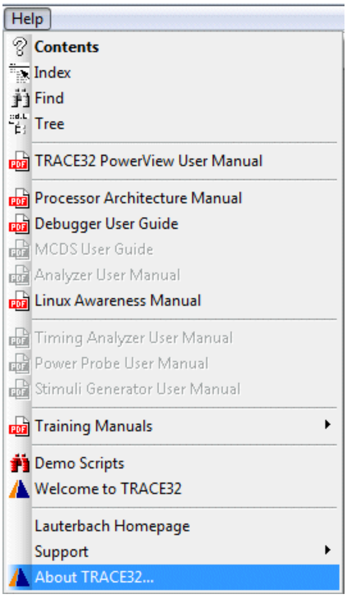
On-chip Trace Features

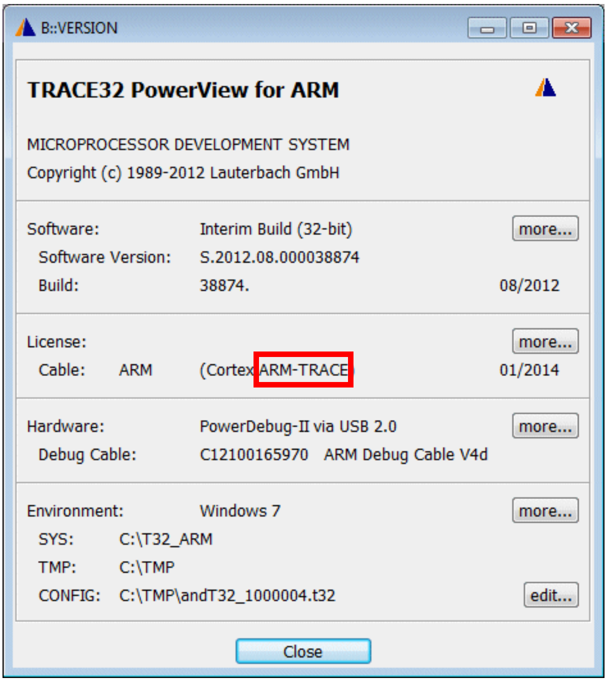
The on-chip trace buffer can store information:

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

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

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





TRACE32 PowerView for ARM

MICROPROCESSOR DEVELOPMENT SYSTEM
Copyright (c) 1989-2012 Lauterbach GmbH

Software: Interim Build (32-bit)
Software Version: S.2012.08.000038874
Build: 38874. 08/2012

License:
Cable: ARM (Cortex ARM-TRACE) 01/2014

Hardware: PowerDebug-II via USB 2.0
Debug Cable: C12100165970 ARM Debug Cable V4d

Environment: Windows 7
SYS: C:\T32_ARM
TMP: C:\TMP
CONFIG: C:\TMP\andT32_1000004.t32

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

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

On-chip Debug Interface plus Trace Port

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

Off-chip Trace Features

The trace port exports in real-time trace information:

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

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

- **"Training Arm CoreSight ETM Tracing"** (training_arm_etm.pdf)
- **"Training Cortex-M Tracing"** (training_cortexm_etm.pdf)
- **"Training AURIX Tracing"** (training_aurix_trace.pdf)
- **"Training Hexagon ETM Tracing"** (training_hexagon_etm.pdf)

NEXUS Interface

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

NEXUS Features

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

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

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

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

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

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

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

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

The display and the evaluation of the trace information is described in **“Training Nexus Tracing”** (training_nexus.pdf)**.**

Starting a TRACE32 PowerView Instance

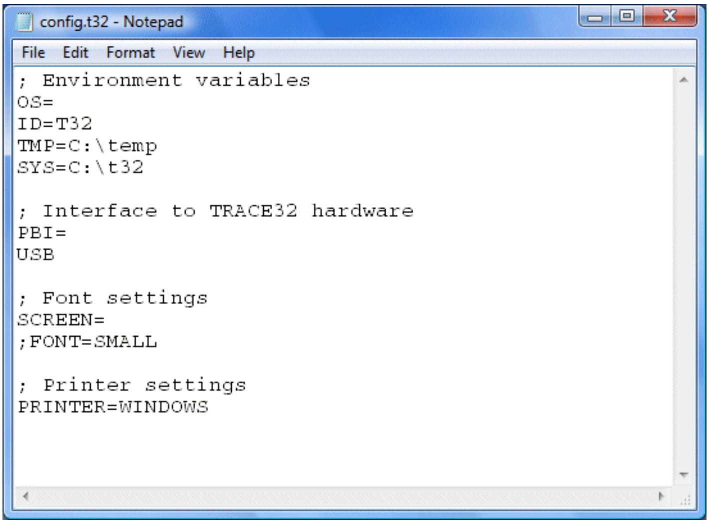
Basic TRACE32 PowerView Parameters

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

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

Configuration File

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



```
; 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.

Parameter	Syntax	Description
Host interface	PBI= *<host_interface>* PBI=ICD *<host_interface>*	Host interface type of TRACE32 tool hardware (USB or ethernet) Full parameter syntax which is not in use.

Environment variables	OS=
	ID=*<identifier>*
	TMP=*<temp_directory>*
	SYS=*<system_directory>*
	HELP=*<help_directory>*
		(ID) Prefix for all files which are saved by the TRACE32 PowerView instance into the TMP directory
		
		(TMP) Temporary directory used by the TRACE32 PowerView instance (*)
		
		(SYS) System directory for all TRACE32 files
		
		(HELP) Directory for the TRACE32 help PDFs (**)

Printer definition	PRINTER=WINDOWS	All standard Windows printer can be used from TRACE32 PowerView
License file	LICENSE=*<license_directory>*	Directory for the TRACE32 license file (not required for new tools)

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

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

Configuration File for USB

Single debugger hardware module connected via USB:

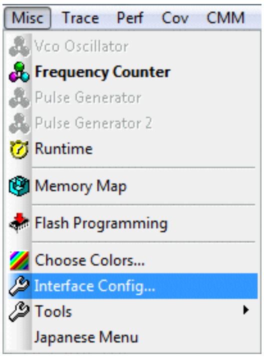
```
; Host interface
PBI=
USB

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

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

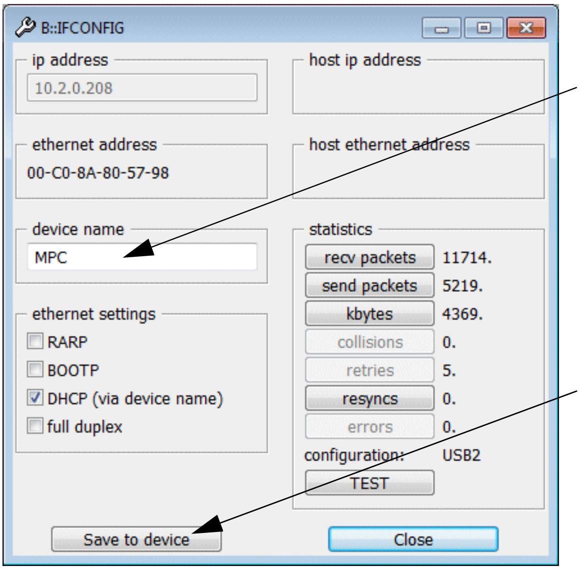
Multiple debugger hardware modules connected via USB:

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

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

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

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

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





ip address: 10.2.0.208

host ip address:

ethernet address: 00-C0-8A-80-57-98

host ethernet address:

device name: MPC

ethernet settings:

- ☐ RARP
- ☐ BOOTP
- ☑ DHCP (via device name)
- ☐ full duplex

statistics:

recv packets	11714.
send packets	5219.
kbytes	4369.
collisions	0.
retries	5.
resyncs	0.
errors	0.
configuration:	USB2

Enter device name

Save device name to debugger hardware module

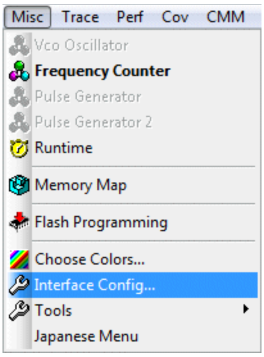
IFCONFIG	Dialog to assign USB device name Please be aware that USB device names are case-sensitive

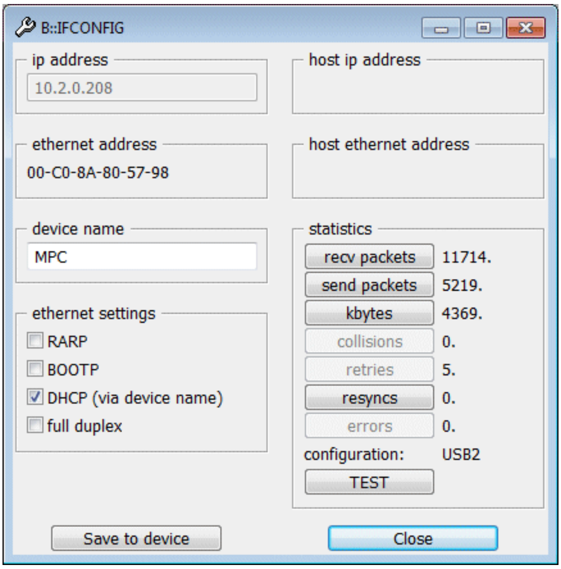
TRACE32 allows to communicate with a POWER DEBUG INTERFACE USB from a remote PC. For an example, see **“Example: Remote Control for POWER DEBUG INTERFACE / USB”** in TRACE32 Installation Guide, page 56 (installation.pdf).

Configuration File for Ethernet

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

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

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

Ethernet Configuration and Operation Profile




B::IFCONFIG

ip address: 10.2.0.208

host ip address:

ethernet address: 00-C0-8A-80-57-98

host ethernet address:

device name: MPC

ethernet settings

☐ RARP
☐ BOOTP
☑ DHCP (via device name)
☐ full duplex

statistics

recv packets 11714.
send packets 5219.
kbytes 4369.
collisions 0.
retries 5.
resyncs 0.
errors 0.
configuration: USB2

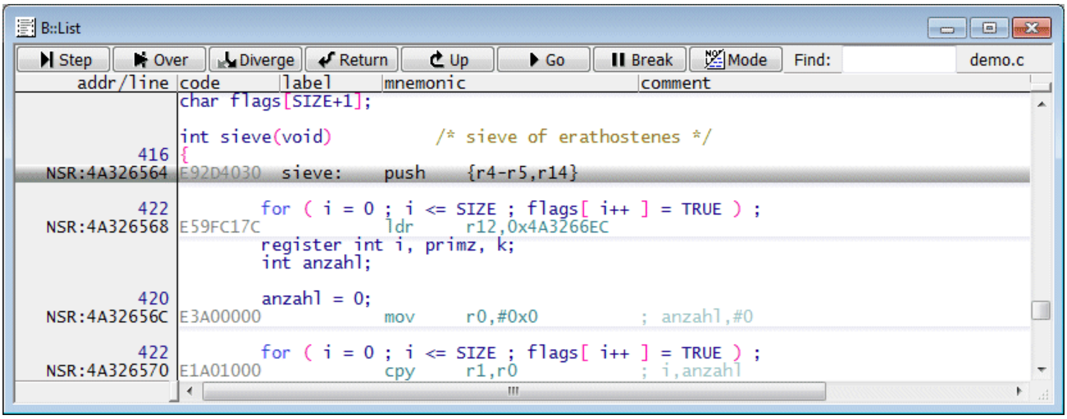
IFCONFIG Dialog to display and change information for the Ethernet interface

Additional Parameters

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

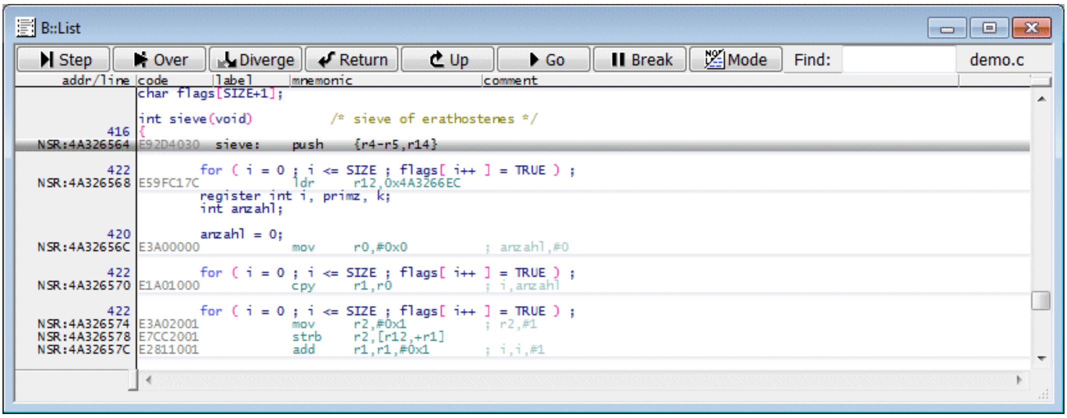
```
; Screen settings
SCREEN=
FONT=SMALL                    ; Use small fonts
```

Display with normal font:



```
    addr/line code      label    mnemonic               comment
              char flags[SIZE+1];

              int sieve(void)          /* sieve of erathostenes */
          416 {
NSR:4A326564 E92D4030  sieve:    push     {r4-r5,r14}

          422         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
NSR:4A326568 E59FC17C            ldr      r12,0x4A3266EC
                      register int i, primz, k;
                      int anzahl;

          420         anzahl = 0;
NSR:4A32656C E3A00000            mov      r0,#0x0             ; anzahl,#0

          422         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
NSR:4A326570 E1A01000            cpy      r1,r0               ; i,anzahl
```

Display with small font:

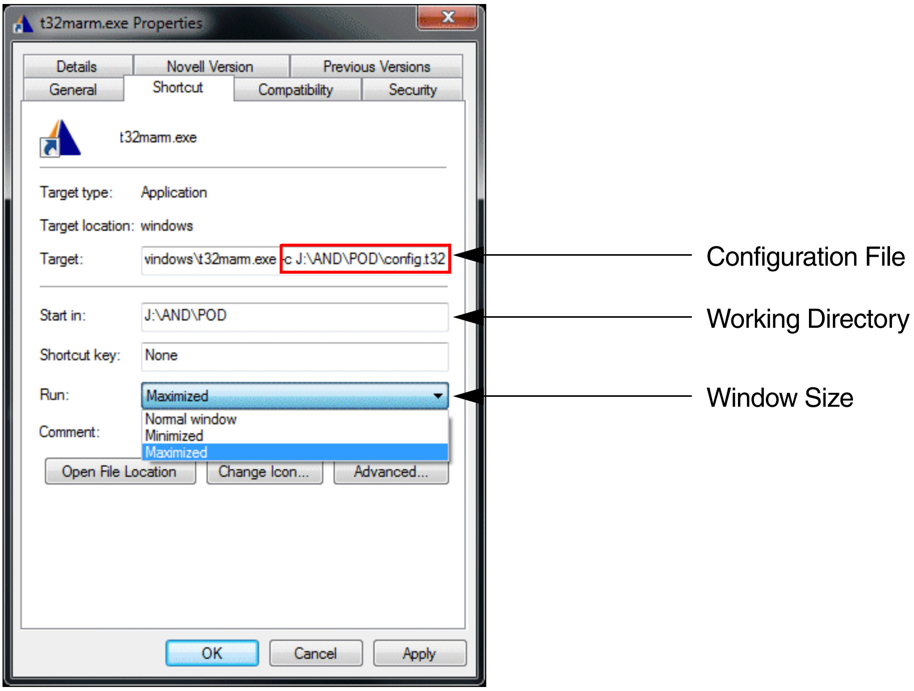


```
    addr/line code      label    mnemonic               comment
              char flags[SIZE+1];

              int sieve(void)          /* sieve of erathostenes */
          416 {
NSR:4A326564 E92D4030  sieve:    push     {r4-r5,r14}

          422         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
NSR:4A326568 E59FC17C            ldr      r12,0x4A3266EC
                      register int i, primz, k;
                      int anzahl;

          420         anzahl = 0;
NSR:4A32656C E3A00000            mov      r0,#0x0             ; anzahl,#0

          422         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
NSR:4A326570 E1A01000            cpy      r1,r0               ; i,anzahl

          422         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
NSR:4A326574 E3A02001            mov      r2,#0x1             ; r2,#1
NSR:4A326578 E7CC2001            strb     r2,[r12,+r1]
NSR:4A32657C E2811001            add      r1,r1,#0x1          ; i,i,#1
```

Application Properties (Windows only)

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

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



t32marm.exe Properties

Target type: Application

Target location: windows

Target: windows\t32marm.exe -c J:\AND\POD\config.t32

Start in: J:\AND\POD

Shortcut key: None

Run: Maximized

Comment:

Configuration File

Working Directory

Window Size

Definition of the Configuration File

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

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

Definition of a Working Directory

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

PWD	TRACE32 command to display the current working directory

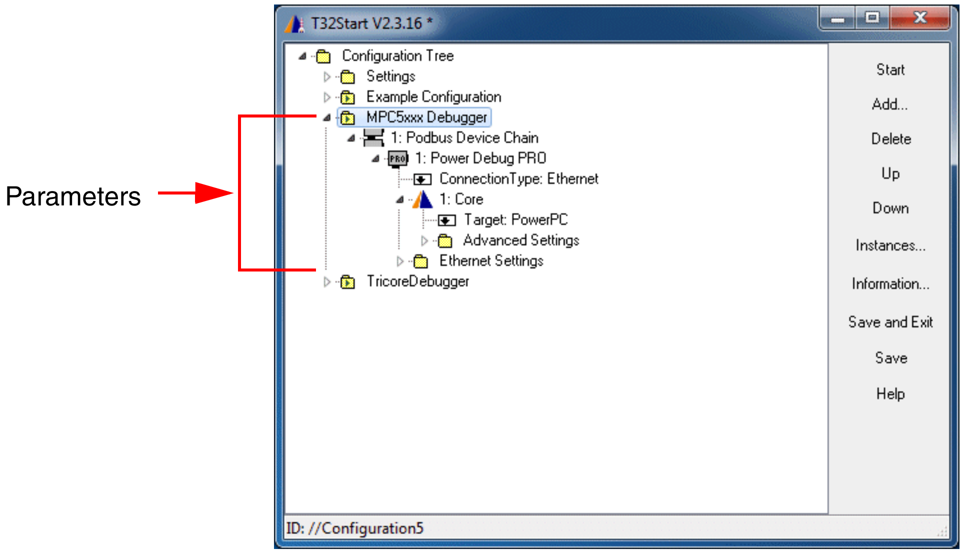
Definition of the Window Size for TRACE32 PowerView

You can choose between Normal window, Minimized and Maximized.

Configuration via T32Start (Windows only)

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

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

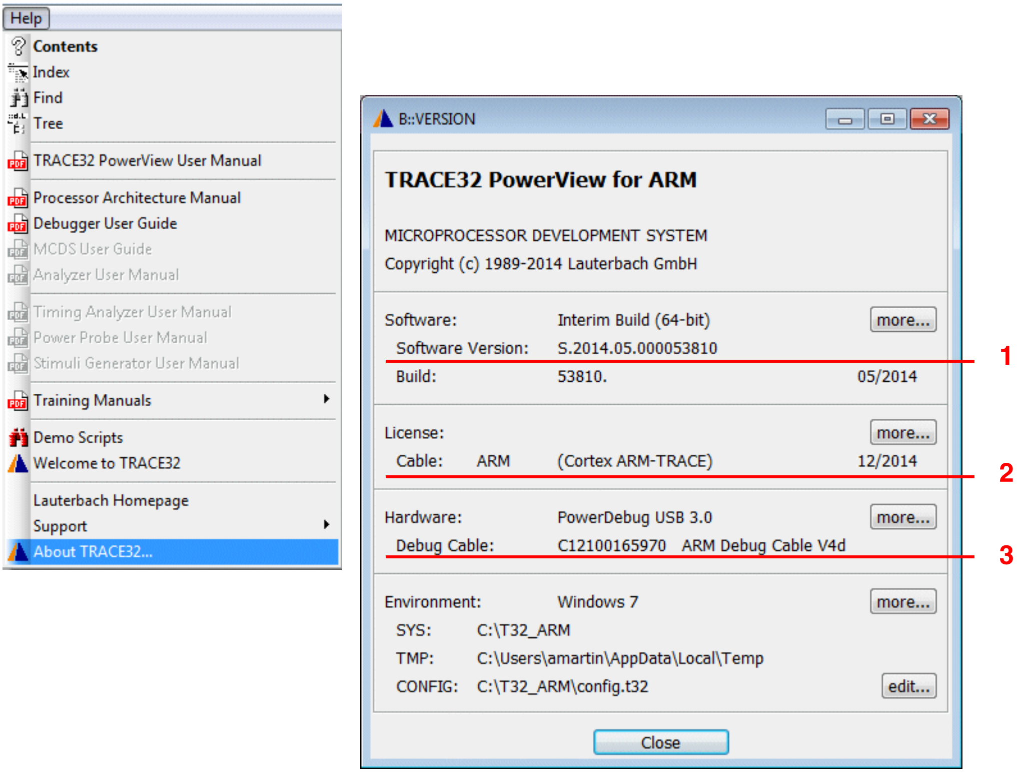


```
Configuration Tree
  Settings
  Example Configuration
  MPC5xxx Debugger
    1: Podbus Device Chain
      1: Power Debug PRO
        ConnectionType: Ethernet
        1: Core
          Target: PowerPC
          Advanced Settings
        Ethernet Settings
  TricoreDebugger
ID: //Configuration5
```

About TRACE32

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

Version Information



TRACE32 PowerView for ARM

MICROPROCESSOR DEVELOPMENT SYSTEM
Copyright (c) 1989-2014 Lauterbach GmbH

Software: Interim Build (64-bit)
Software Version: S.2014.05.000053810
Build: 53810. 05/2014

License:
Cable: ARM (Cortex ARM-TRACE) 12/2014

Hardware: PowerDebug USB 3.0
Debug Cable: C12100165970 ARM Debug Cable V4d

Environment: Windows 7
SYS: C:\T32_ARM
TMP: C:\Users\amartin\AppData\Local\Temp
CONFIG: C:\T32_ARM\config.t32

The VERSION window informs you about:

1. The version of the TRACE32 software.

2. The debug licenses programmed into the debug cable and the expiration date of your software warranty respectively the expiration date of your software warranty.

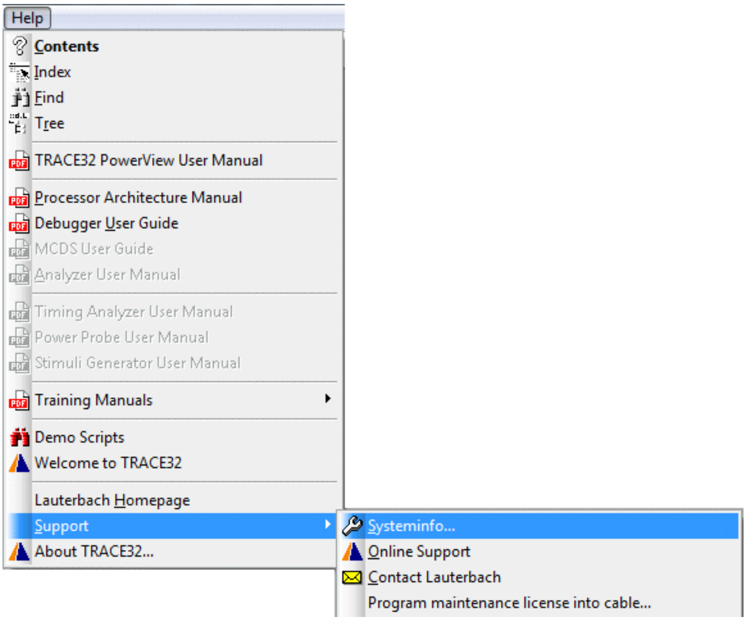
3. The serial number of the debug cable.

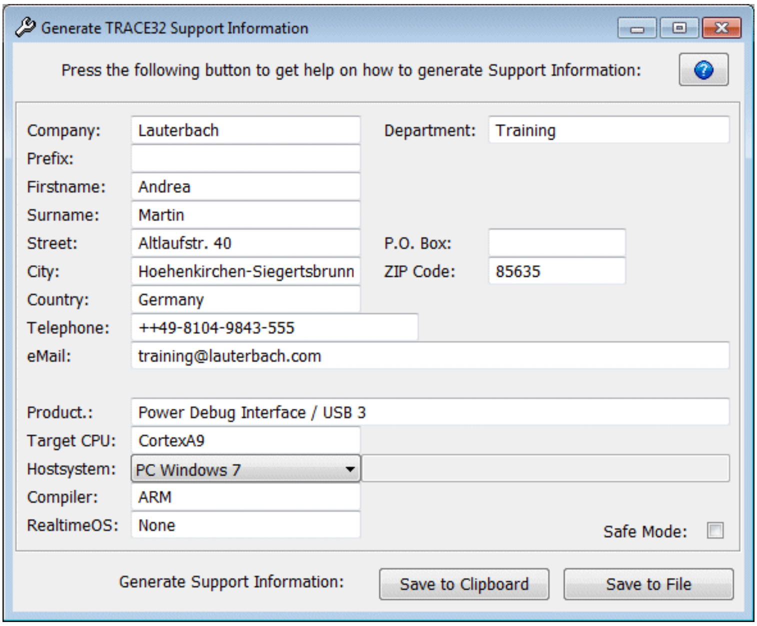
VERSION.view	Display the VERSION window.
VERSION.HARDWARE	Display more details about the TRACE32 hardware modules.
VERSION.SOFTWARE	Display more details about the TRACE32 software.

Prepare Full Information for a Support Email

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

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




Generate TRACE32 Support Information

Press the following button to get help on how to generate Support Information:

Company:	Lauterbach	Department:	Training
Prefix:			
Firstname:	Andrea		
Surname:	Martin		
Street:	Altlaufstr. 40	P.O. Box:	
City:	Hoehenkirchen-Siegertsbrunn	ZIP Code:	85635
Country:	Germany		
Telephone:	++49-8104-9843-555		
eMail:	training@lauterbach.com		
Product.:	Power Debug Interface / USB 3		
Target CPU:	CortexA9		
Hostsystem:	PC Windows 7		
Compiler:	ARM		
RealtimeOS:	None	Safe Mode:	□

Generate Support Information:

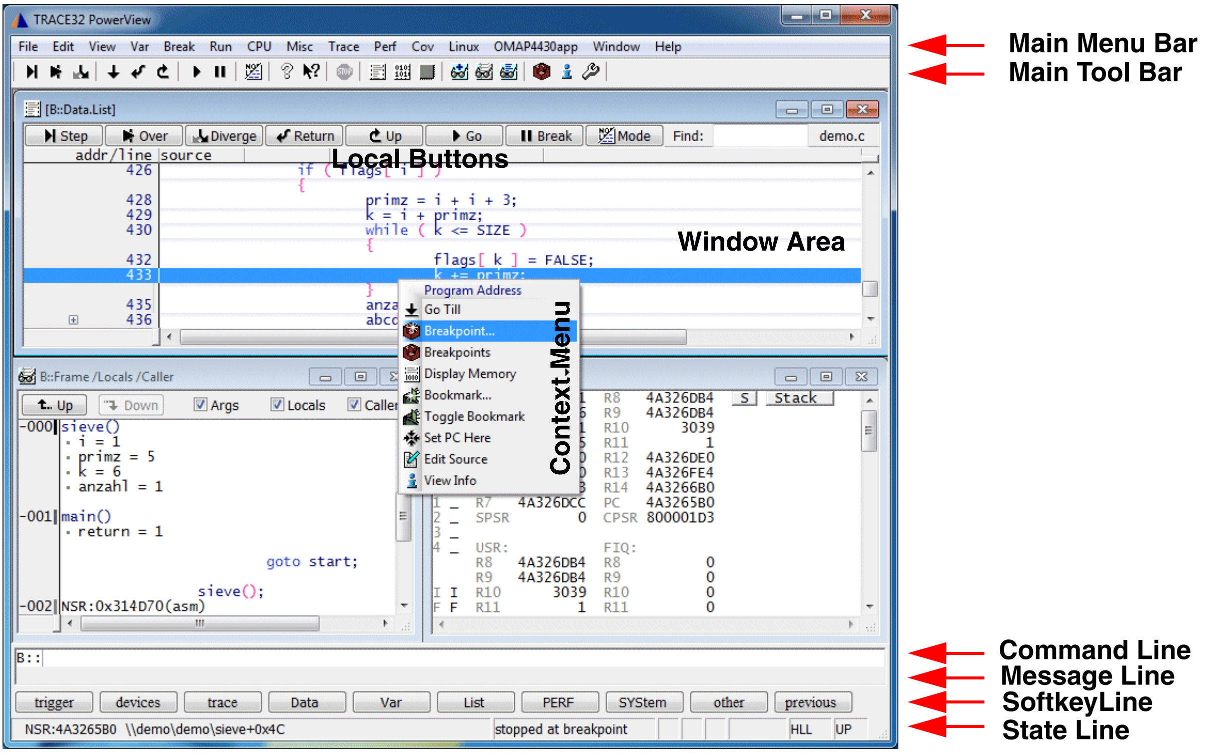
2. Preferred: click **Save to File**, and send the system information as an attachment to your e-mail.

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

Establish your Debug Session

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

TRACE32 PowerView Components



```
426          if ( flags[ i ] )
             {
428               primz = i + i + 3;
429               k = i + primz;
430               while ( k <= SIZE )
                  {
432                    flags[ k ] = FALSE;
433                    k += primz;
                  }
435               anza
436               abcc
```

```
-000 sieve()
      i = 1
      primz = 5
      k = 6
      anzahl = 1

-001 main()
      return = 1

                        goto start;

                  sieve();
-002 NSR:0x314D70(asm)
```

```
R8    4A326DB4
R9    4A326DB4
R10       3039
R11          1
R12   4A326DE0
R13   4A326FE4
R14   4A3266B0
R7   4A326DCC   PC    4A3265B0
SPSR        0   CPSR  800001D3

USR:            FIQ:
R8   4A326DB4   R8           0
R9   4A326DB4   R9           0
R10      3039   R10          0
R11         1   R11          0
```

```
B::
```

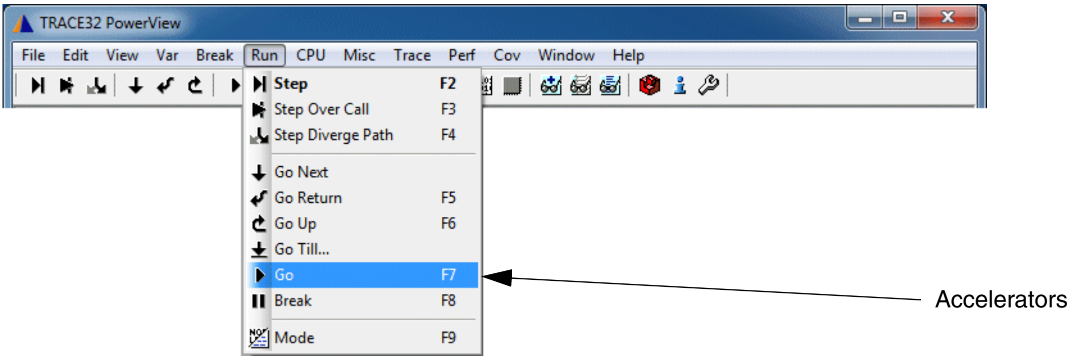
NSR:4A3265B0 \\demo\demo\sieve+0x4C stopped at breakpoint HLL UP

	The structure of the menu bar and the tool bar are defined by the file **`t32.men`** which is located in the TRACE32 system directory. TRACE32 allows you to modify the menu bar and the tool bar so they will better fit your requirements. Refer to **“Training Menu Programming”** (training_menu.pdf) for details.

Main Menu Bar and Accelerators

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

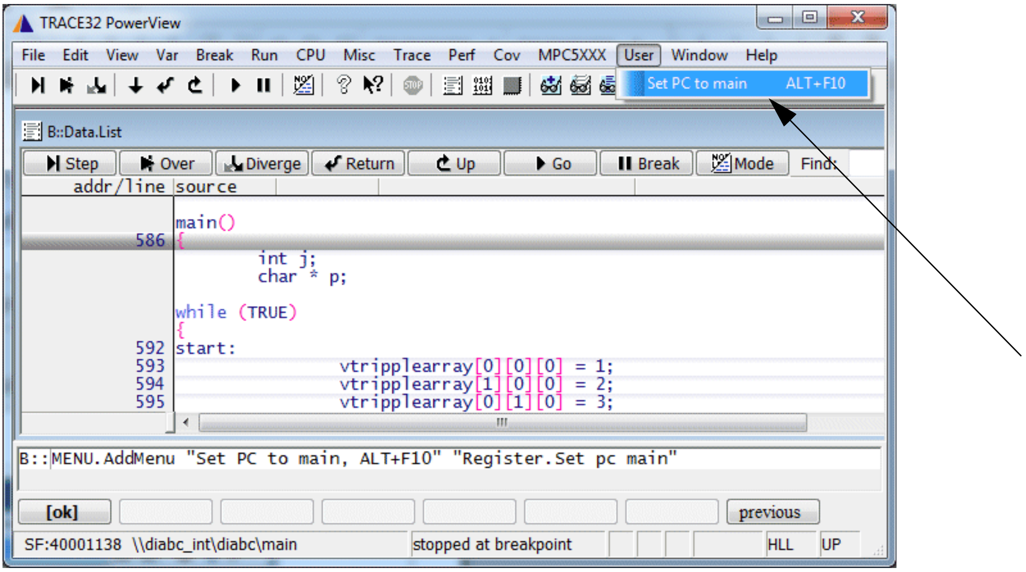
For often used commands accelerators are defined.



A user specific menu can be defined very easily:

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

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

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



```
         main()
586      {
                 int j;
                 char * p;

         while (TRUE)
         {
592      start:
593                      vtripplearray[0][0][0] = 1;
594                      vtripplearray[1][0][0] = 2;
595                      vtripplearray[0][1][0] = 3;
```

```
B::MENU.AddMenu "Set PC to main, ALT+F10" "Register.Set pc main"
```

SF:40001138 \\diabc_int\diabc\main stopped at breakpoint HLL UP

User Menu

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

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

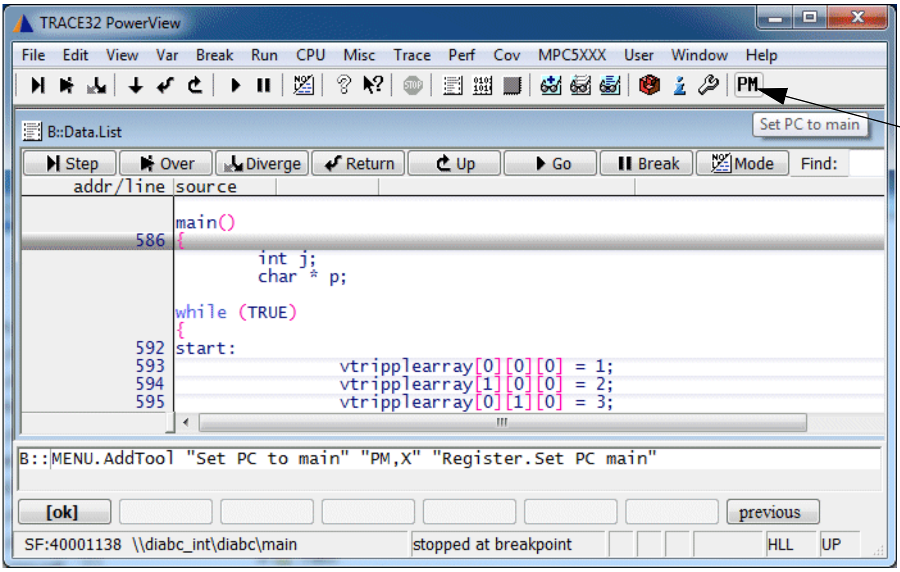
Main Tool Bar

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

The user can add his own buttons very easily:

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

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

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


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX User Window Help
PM
Set PC to main
B::Data.List
Step Over Diverge Return Up Go Break Mode Find:
addr/line source
[ok]
previous

```
main()
{
        int j;
        char * p;

while (TRUE)
{
start:
                vtripplearray[0][0][0] = 1;
                vtripplearray[1][0][0] = 2;
                vtripplearray[0][1][0] = 3;
```

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

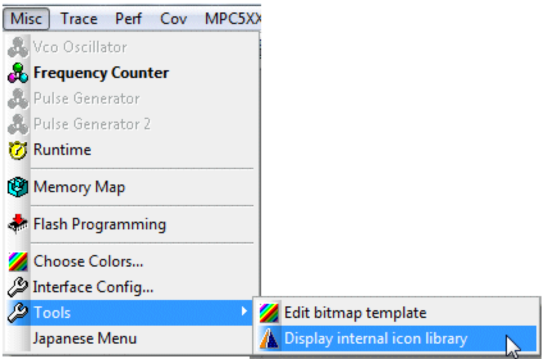
SF:40001138 \\diabc_int\diabc\main stopped at breakpoint HLL UP

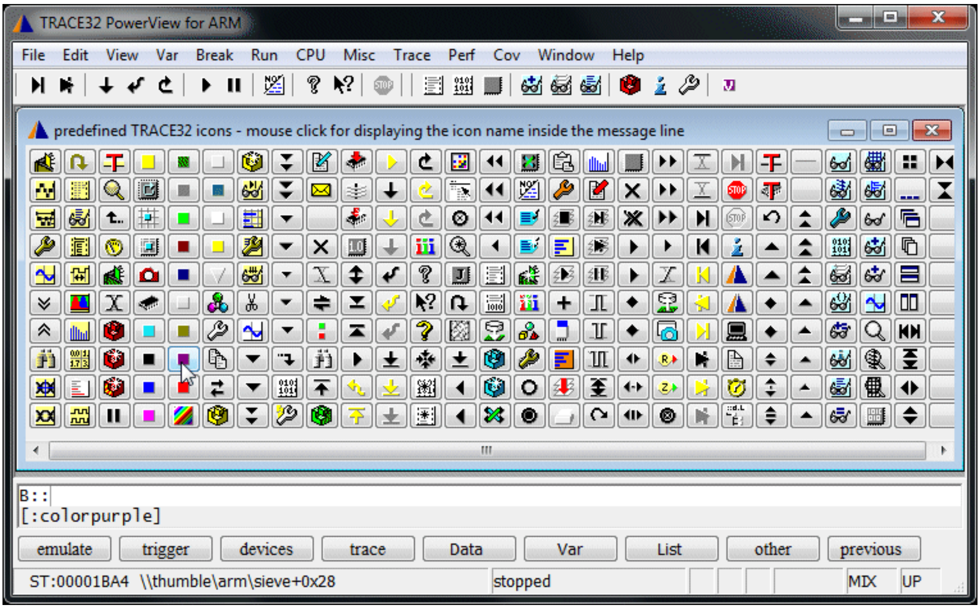
User specific button

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

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

All predefined TRACE32 icons can be inspected as follows:




TRACE32 PowerView for ARM
File
Edit
View
Var
Break
Run
CPU
Misc
Trace
Perf
Cov
Window
Help
predefined TRACE32 icons - mouse click for displaying the icon name inside the message line
emulate
trigger
devices
trace
Data
Var
List
other
previous
MIX
UP

```
B::
[:colorpurple]
```

ST:00001BA4 \\thumble\arm\sieve+0x28 stopped

Or by following TRACE32 command:

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

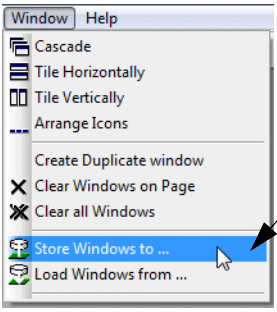
The predefined icons can easily be used to create new icons.

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

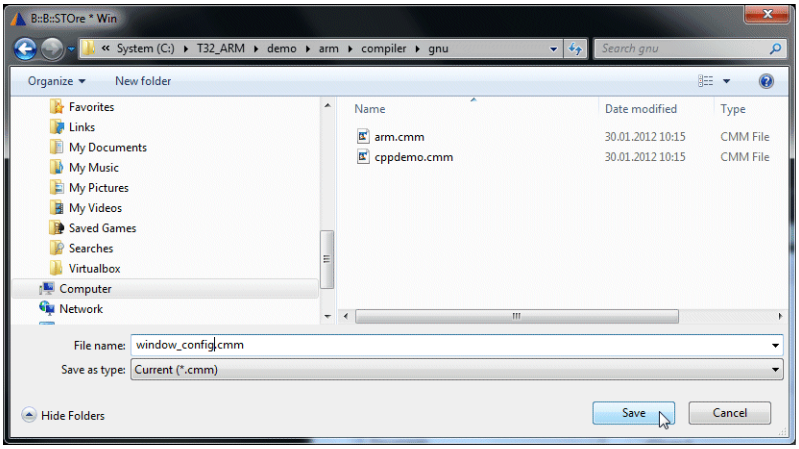
!	For more complex changes to the main tool bar refer to **“Training Menu Programming”** (training_menu.pdf). Videos about the menu programming can be found here: **support.lauterbach.com/kb/articles/trace32-user-interface-customization**

Save Page Layout

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



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



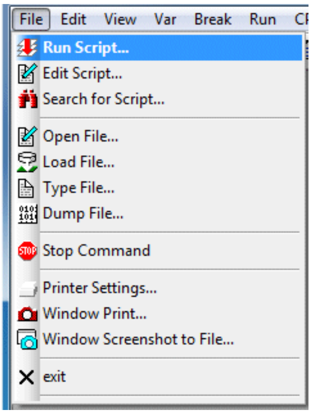
Name	Date modified	Type
arm.cmm	30.01.2012 10:15	CMM File
cppdemo.cmm	30.01.2012 10:15	CMM File

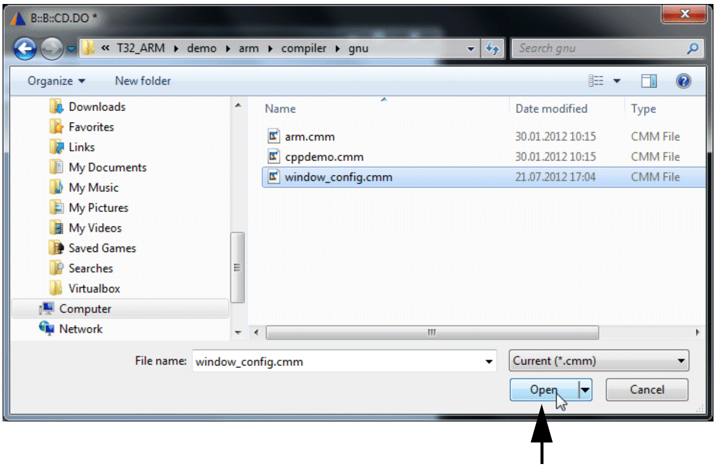
File name: window_config.cmm

Save as type: Current (*.cmm)

Script example:

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

 B::

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

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

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

 WinPAGE.select P000

 ENDDO
```



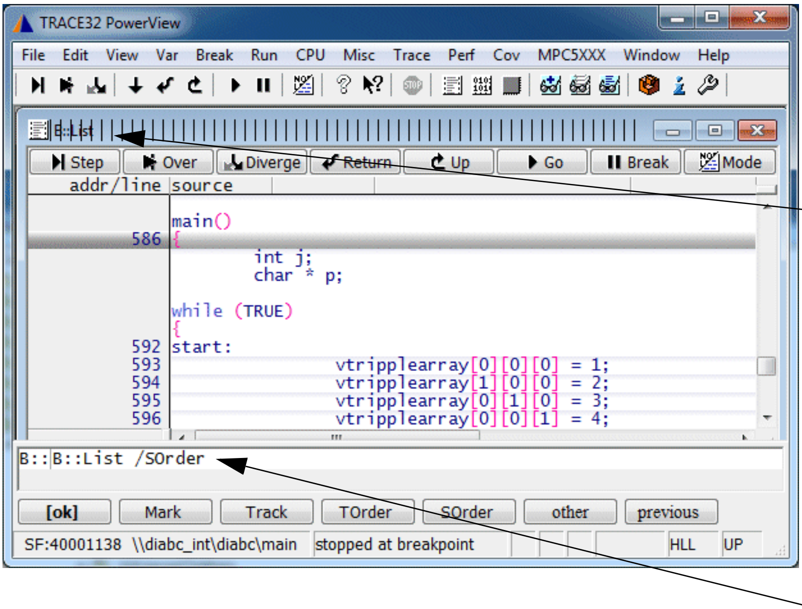


Name	Date modified	Type
arm.cmm	30.01.2012 10:15	CMM File
cppdemo.cmm	30.01.2012 10:15	CMM File
window_config.cmm	21.07.2012 17:04	CMM File

File name: window_config.cmm

Current (*.cmm)

Run the script to reactivate the stored window-configuration



```
     addr/line source

              main()
          586 {
                      int j;
                      char * p;

              while (TRUE)
              {
          592 start:
          593                 vtripplearray[0][0][0] = 1;
          594                 vtripplearray[1][0][0] = 2;
          595                 vtripplearray[0][1][0] = 3;
          596                 vtripplearray[0][0][1] = 4;
```

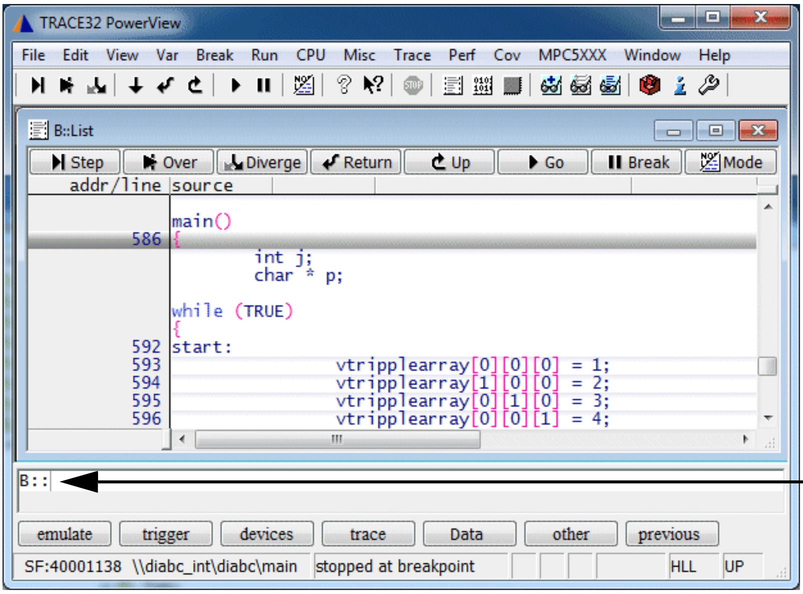
```
B::B::List /SOrder
```

```
SF:40001138  \\diabc_int\diabc\main  stopped at breakpoint  HLL  UP
```

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

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

Command Line


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode
addr/line source
B::
emulate trigger devices trace Data other previous

```
main()
{
        int j;
        char * p;

while (TRUE)
{
start:
                vtripplearray[0][0][0] = 1;
                vtripplearray[1][0][0] = 2;
                vtripplearray[0][1][0] = 3;
                vtripplearray[0][0][1] = 4;
```

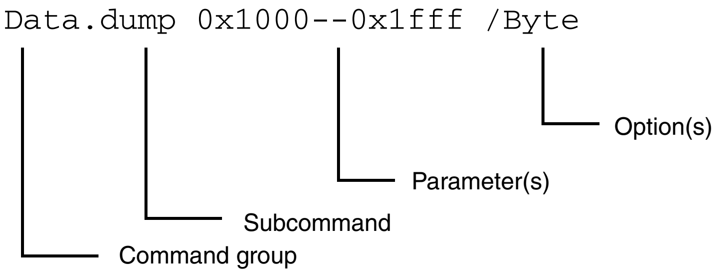
```
SF:40001138  \\diabc_int\diabc\main  stopped at breakpoint  HLL  UP
```

Command line

Command Structure

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

A TRACE32 command has the following structure:



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

Option(s)

Parameter(s)

Subcommand

Command group

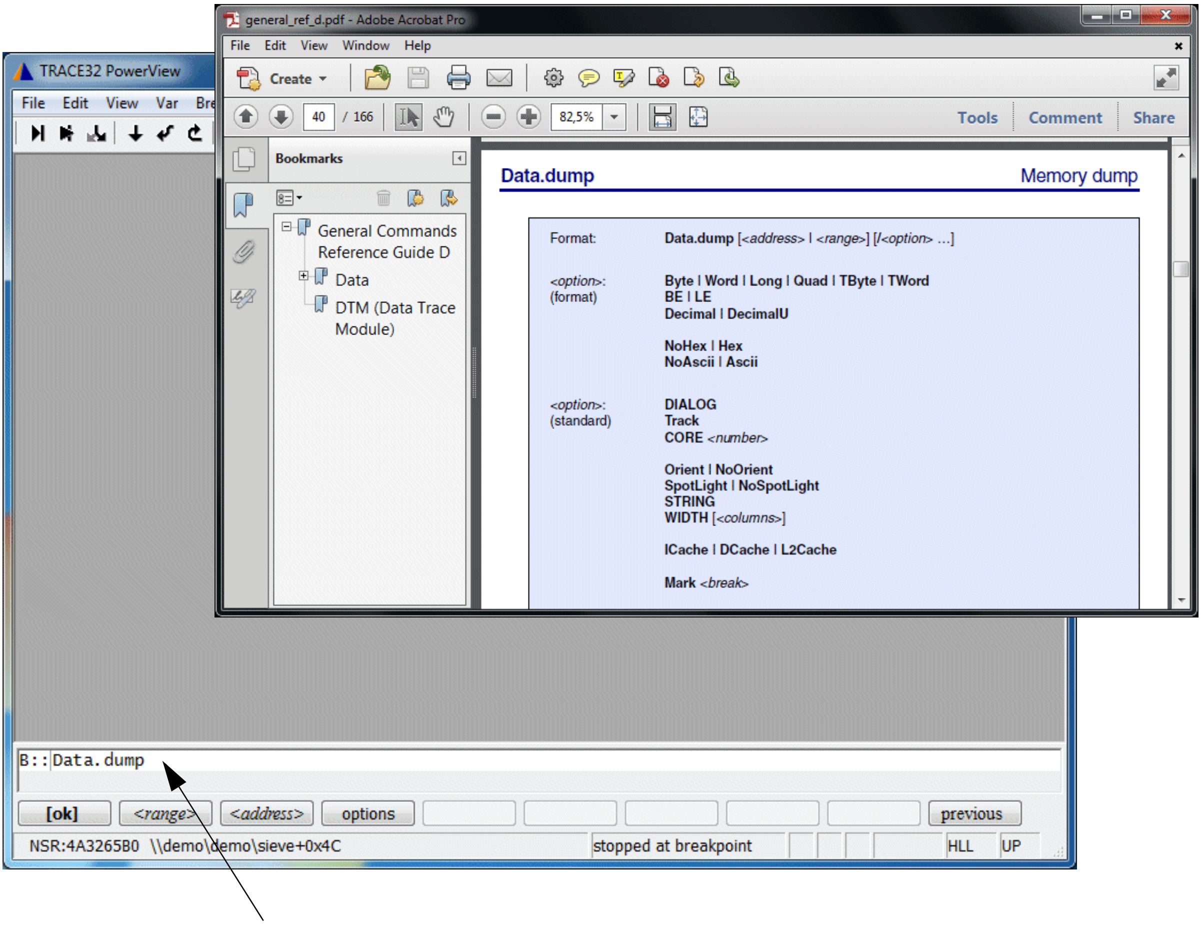
Command Examples

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

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

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

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



Data.dump Memory dump

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

<*option*>: (format)
Byte | **Word** | **Long** | **Quad** | **TByte** | **TWord**
BE | **LE**
Decimal | **DecimalU**

NoHex | **Hex**
NoAscii | **Ascii**

<*option*>: (standard)
DIALOG
Track
CORE <*number*>

Orient | **NoOrient**
SpotLight | **NoSpotLight**
STRING
WIDTH [<*columns*>]

ICache | **DCache** | **L2Cache**

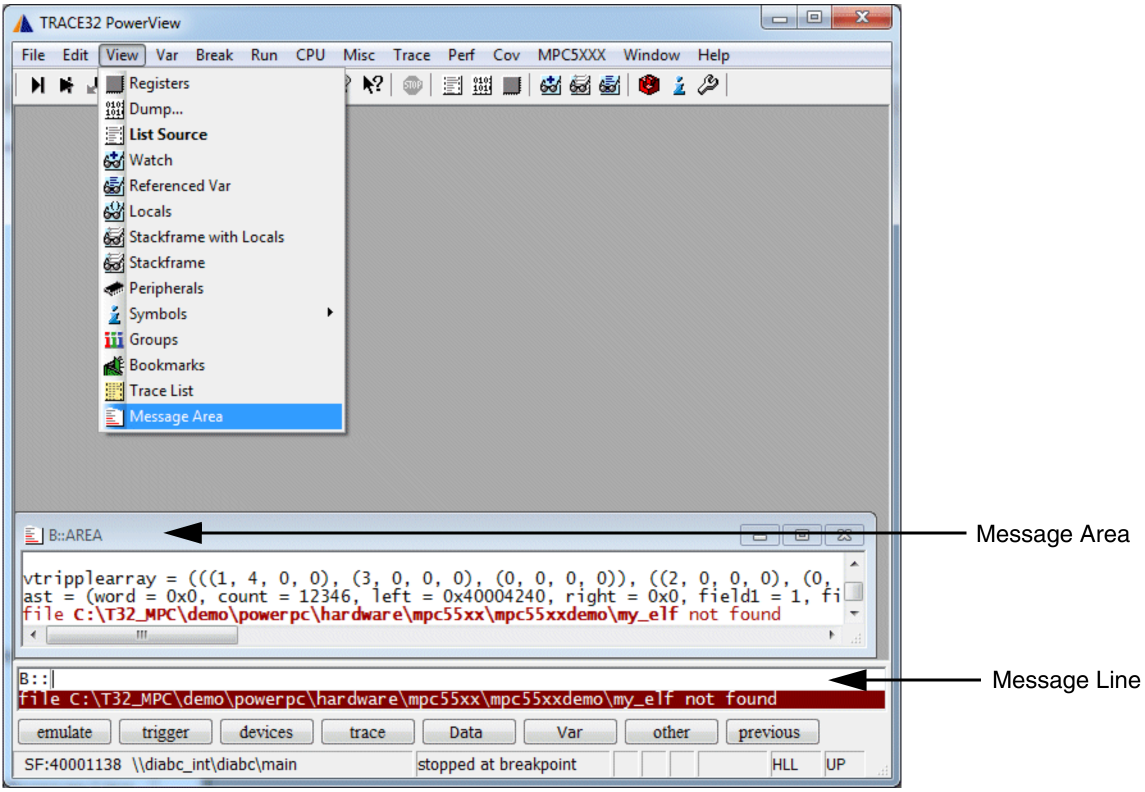
Mark <*break*>

```
B::Data.dump
```

```
NSR:4A3265B0  \\demo\demo\sieve+0x4C        stopped at breakpoint        HLL  UP
```

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

Message Line



```
vtripplearray = (((1, 4, 0, 0), (3, 0, 0, 0), (0, 0, 0, 0)), ((2, 0, 0, 0), (0,
ast = (word = 0x0, count = 12346, left = 0x40004240, right = 0x0, field1 = 1, fi
file C:\T32_MPC\demo\powerpc\hardware\mpc55xx\mpc55xxdemo\my_elf not found
```

```
B::
file C:\T32_MPC\demo\powerpc\hardware\mpc55xx\mpc55xxdemo\my_elf not found
```

Message Area

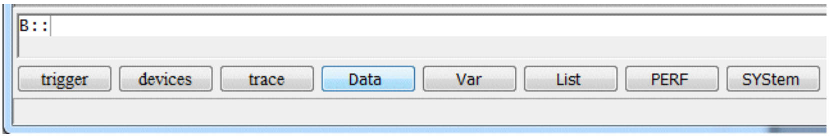
Message Line

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

Softkeys

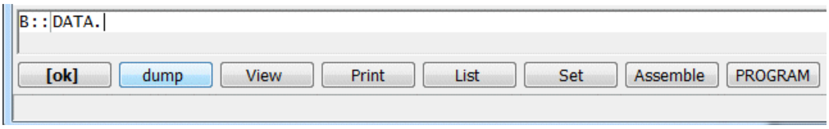
The softkey line allows to enter a specific command step by step. Here an example:

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



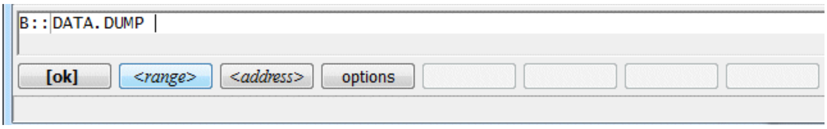
```
B::
```

Select the subcommand, here **dump.**



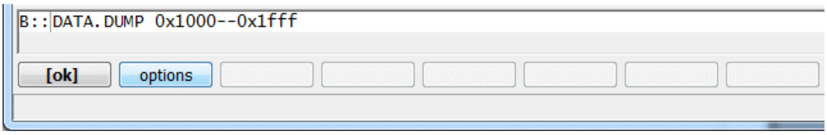
```
B::DATA.
```

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



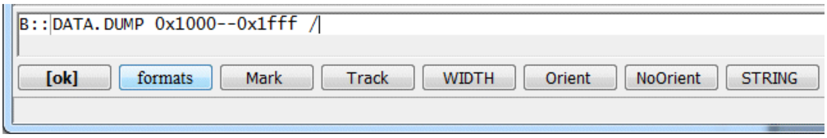
```
B::DATA.DUMP
```

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



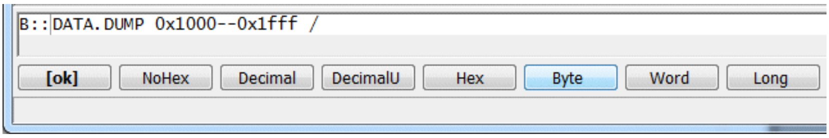
```
B::DATA.DUMP 0x1000--0x1fff
```

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



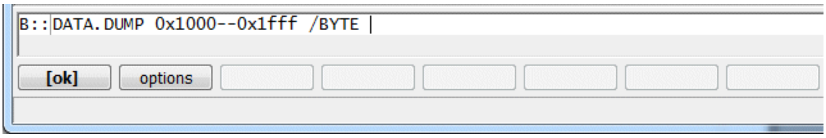
```
B::DATA.DUMP 0x1000--0x1fff /
```

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

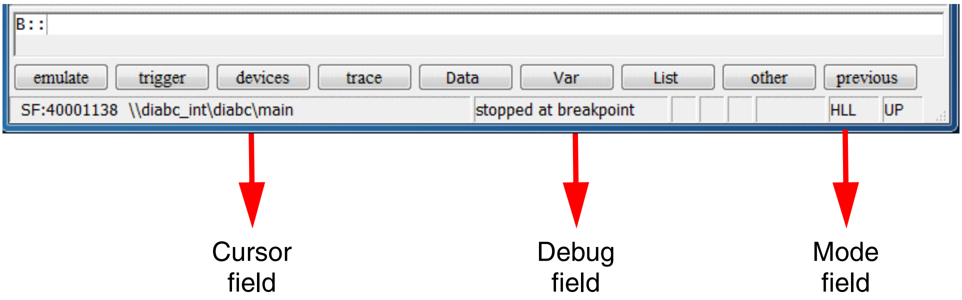


```
B::DATA.DUMP 0x1000--0x1fff /
```

The command is complete now.



```
B::DATA.DUMP 0x1000--0x1fff /BYTE
```



```
SF:40001138  \\diabc_int\diabc\main        stopped at breakpoint        HLL   UP
```

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

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

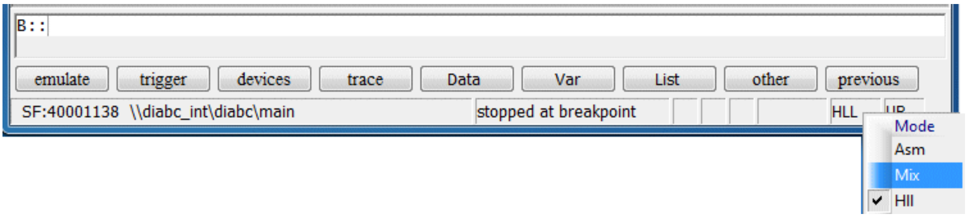
The **Debug** field of the state line provides:

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

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

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

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



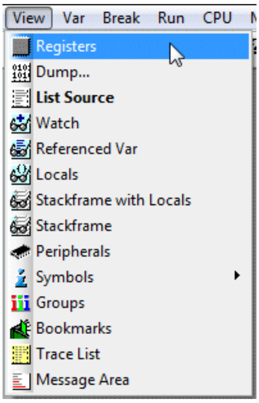
```
SF:40001138  \\diabc_int\diabc\main        stopped at breakpoint        HLL   UP
```

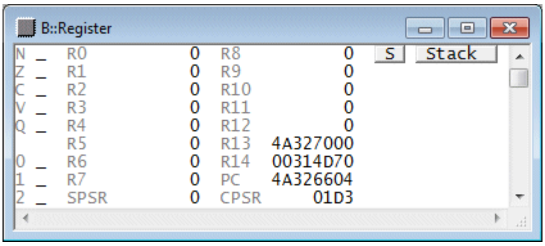
The debug mode can be changed by using the **Mode** pull-down.

Registers

Core Registers

Display the Core Registers





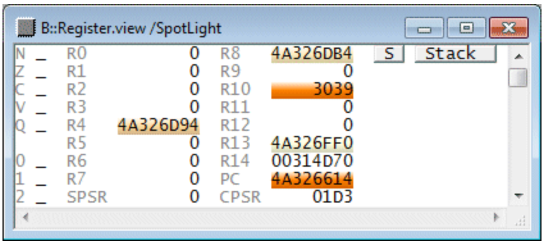
```
N _  R0      0  R8          0   S  Stack
Z _  R1      0  R9          0
C _  R2      0  R10         0
V _  R3      0  R11         0
Q _  R4      0  R12         0
     R5      0  R13  4A327000
0 _  R6      0  R14  00314D70
1 _  R7      0  PC   4A326604
2 _  SPSR    0  CPSR     01D3
```

```
Register.view
```

Colored Display of Changed Registers

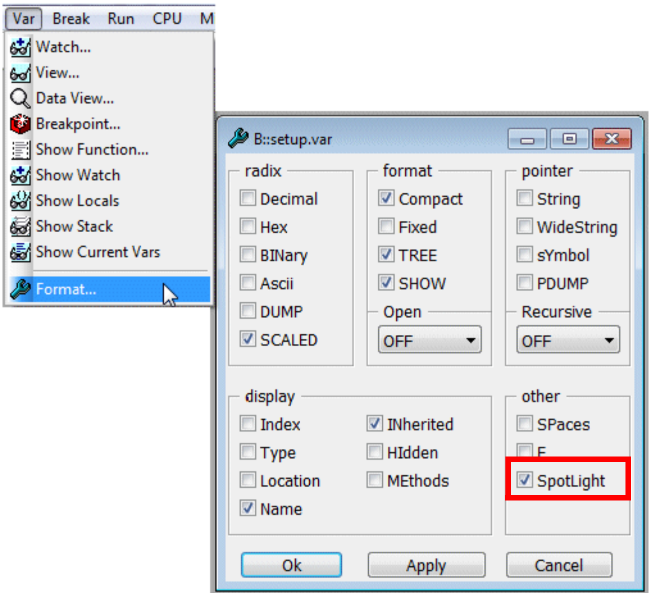
The option /SpotLight advises TRACE32 PowerView to mark changes.

```
  Register.view /SpotLight                 ; The registers changed by the last
                                           ; step are marked in dark red.

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

                                           ; This works up to a level of 4.
```


B::Register.view /SpotLight

```
N _  R0         0  R8   4A326DB4   S   Stack
Z _  R1         0  R9          0
C _  R2         0  R10      3039
V _  R3         0  R11         0
Q _  R4  4A326D94  R12         0
     R5         0  R13  4A326FF0
0 _  R6         0  R14  00314D70
1 _  R7         0  PC   4A326614
2 _  SPSR       0  CPSR     01D3
```

Establish /SpotLight as default setting



B::setup.var

radix: □ Decimal □ Hex □ BINary □ Ascii □ DUMP ☑ SCALED

format: ☑ Compact □ Fixed ☑ TREE ☑ SHOW

Open: OFF

pointer: □ String □ WideString □ sYmbol □ PDUMP

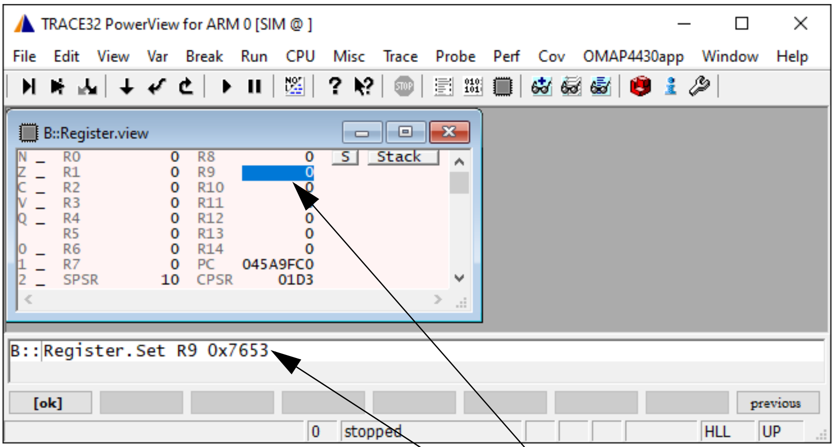
Recursive: OFF

display: □ Index □ Type □ Location ☑ Name ☑ INherited □ HIdden □ MEthods

other: □ SPaces □ E ☑ SpotLight

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


TRACE32 PowerView for ARM 0 [SIM @]
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov OMAP4430app Window Help
B::Register.view
[ok]
previous
0
stopped
HLL
UP

```
N _  R0         0  R8         0   S  Stack
Z _  R1         0  R9         0
C _  R2         0  R10        0
V _  R3         0  R11        0
Q _  R4         0  R12        0
     R5         0  R13        0
0 _  R6         0  R14        0
1 _  R7         0  PC   045A9FC0
2 _  SPSR      10  CPSR     01D3
```

```
B::Register.Set R9 0x7653
```

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

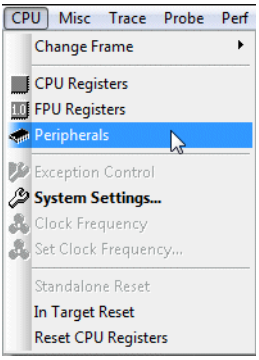
Register.Set *<register> <value>*	Modify register

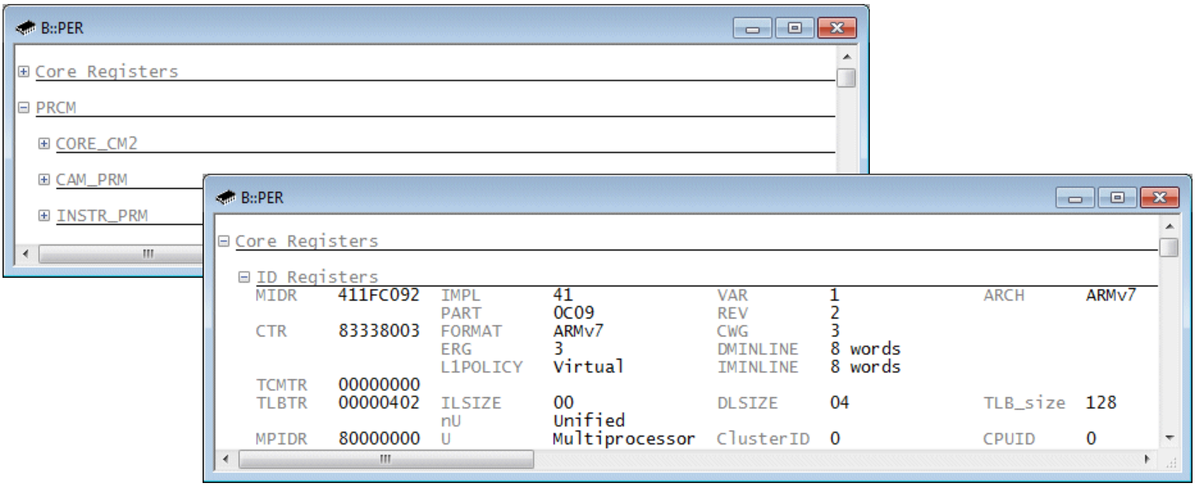
Display the Special Function Registers

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

Tree Display

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

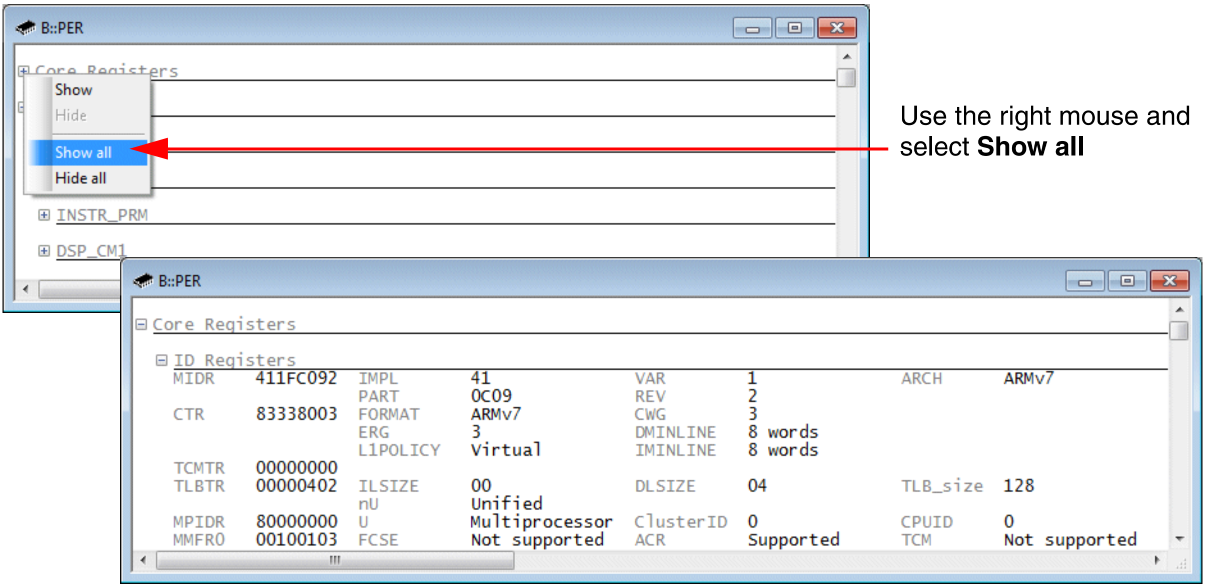



```
B::PER
⊞ Core Registers
⊟ PRCM
  ⊞ CORE_CM2
  ⊞ CAM_PRM
  ⊞ INSTR_PRM
```

```
B::PER
⊟ Core Registers
  ⊟ ID Registers
  MIDR     411FC092  IMPL      41              VAR       1           ARCH      ARMv7
                     PART      0C09            REV       2
  CTR      83338003  FORMAT    ARMv7           CWG       3
                     ERG       3               DMINLINE  8 words
                     L1POLICY  Virtual         IMINLINE  8 words
  TCMTR    00000000
  TLBTR    00000402  ILSIZE    00              DLSIZE    04          TLB_size  128
                     nU        Unified
  MPIDR    80000000  U         Multiprocessor  ClusterID 0           CPUID     0
```

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

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



```
⊟ Core Registers
  ⊟ ID Registers
    MIDR    411FC092  IMPL      41              VAR        1          ARCH      ARMv7
                      PART      0C09            REV        2
    CTR     83338003  FORMAT    ARMv7           CWG        3
                      ERG       3               DMINLINE   8 words
                      L1POLICY  Virtual         IMINLINE   8 words
    TCMTR   00000000
    TLBTR   00000402  ILSIZE    00              DLSIZE     04         TLB_size  128
                      nU        Unified
    MPIDR   80000000  U         Multiprocessor  ClusterID  0          CPUID     0
    MMFR0   00100103  FCSE      Not supported   ACR        Supported  TCM       Not supported
```

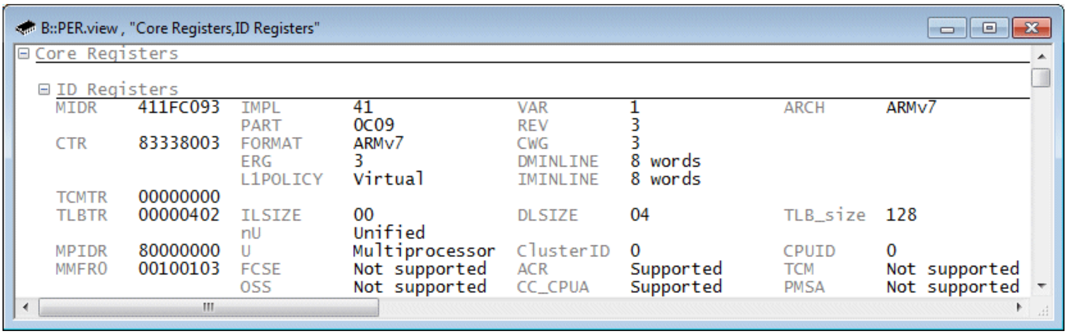
Use the right mouse and select **Show all**

Commands:

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

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

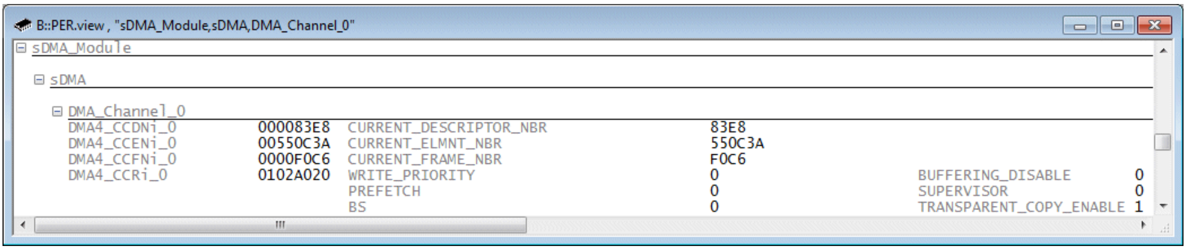
```
; Display the functional unit "ID Registers" within "Core Registers"
; in expanded mode
PER.view , "Core Registers,ID Registers"
```



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

```
⊟ Core Registers

  ⊟ ID Registers
   MIDR     411FC093  IMPL      41              VAR        1          ARCH      ARMv7
                      PART      0C09            REV        3
   CTR      83338003  FORMAT    ARMv7           CWG        3
                      ERG       3               DMINLINE   8 words
                      L1POLICY  Virtual         IMINLINE   8 words
   TCMTR    00000000
   TLBTR    00000402  ILSIZE    00              DLSIZE     04         TLB_size  128
                      nU        Unified
   MPIDR    80000000  U         Multiprocessor  ClusterID  0          CPUID     0
   MMFR0    00100103  FCSE      Not supported   ACR        Supported  TCM       Not supported
                      OSS       Not supported   CC_CPUA    Supported  PMSA      Not supported
```

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

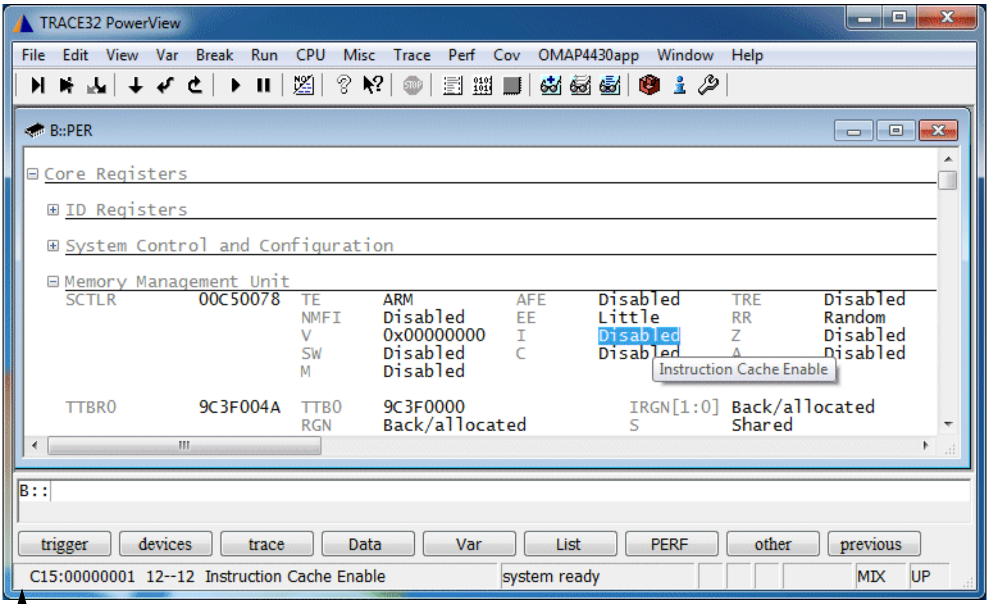


B::PER.view , "sDMA_Module,sDMA,DMA_Channel_0"

```
⊟ sDMA_Module

  ⊟ sDMA

    ⊟ DMA_Channel_0
     DMA4_CCDNi_0    000083E8  CURRENT_DESCRIPTOR_NBR   83E8
     DMA4_CCENi_0    00550C3A  CURRENT_ELMNT_NBR        550C3A
     DMA4_CCFNi_0    0000F0C6  CURRENT_FRAME_NBR        F0C6
     DMA4_CCRi_0     0102A020  WRITE_PRIORITY           0       BUFFERING_DISABLE        0
                               PREFETCH                 0       SUPERVISOR               0
                               BS                       0       TRANSPARENT_COPY_ENABLE  1
```

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

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

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

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


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
B::PER
trigger
devices
trace
Data
Var
List
PERF
other
previous

```
⊟ Core Registers
  ⊞ ID Registers
  ⊞ System Control and Configuration
  ⊟ Memory Management Unit
  SCTLR         00C50078  TE    ARM             AFE  Disabled     TRE        Disabled
                          NMFI  Disabled        EE   Little       RR         Random
                          V     0x00000000      I    Disabled     Z          Disabled
                          SW    Disabled        C    Disabled     A          Disabled
                          M     Disabled                Instruction Cache Enable

  TTBR0         9C3F004A  TTB0  9C3F0000                     IRGN[1:0]  Back/allocated
                          RGN   Back/allocated               S          Shared
```

```
B::
```

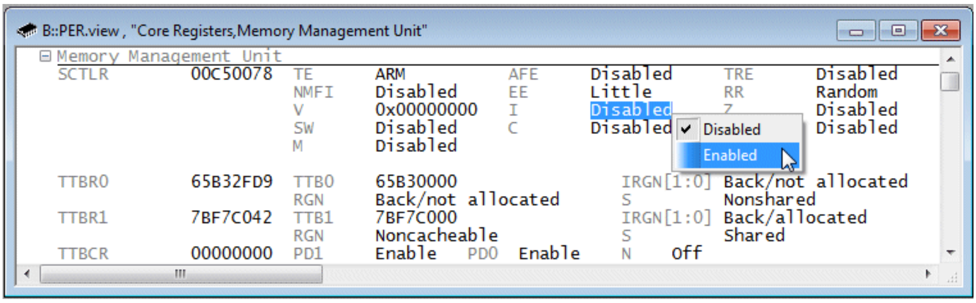
```
C15:00000001  12--12  Instruction Cache Enable      system ready      MIX   UP
```

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

Modify a Special Function Register

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

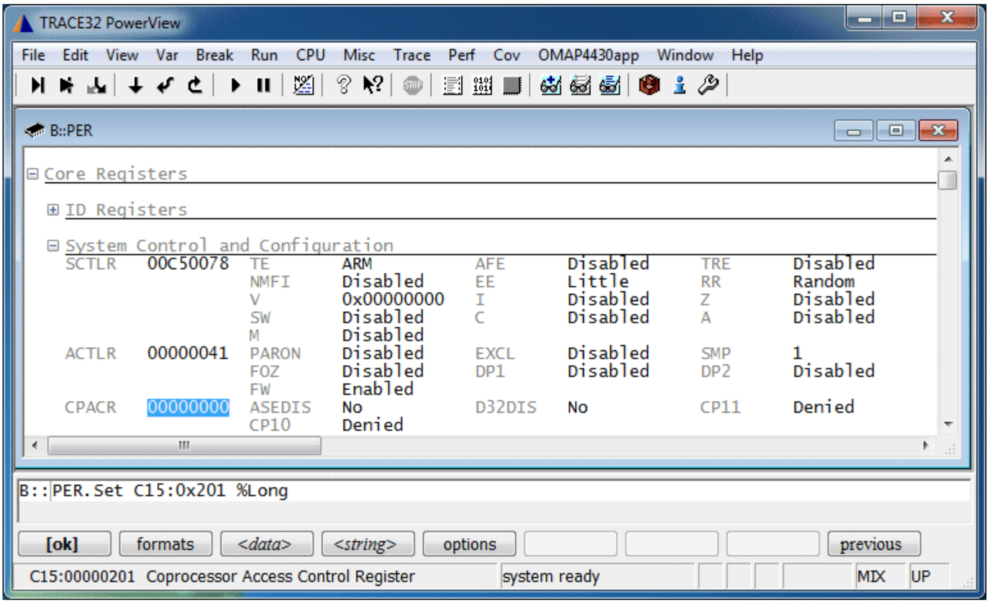
- By pressing the right mouse button and selecting one of the predefined values from the pull-down menu.



B::PER.view , "Core Registers,Memory Management Unit"

```
⊟ Memory Management Unit
SCTLR      00C50078  TE     ARM              AFE   Disabled   TRE   Disabled
                     NMFI   Disabled         EE    Little     RR    Random
                     V      0x00000000       I     Disabled   Z     Disabled
                     SW     Disabled         C     Disabled   A     Disabled
                     M      Disabled

TTBR0      65B32FD9  TTB0   65B30000                    IRGN[1:0] Back/not allocated
                     RGN    Back/not allocated          S         Nonshared
TTBR1      7BF7C042  TTB1   7BF7C000                    IRGN[1:0] Back/allocated
                     RGN    Noncacheable                S         Shared
TTBCR      00000000  PD1    Enable   PD0  Enable        N     off
```

- 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.



TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
B::PER

```
⊟ Core Registers
  ⊞ ID Registers
  ⊟ System Control and Configuration
  SCTLR    00C50078  TE      ARM          AFE     Disabled   TRE   Disabled
                     NMFI    Disabled     EE      Little     RR    Random
                     V       0x00000000   I       Disabled   Z     Disabled
                     SW      Disabled     C       Disabled   A     Disabled
                     M       Disabled
  ACTLR    00000041  PARON   Disabled     EXCL    Disabled   SMP   1
                     FOZ     Disabled     DP1     Disabled   DP2   Disabled
                     FW      Enabled
  CPACR    00000000  ASEDIS  No           D32DIS  No         CP11  Denied
                     CP10    Denied
```

```
B::PER.Set C15:0x201 %Long
```

C15:00000201 Coprocessor Access Control Register | system ready | MIX | UP

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

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

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

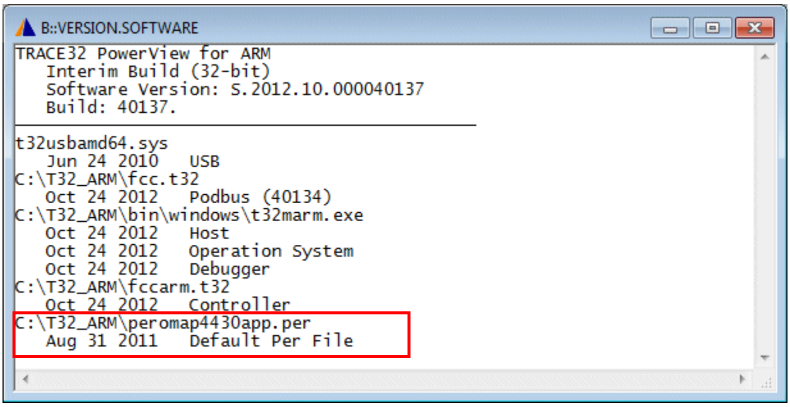
The PER Definition File

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

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

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

VERSION.SOFTWARE



```
TRACE32 PowerView for ARM
   Interim Build (32-bit)
   Software Version: S.2012.10.000040137
   Build: 40137.
_____________________________________________
t32usbamd64.sys
   Jun 24 2010    USB
C:\T32_ARM\fcc.t32
   Oct 24 2012    Podbus (40134)
C:\T32_ARM\bin\windows\t32marm.exe
   Oct 24 2012    Host
   Oct 24 2012    Operation System
   Oct 24 2012    Debugger
C:\T32_ARM\fccarm.t32
   Oct 24 2012    Controller
C:\T32_ARM\peromap4430app.per
   Aug 31 2011    Default Per File
```

PER.view *<filename>*	Display the configuration registers/on-chip peripherals specified by *<filename>*

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

Memory Display and Modification

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

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

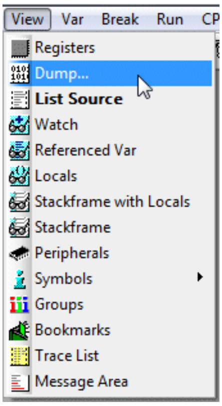
A so-called **access class** is always displayed together with a memory address. The following access classes are available for all processor architectures:

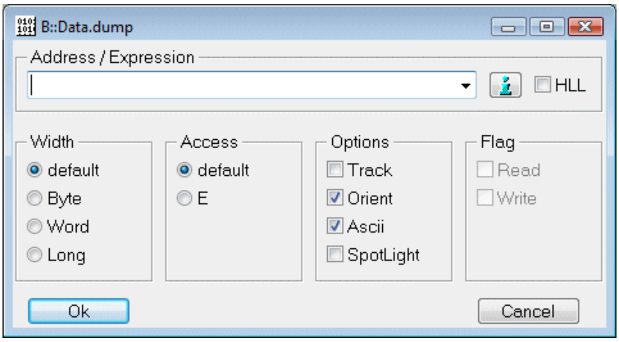
P:1000	**Program** address 0x1000
D:6814	**Data** address 0x6814

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

The Data.dump Window

Display the Memory Contents





B::Data.dump

Address / Expression

□ HLL

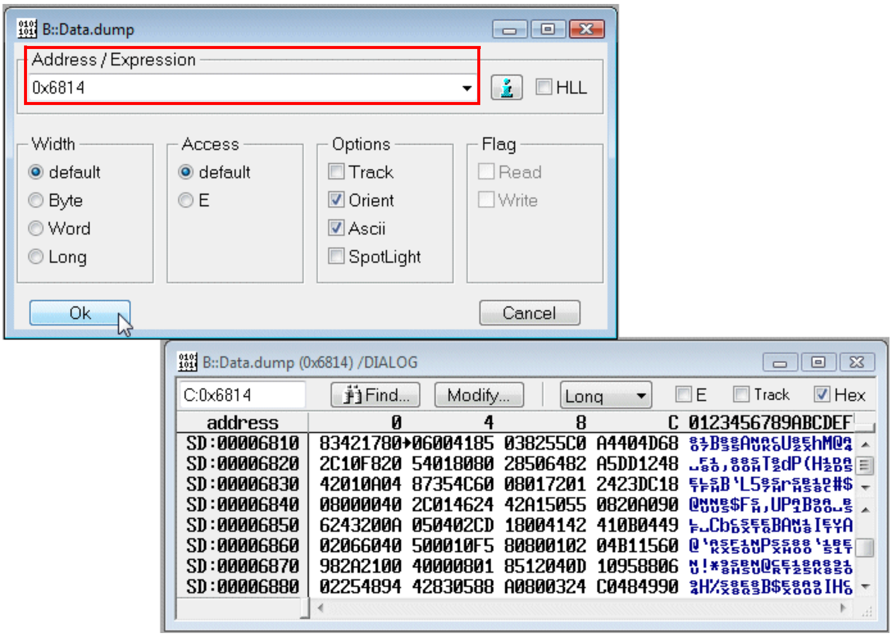
Width: ● default ○ Byte ○ Word ○ Long

Access: ● default ○ E

Options: □ Track ☑ Orient ☑ Ascii □ SpotLight

Flag: □ Read □ Write

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



B::Data.dump

Address / Expression: 0x6814 □ HLL

Width: ● default ○ Byte ○ Word ○ Long

Access: ● default ○ E

Options: □ Track ☑ Orient ☑ Ascii □ SpotLight

Flag: □ Read □ Write

B::Data.dump (0x6814) /DIALOG

C:0x6814 | Long | □ E □ Track ☑ Hex

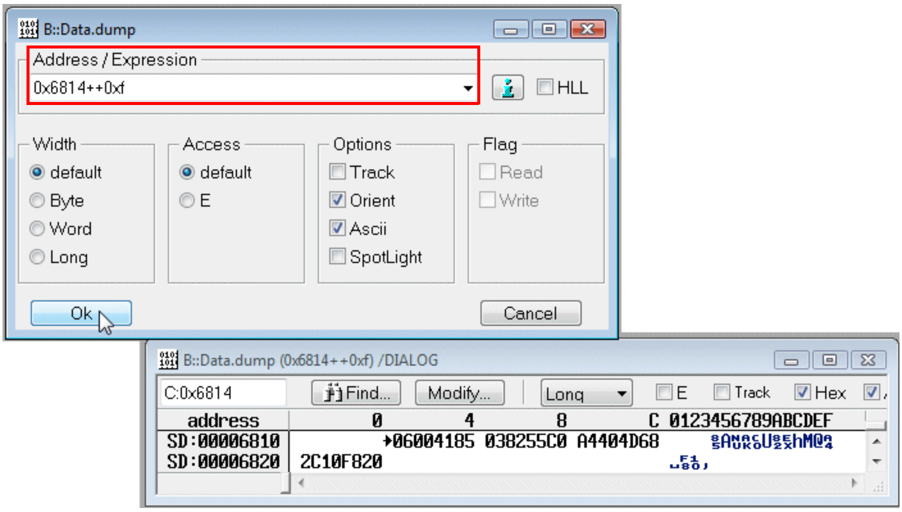
```
   address           0        4        8        C 0123456789ABCDEF
SD:00006810  83421780▸06004185 038255C0 A4404D68
SD:00006820  2C10F820 54018080 28506482 A5DD1248
SD:00006830  42010A04 87354C60 08017201 2423DC18
SD:00006840  08000040 2C014624 42A15055 0820A090
SD:00006850  6243200A 050402CD 18004142 410B0449
SD:00006860  02066040 500010F5 80800102 04B11560
SD:00006870  982A2100 40000801 8512040D 10958806
SD:00006880  02254894 42830588 A0800324 C0484990
```

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

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

Conventions for address ranges:

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



B::Data.dump

Address / Expression: 0x6814++0xf

□ HLL

Width: ● default ○ Byte ○ Word ○ Long

Access: ● default ○ E

Options: □ Track ☑ Orient ☑ Ascii □ SpotLight

Flag: □ Read □ Write

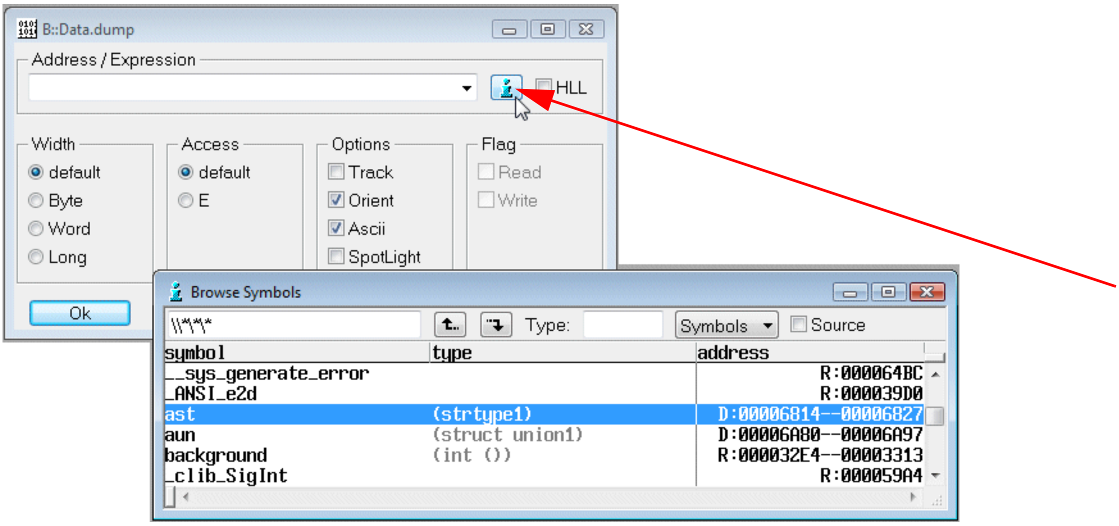
B::Data.dump (0x6814++0xf) /DIALOG

C:0x6814

Long □ E □ Track ☑ Hex ☑

```
  address          0        4        8        C 0123456789ABCDEF
SD:00006810          ▸06004185 038255C0 A4404D68
SD:00006820 2C10F820
```

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



B::Data.dump

Address / Expression: (empty) □ HLL

Width: ● default ○ Byte ○ Word ○ Long

Access: ● default ○ E

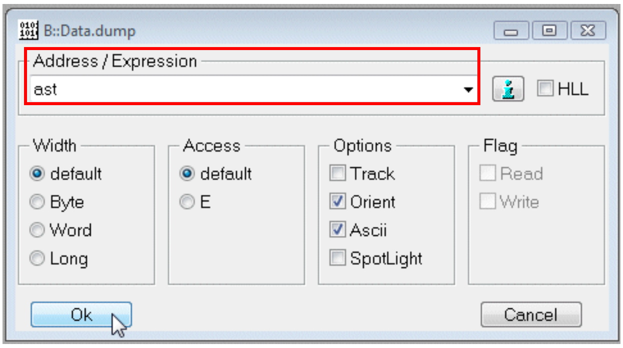
Options: □ Track ☑ Orient ☑ Ascii □ SpotLight

Flag: □ Read □ Write

Browse Symbols

`\\*\*\*` Type: Symbols □ Source

```
symbol                    type                 address
__sys_generate_error                                       R:000064BC
_ANSI_e2d                                                  R:000039D0
ast                       (strtype1)           D:00006814--00006827
aun                       (struct union1)      D:00006A80--00006A97
background                (int ())             R:000032E4--00003313
_clib_SigInt                                               R:000059A4
```



B::Data.dump

Address / Expression: ast □ HLL

Width: ● default ○ Byte ○ Word ○ Long

Access: ● default ○ E

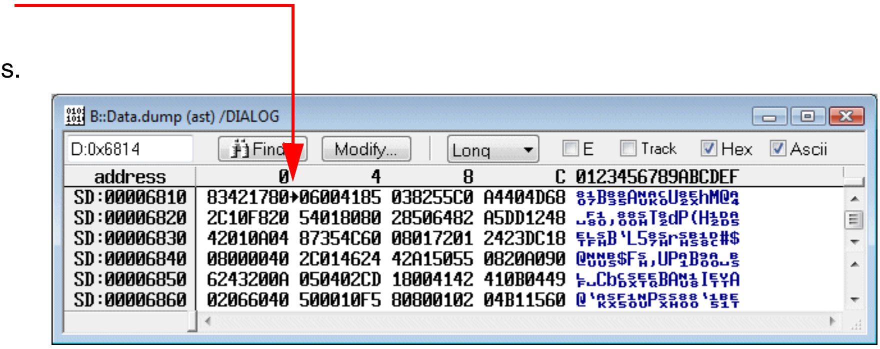
Options: □ Track ☑ Orient ☑ Ascii □ SpotLight

Flag: □ Read □ Write

Ok Cancel

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

A small arrow indicates the specified dump address.



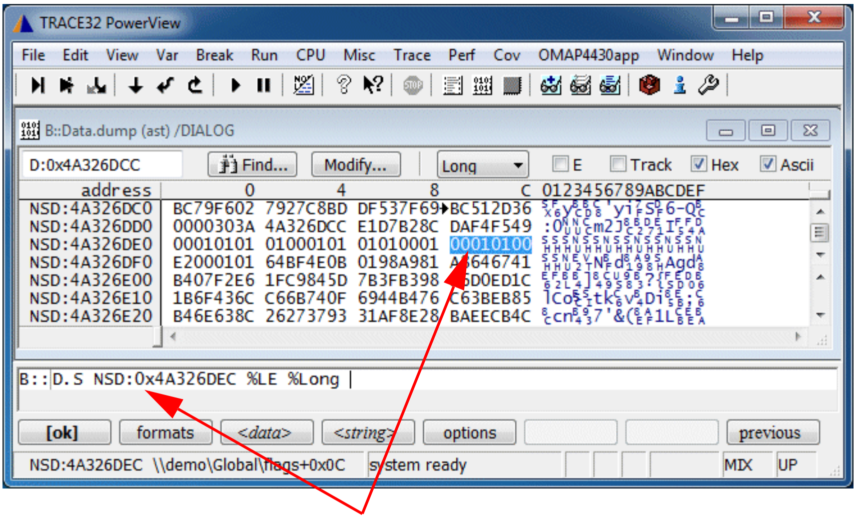
B::Data.dump (ast) /DIALOG

D:0x6814 Long □ E □ Track ☑ Hex ☑ Ascii

```
   address           0        4        8        C 0123456789ABCDEF
SD:00006810  83421780▸06004185 038255C0 A4404D68
SD:00006820  2C10F820 54018080 28506482 A5DD1248
SD:00006830  42010A04 87354C60 08017201 2423DC18
SD:00006840  08000040 2C014624 42A15055 0820A090
SD:00006850  6243200A 050402CD 18004142 410B0449
SD:00006860  02066040 500010F5 80800102 04B11560
```

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

```
Data.dump 0x6814                      ; Display a hex dump starting at
                                      ; address 0x6814

Data.dump 0x6810--0x682f              ; Display a hex dump of the
                                      ; specified address range

Data.dump 0x6810..0x682f              ; Display a hex dump of the
                                      ; specified address range

Data.dump 0x6810++0x1f                ; Display a hex dump of the
                                      ; specified address range

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

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



```
      address       0        4        8        C  0123456789ABCDEF
NSD:4A326DC0  BC79F602 7927C8BD DF537F69 BC512D36
NSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
NSD:4A326DE0  00010101 01000101 01010001 00010100
NSD:4A326DF0  E2000101 64BF4E0B 0198A981 A[illegible]646741
NSD:4A326E00  B407F2E6 1FC9845D 7B3FB398 [illegible]5D0ED1C
NSD:4A326E10  1B6F436C C66B740F 6944B476 C63BEB85
NSD:4A326E20  B46E638C 26273793 31AF8E28 BAEECB4C
```

```
B::D.S NSD:0x4A326DEC %LE %Long
```

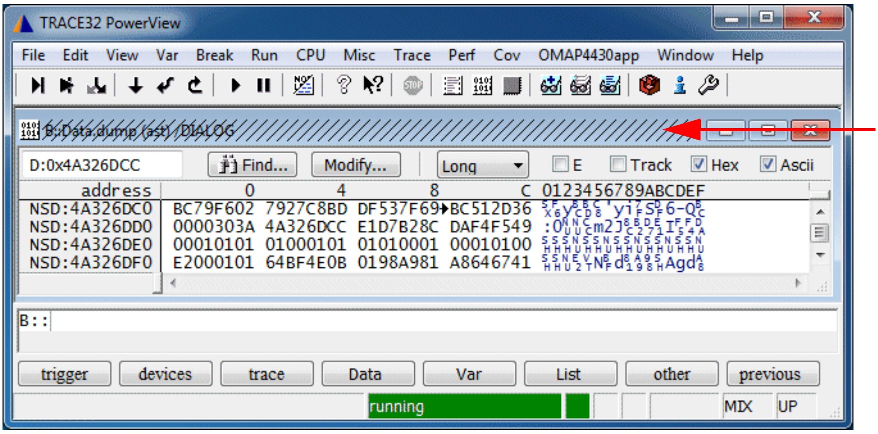
```
NSD:4A326DEC  \\demo\Global\flags+0x0C    system ready    MIX  UP
```

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

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

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

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

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

                                        ; Use Little Endian mode
```

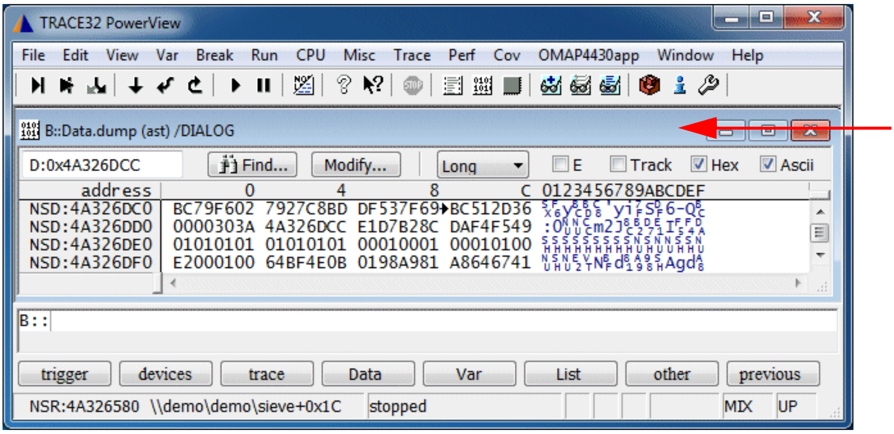
TRACE32 PowerView updates the displayed memory contents by default only if the cores is stopped.



```
     address    0        4        8        C     0123456789ABCDEF
NSD:4A326DC0  BC79F602 7927C8BD DF537F69▸BC512D36
NSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
NSD:4A326DE0  00010101 01000101 01010001 00010100
NSD:4A326DF0  E2000101 64BF4E0B 0198A981 A8646741
```

running

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



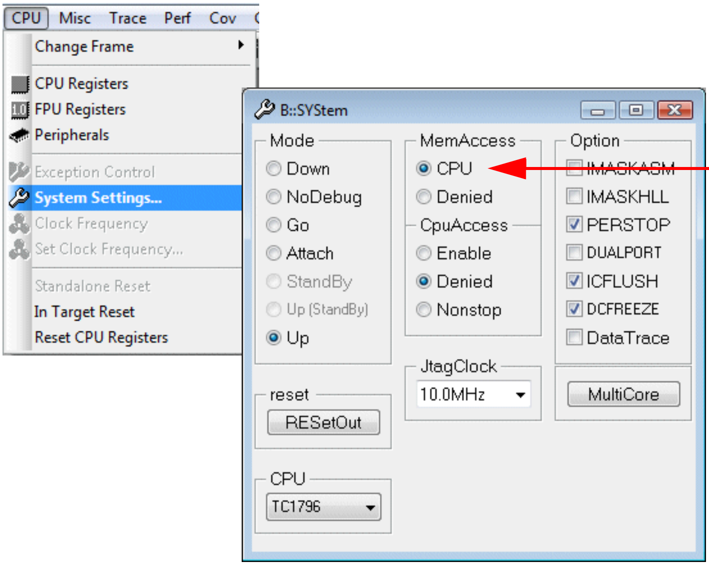
```
     address    0        4        8        C     0123456789ABCDEF
NSD:4A326DC0  BC79F602 7927C8BD DF537F69▸BC512D36
NSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
NSD:4A326DE0  01010101 01010101 00010001 00010100
NSD:4A326DF0  E2000100 64BF4E0B 0198A981 A8646741
```

NSR:4A326580 \\demo\demo\sieve+0x1C stopped

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

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

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



Mode
○ Down
○ NoDebug
○ Go
○ Attach
○ StandBy
○ Up (StandBy)
● Up

MemAccess
● CPU
○ Denied

CpuAccess
○ Enable
● Denied
○ Nonstop

Option
□ IMASKASM
□ IMASKHLL
☑ PERSTOP
□ DUALPORT
☑ ICFLUSH
☑ DCFREEZE
□ DataTrace

reset

JtagClock
10.0MHz

CPU
TC1796

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

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

Restrictions

The following description is only a rough overview on the restrictions. Details about your core can be found in the **Processor Architecture Manual**.

Cache

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

- **Program execution is stopped**

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

- **Program execution is running**

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

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

MMU

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

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

Tightly-coupled Memory

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

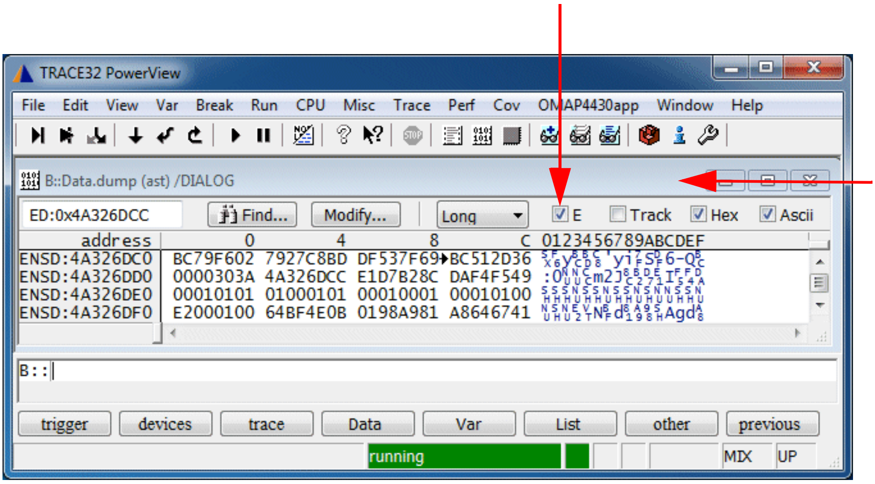
Usage

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

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

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

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


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
B::Data.dump (ast) /DIALOG
Find...
Modify...
trigger
devices
trace
Data
Var
List
other
previous

ED:0x4A326DCC | Long | ☑ E | □ Track | ☑ Hex | ☑ Ascii

```
        address    0        4        8        C  0123456789ABCDEF
ENSD:4A326DC0  BC79F602 7927C8BD DF537F69▸BC512D36
ENSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
ENSD:4A326DE0  00010101 01000101 00010001 00010100
ENSD:4A326DF0  E2000100 64BF4E0B 0198A981 A8646741
```

```
B::
```

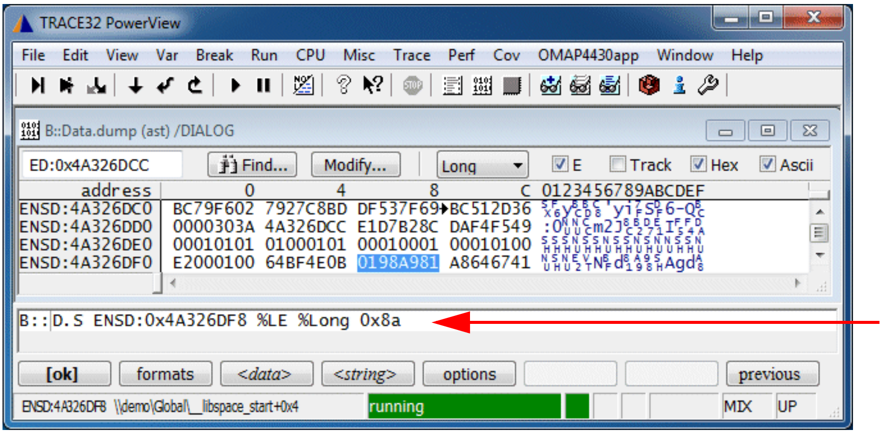
running | MIX | UP

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

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

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

Write accesses to the memory work correspondingly:


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
B::Data.dump (ast) /DIALOG
Find...
Modify...
[ok]
formats
<data>
<string>
options
previous

ED:0x4A326DCC | Long | ☑ E | □ Track | ☑ Hex | ☑ Ascii

```
        address    0        4        8        C  0123456789ABCDEF
ENSD:4A326DC0  BC79F602 7927C8BD DF537F69▸BC512D36
ENSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
ENSD:4A326DE0  00010101 01000101 00010001 00010100
ENSD:4A326DF0  E2000100 64BF4E0B 0198A981 A8646741
```

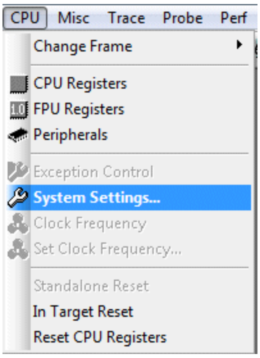
```
B::D.S ENSD:0x4A326DF8 %LE %Long 0x8a
```

ENSD:4A326DF8 \\demo\Global\__libspace_start+0x4 | running | MIX | UP

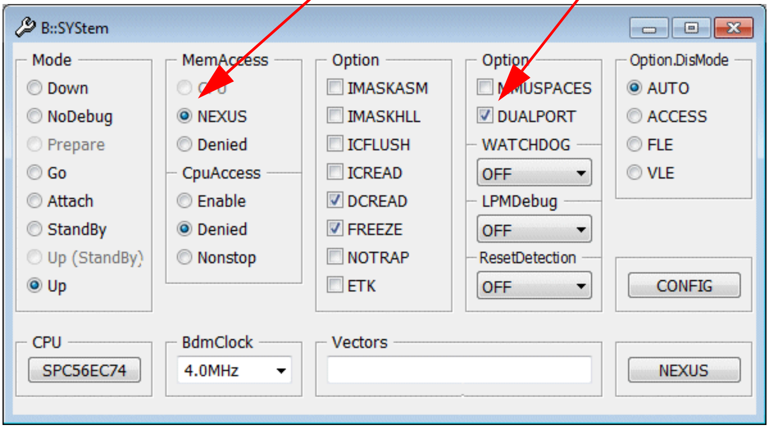
`Data.Set` via run-time memory access (attribute E)

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

;…

Go                                    ; Start program execution

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

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

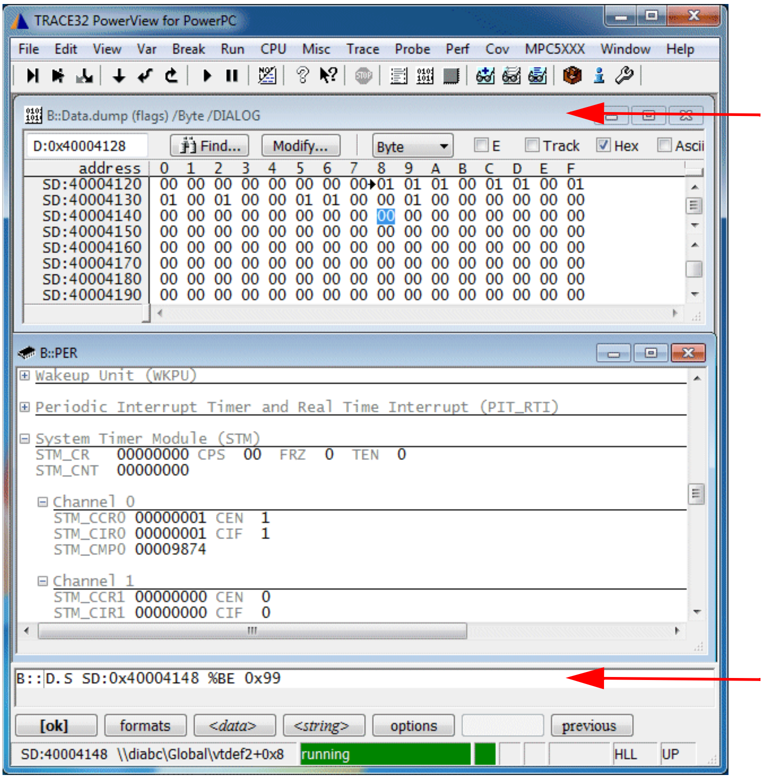
Configure the run-time memory access for all windows that display memory (not available for all cores):



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


B::SYStem

- Mode: ○ Down, ○ NoDebug, ○ Prepare, ○ Go, ○ Attach, ○ StandBy, ○ Up (StandBy), ● Up
- MemAccess: ○ CPU, ● NEXUS, ○ Denied
- CpuAccess: ○ Enable, ● Denied, ○ Nonstop
- Option: □ IMASKASM, □ IMASKHLL, □ ICFLUSH, □ ICREAD, ☑ DCREAD, ☑ FREEZE, □ NOTRAP, □ ETK
- Option: □ MMUSPACES, ☑ DUALPORT
- WATCHDOG: OFF
- LPMDebug: OFF
- ResetDetection: OFF
- Option.DisMode: ● AUTO, ○ ACCESS, ○ FLE, ○ VLE
- CPU: SPC56EC74
- BdmClock: 4.0MHz
- Vectors:



B::Data.dump (flags) /Byte /DIALOG

D:0x40004128

```
    address  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
 SD:40004120 00 00 00 00 00 00 00 00 01 01 01 00 01 01 00 01
 SD:40004130 01 00 01 00 00 01 01 00 00 01 00 00 00 00 00 00
 SD:40004140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 SD:40004150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 SD:40004160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 SD:40004170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 SD:40004180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 SD:40004190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

B::PER

```
⊞ Wakeup Unit (WKPU)

⊞ Periodic Interrupt Timer and Real Time Interrupt (PIT_RTI)

⊟ System Timer Module (STM)
  STM_CR    00000000 CPS   00   FRZ   0   TEN   0
  STM_CNT   00000000

  ⊟ Channel 0
    STM_CCR0  00000001 CEN   1
    STM_CIR0  00000001 CIF   1
    STM_CMP0  00009874

  ⊟ Channel 1
    STM_CCR1  00000000 CEN   0
    STM_CIR1  00000000 CIF   0
```

```
B::D.S SD:0x40004148 %BE 0x99
```

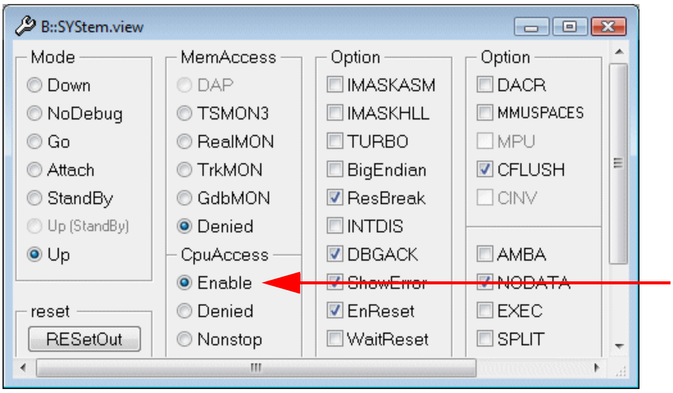
SD:40004148 \\diabc\Global\vtdef2+0x8 running HLL UP

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

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

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

SYStem.Option.DUALPORT ON                    ; Activate the run-time memory
                                             ; access for all windows that
                                             ; display memory

                                             ; this SYStem.Option is only
                                             ; available for some processor
                                             ; architectures

;…

Go                                           ; Start program execution

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

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

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



B::SYStem.view

Mode
- ○ Down
- ○ NoDebug
- ○ Go
- ○ Attach
- ○ StandBy
- ○ Up (StandBy)
- ● Up

reset
- RESetOut

MemAccess
- ○ DAP
- ○ TSMON3
- ○ RealMON
- ○ TrkMON
- ○ GdbMON
- ● Denied

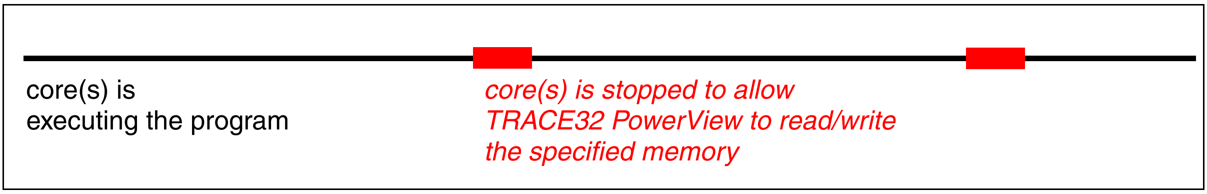
CpuAccess
- ● Enable
- ○ Denied
- ○ Nonstop

Option
- □ IMASKASM
- □ IMASKHLL
- □ TURBO
- □ BigEndian
- ☑ ResBreak
- □ INTDIS
- ☑ DBGACK
- ☑ ShowError
- ☑ EnReset
- □ WaitReset

Option
- □ DACR
- □ MMUSPACES
- □ MPU
- ☑ CFLUSH
- □ CINV
- □ AMBA
- ☑ NODATA
- □ EXEC
- □ SPLIT

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**.



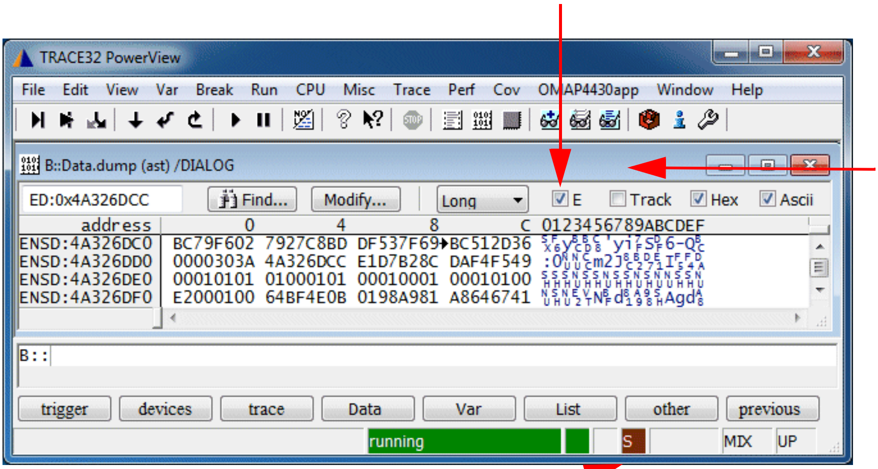
core(s) is executing the program

core(s) is stopped to allow TRACE32 PowerView to read/write the specified memory

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

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

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



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

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

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

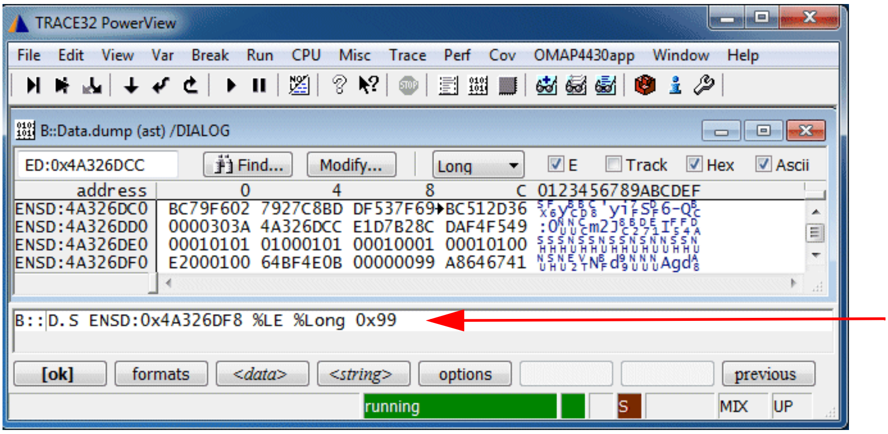
B::Data.dump (ast) /DIALOG

ED:0x4A326DCC   Long   ☑ E   □ Track   ☑ Hex   ☑ Ascii

```
   address    0        4        8        C     0123456789ABCDEF
ENSD:4A326DC0  BC79F602 7927C8BD DF537F69▸BC512D36
ENSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
ENSD:4A326DE0  00010101 01000101 00010001 00010100
ENSD:4A326DF0  E2000100 64BF4E0B 0198A981 A8646741
```

TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
Find... Modify...
B::
trigger devices trace Data Var List other previous
running S MIX UP

Write accesses to the memory work correspondingly:



B::Data.dump (ast) /DIALOG

ED:0x4A326DCC   Long   ☑ E   □ Track   ☑ Hex   ☑ Ascii

```
   address    0        4        8        C     0123456789ABCDEF
ENSD:4A326DC0  BC79F602 7927C8BD DF537F69▸BC512D36
ENSD:4A326DD0  0000303A 4A326DCC E1D7B28C DAF4F549
ENSD:4A326DE0  00010101 01000101 00010001 00010100
ENSD:4A326DF0  E2000100 64BF4E0B 00000099 A8646741
```

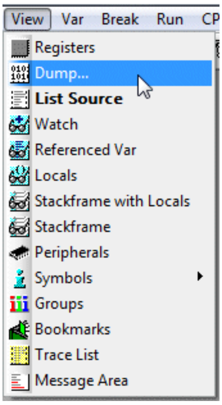
```
B::D.S ENSD:0x4A326DF8 %LE %Long 0x99
```

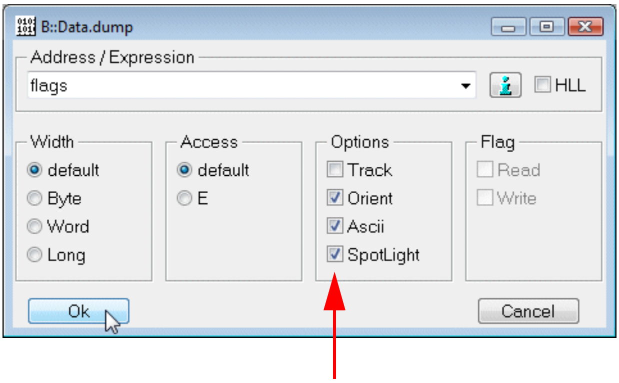
TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
Find... Modify...
[ok] formats <data> <string> options previous
running S MIX UP

`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





Address / Expression: flags □ HLL

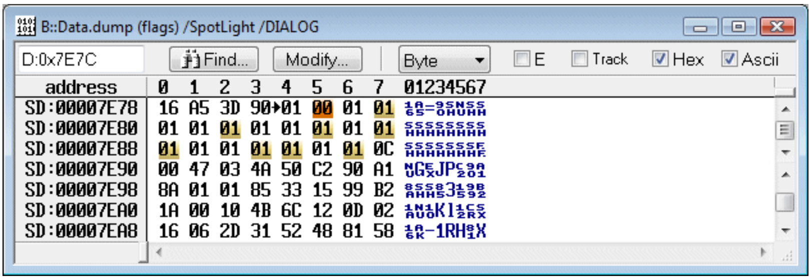
Width: ● default ○ Byte ○ Word ○ Long

Access: ● default ○ E

Options: □ Track ☑ Orient ☑ Ascii ☑ SpotLight

Flag: □ Read □ Write

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



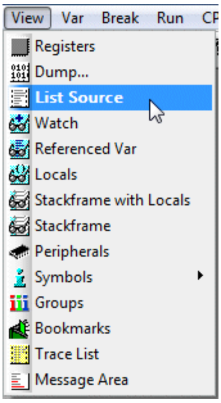
D:0x7E7C Byte □ E □ Track ☑ Hex ☑ Ascii

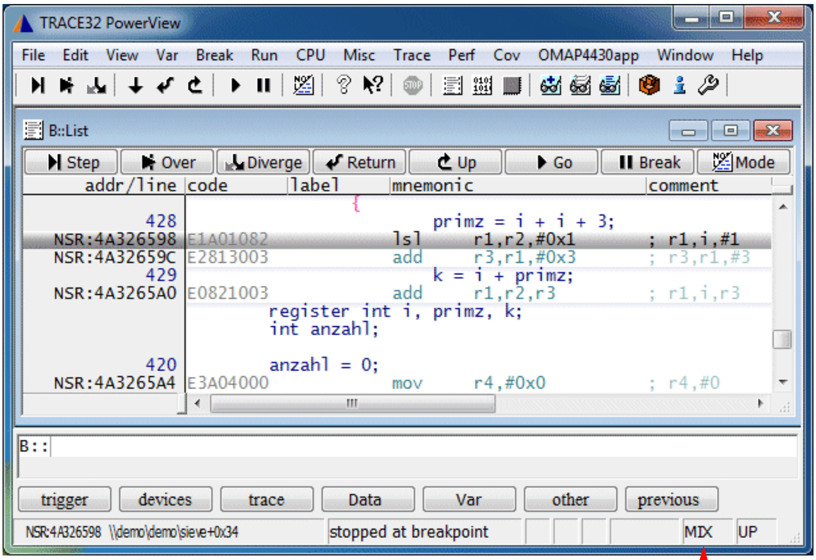
```
  address      0  1  2  3  4  5  6  7
SD:00007E78   16 A5 3D 90▸01 00 01 01
SD:00007E80   01 01 01 01 01 01 01 01
SD:00007E88   01 01 01 01 01 01 01 0C
SD:00007E90   00 47 03 4A 50 C2 90 A1
SD:00007E98   8A 01 01 85 33 15 99 B2
SD:00007EA0   1A 00 10 4B 6C 12 0D 02
SD:00007EA8   16 06 2D 31 52 48 81 58
```

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

                                       ; Mark changes
```

The List Window

Displays the Source Listing Around the PC



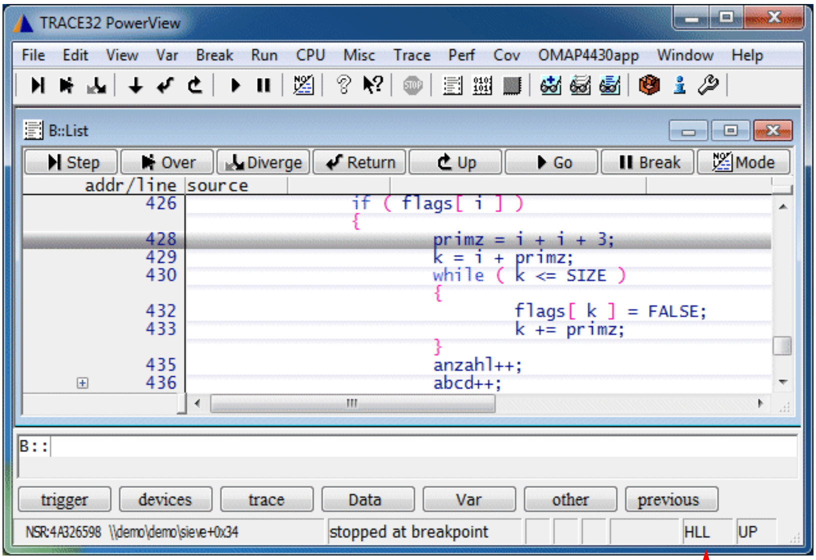


```
     addr/line code      label     mnemonic                   comment
                                   {
           428                           primz = i + i + 3;
  NSR:4A326598 E1A01082            lsl       r1,r2,#0x1       ; r1,i,#1
  NSR:4A32659C E2813003            add       r3,r1,#0x3       ; r3,r1,#3
           429                           k = i + primz;
  NSR:4A3265A0 E0821003            add       r1,r2,r3         ; r1,i,r3
                           register int i, primz, k;
                           int anzahl;

           420             anzahl = 0;
  NSR:4A3265A4 E3A04000            mov       r4,#0x0          ; r4,#0
```

B::

NSR:4A326598 \\demo\demo\sieve+0x34 stopped at breakpoint MIX UP

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



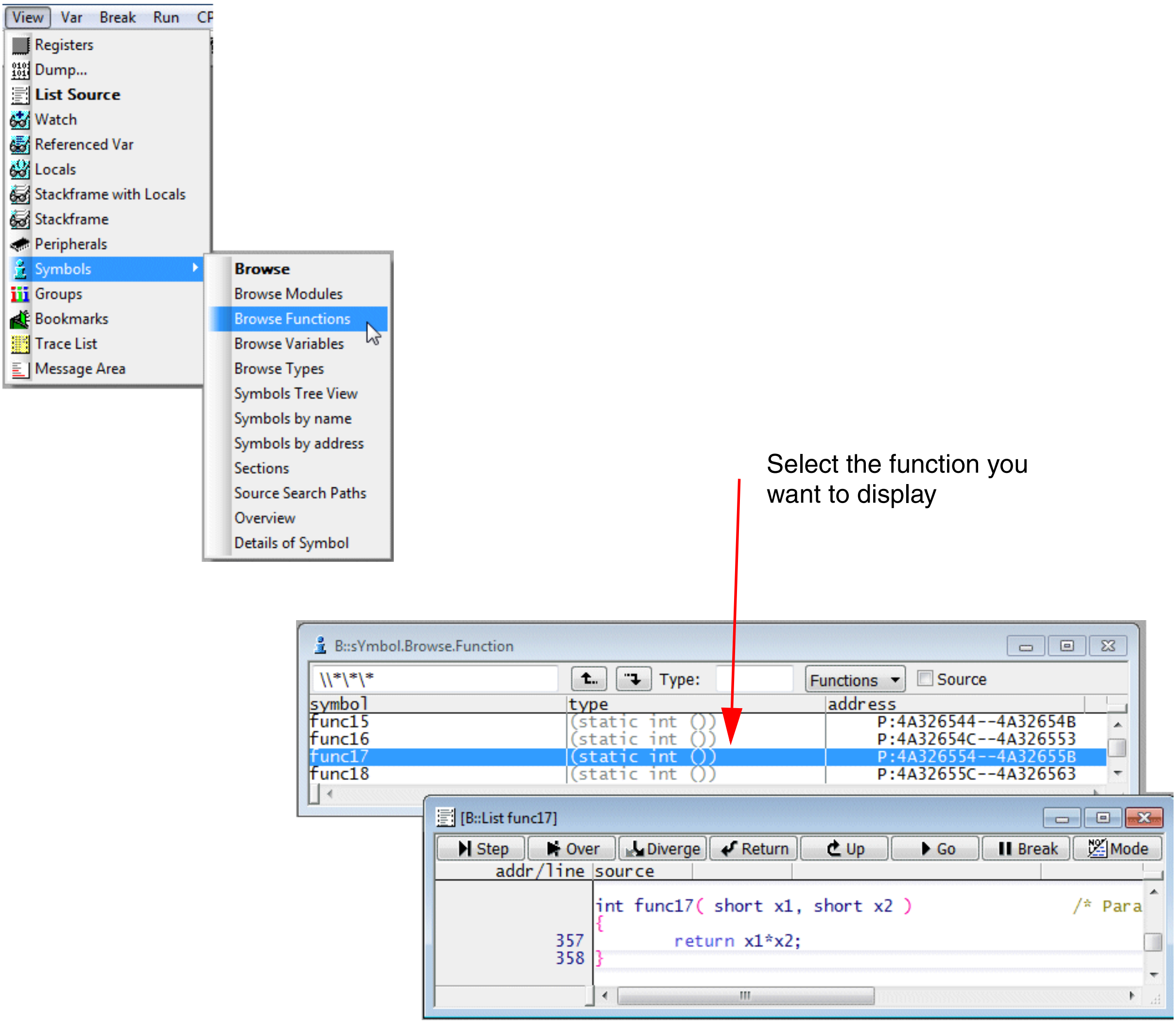
```
  addr/line source
        426            if ( flags[ i ] )
                       {
        428                    primz = i + i + 3;
        429                    k = i + primz;
        430                    while ( k <= SIZE )
                               {
        432                            flags[ k ] = FALSE;
        433                            k += primz;
                               }
        435                    anzahl++;
        436                    abcd++;
```

B::

NSR:4A326598 \\demo\demo\sieve+0x34 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

***

symbol	type	address
func15	(static int ())	P:4A326544--4A32654B
func16	(static int ())	P:4A32654C--4A326553
func17	(static int ())	P:4A326554--4A32655B
func18	(static int ())	P:4A32655C--4A326563

```
addr/line source
          int func17( short x1, short x2 )          /* Para
          {
357           return x1*x2;
358       }
```

List [*<address>*] [**/***<option>*]	Display source listing
Data.List [*<address>*] [**/***<option>*]	Display source listing

```
List                                      ; Display a source listing
                                          ; around the PC

List E:                                   ; Display a source listing,
                                          ; allow scrolling while the
                                          ; program execution is running

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

List func17                               ; Display a source listing of
                                          ; func17
```

Breakpoints

Videos about the breakpoint handling can be found here:
support.lauterbach.com/kb/articles/using-breakpoints-in-trace32

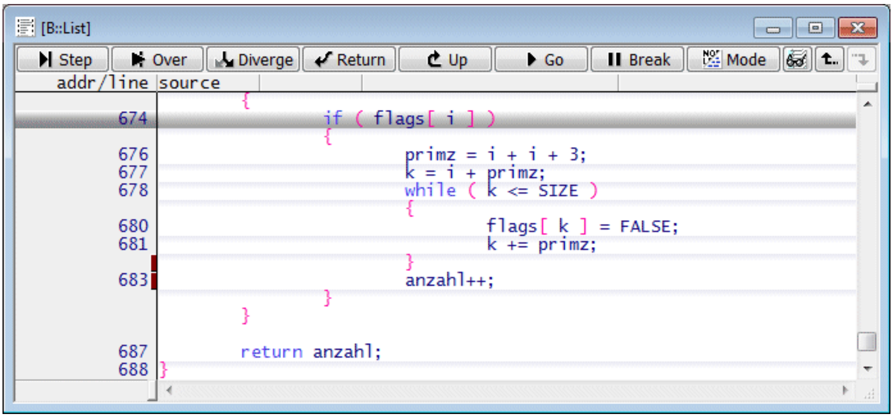
Breakpoint Implementations

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

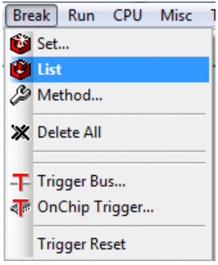
Software Breakpoints in RAM

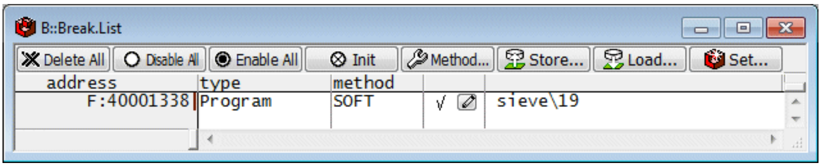
The default implementation for breakpoints on instructions is a Software breakpoint. If a Software breakpoint is set the original instruction at the breakpoint address is patched by a special instruction (usually TRAP) to stop the program and return the control to the debugger.

The number of software breakpoints is unlimited.



```
addr/line source
          {
              if ( flags[ i ] )
              {
                  primz = i + i + 3;
                  k = i + primz;
                  while ( k <= SIZE )
                  {
                      flags[ k ] = FALSE;
                      k += primz;
                  }
                  anzahl++;
              }
          }

          return anzahl;
      }
```





```
 address      type     method
 F:40001338   Program  SOFT     √       sieve\19
```

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

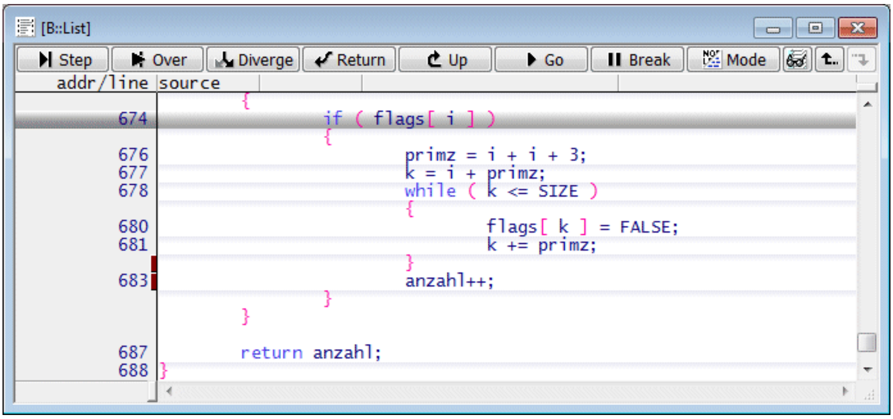
!	Please be aware that TRACE32 PowerView always tries to set an Onchip breakpoint, when the setting of a Software Breakpoint fails.

Software Breakpoints in FLASH

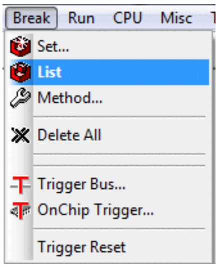
TRACE32 allows to set Software breakpoints to FLASH. Please be aware that the affected FLASH sector has to be erased and programmed in order to patch the break instruction used by the Software breakpoint. This usually takes some time and reduces the number of FLASH erase cycles. For details refer to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

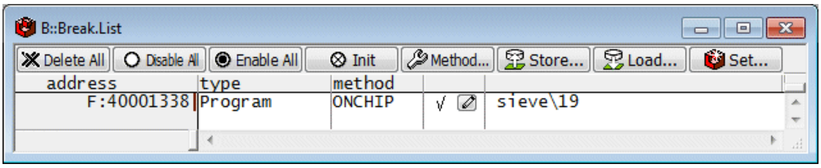
Onchip Breakpoints in NOR Flash

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



```
        {
                if ( flags[ i ] )
                {
                        primz = i + i + 3;
                        k = i + primz;
                        while ( k <= SIZE )
                        {
                                flags[ k ] = FALSE;
                                k += primz;
                        }
                        anzahl++;
                }
        }

        return anzahl;
}
```

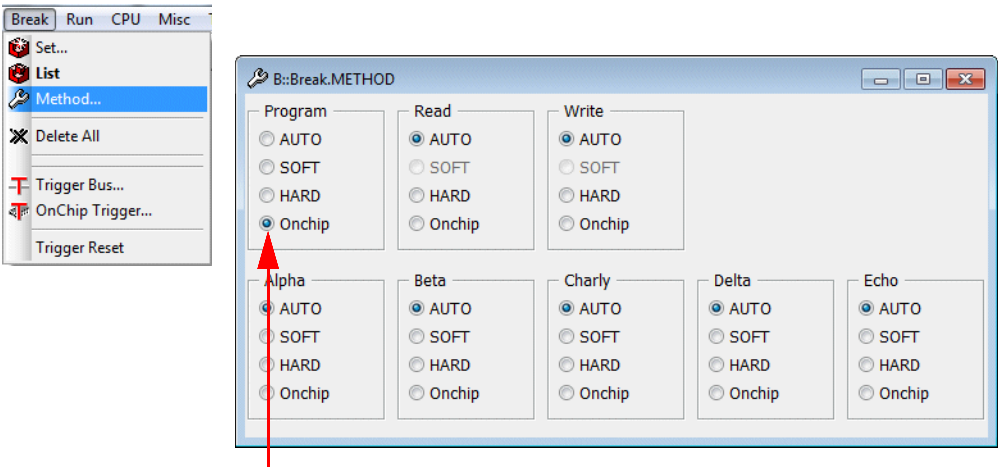




address	type	method		
F:40001338	Program	ONCHIP	√	sieve\19

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

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



Program	Read	Write
○ AUTO	● AUTO	● AUTO
○ SOFT	○ SOFT	○ SOFT
○ HARD	○ HARD	○ HARD
● Onchip	○ Onchip	○ Onchip

Alpha	Beta	Charly	Delta	Echo
● AUTO	● AUTO	● AUTO	● AUTO	● AUTO
○ SOFT	○ SOFT	○ SOFT	○ SOFT	○ SOFT
○ HARD	○ HARD	○ HARD	○ HARD	○ HARD
○ Onchip	○ Onchip	○ Onchip	○ Onchip	○ Onchip

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

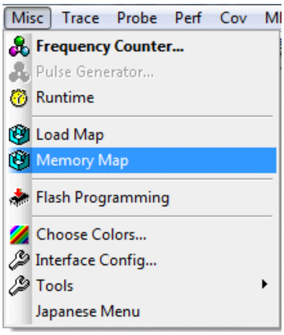
Break.METHOD Program Onchip	Advise TRACE32 PowerView to implement Program breakpoints always as Onchip breakpoints

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

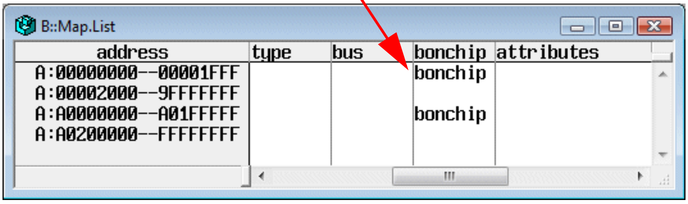
MAP.BOnchip *<range>*	Advise TRACE32 PowerView to implement Program breakpoints as Onchip breakpoints within the defined address range
MAP.List	Check your settings

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

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

Check your settings as follows:



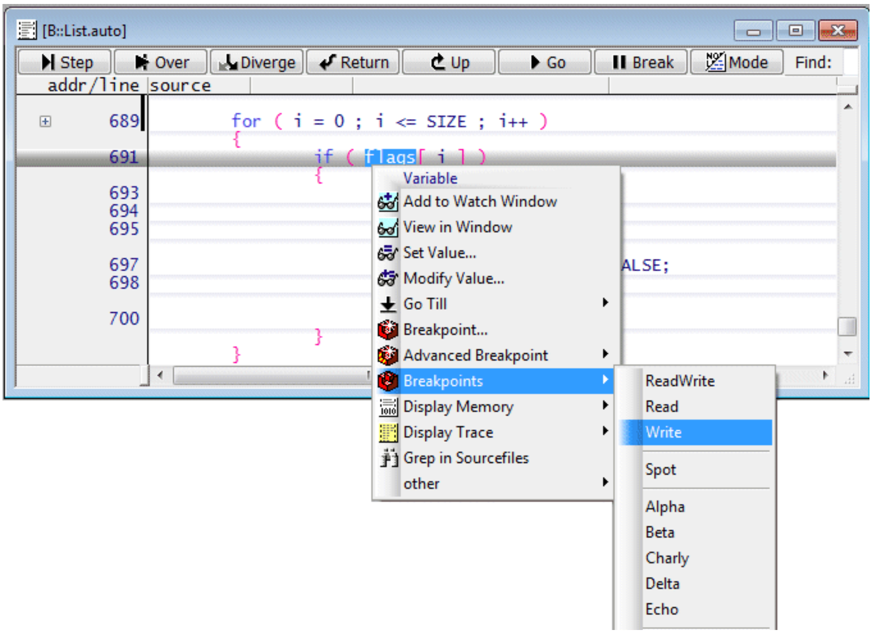
For the specified address ranges Program breakpoints are implemented as Onchip breakpoints. For all other memory areas Software breakpoints are used.



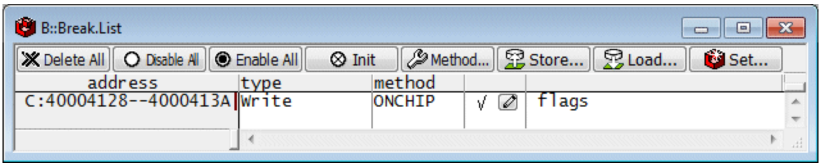
address	type	bus	bonchip	attributes
A:00000000--00001FFF			bonchip	
A:00002000--9FFFFFFF				
A:A0000000--A01FFFFF			bonchip	
A:A0200000--FFFFFFFF				

Onchip Breakpoints on Read/Write Accesses

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



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



address	type	method		
C:40004128--4000413A	Write	ONCHIP	√	flags

Onchip Breakpoints by Processor Architecture

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

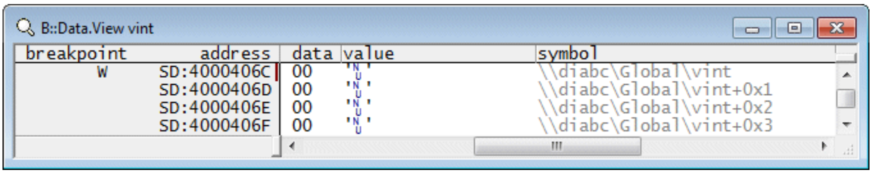
For some processor architectures Onchip breakpoints can only mark **single addresses** (e.g Cortex-A9). Most processor architectures, however, allow to mark **address ranges** with Onchip breakpoints. It is very common that one Onchip breakpoint marks the start address of the address range while the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

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

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

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

Example: the red line in the **Data.View** window shows the range of the Onchip breakpoint.

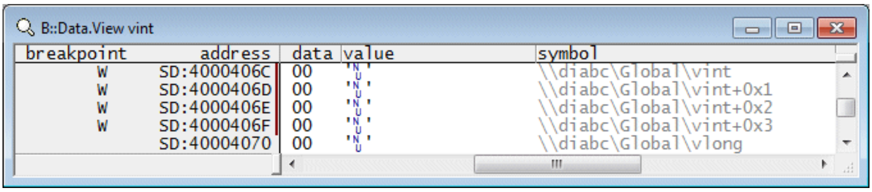


B::Data.View vint

```
breakpoint       address  data value          symbol
         W  SD:4000406C | 00   'NU'           \\diabc\Global\vint
            SD:4000406D   00   'NU'           \\diabc\Global\vint+0x1
            SD:4000406E   00   'NU'           \\diabc\Global\vint+0x2
            SD:4000406F   00   'NU'           \\diabc\Global\vint+0x3
```

```
; Set an Onchip breakpoint to the start address of the variable vint
Break.CONFIG.VarConvert ON
Var.Break.Set vint /Write
Data.View vint
```

```
; Set an Onchip breakpoint to the whole memory range address of the
; variable vint
Break.CONFIG.VarConvert OFF
Var.Break.Set vint /Write
Data.View vin
```



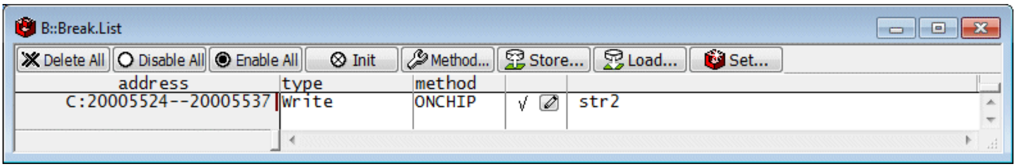
```
B::Data.View vint
breakpoint      address  data value               symbol
         W  SD:4000406C  00   'NU'                \\diabc\Global\vint
         W  SD:4000406D  00   'NU'                \\diabc\Global\vint+0x1
         W  SD:4000406E  00   'NU'                \\diabc\Global\vint+0x2
         W  SD:4000406F  00   'NU'                \\diabc\Global\vint+0x3
            SD:40004070  00   'NU'                \\diabc\Global\vlong
```

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

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

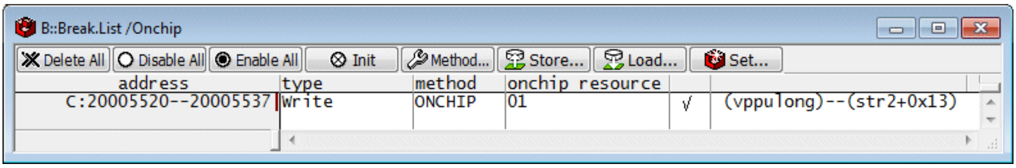
Breakpoint setting:

```
Var.Break.Set str2

Break.List
```



```
B::Break.List
           address        type        method
C:20005524--20005537  Write       ONCHIP       √      str2
```

```
Break.List /Onchip
```



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

ETM Breakpoints for ARM or Cortex-A/-R

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

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

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

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

	Program Breakpoints	Read/Write Breakpoints	Data Value Breakpoints
Cortex-A3x **Cortex-A5x** **Cortex-A6x** **Cortex-A7x** **Cortex-R82** **Cortex-X** **Neoverse**	**Onchip breakpoints:** 6, but only single addresses **ETM breakpoints:** 2 exact address ranges (more on request)	**Onchip breakpoints:** 4, but address range only as bit mask **ETM breakpoints:** —	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** —
Cortex-R52	**Onchip breakpoints:** 8, but only single addresses **ETM breakpoints:** up to 2 exact address ranges	**Onchip breakpoints:** 8, but address range only as bit mask **ETM breakpoints:** —	**Onchip breakpoints:** no data value breakpoints possible **ETM breakpoints:** —

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

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

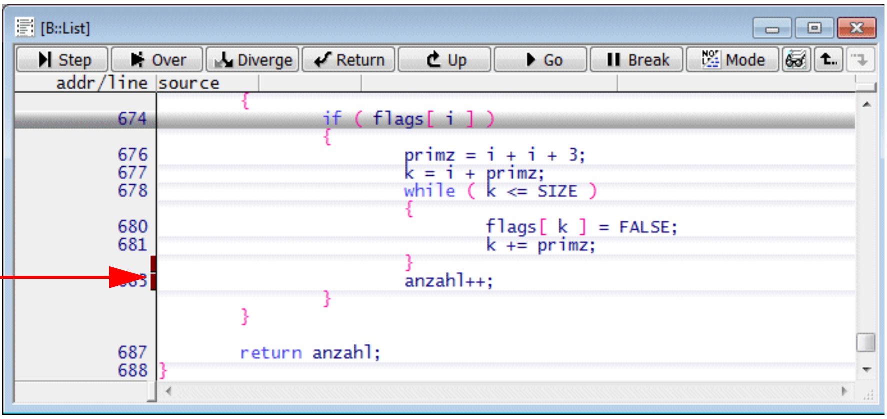
Breakpoint Types

TRACE32 PowerView provides the following breakpoint types for standard debugging.

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

Program Breakpoints

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

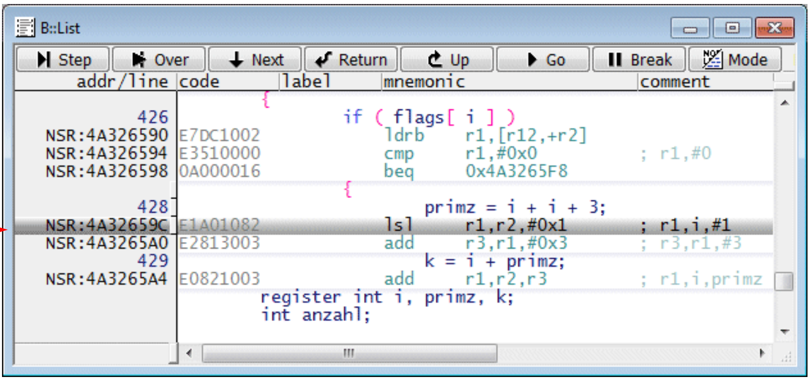


```
addr/line source
                  {
      674                 if ( flags[ i ] )
                          {
      676                         primz = i + i + 3;
      677                         k = i + primz;
      678                         while ( k <= SIZE )
                                  {
      680                                 flags[ k ] = FALSE;
      681                                 k += primz;
                                  }
      683                         anzahl++;
                          }
                  }
      687         return anzahl;
      688 }
```

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).

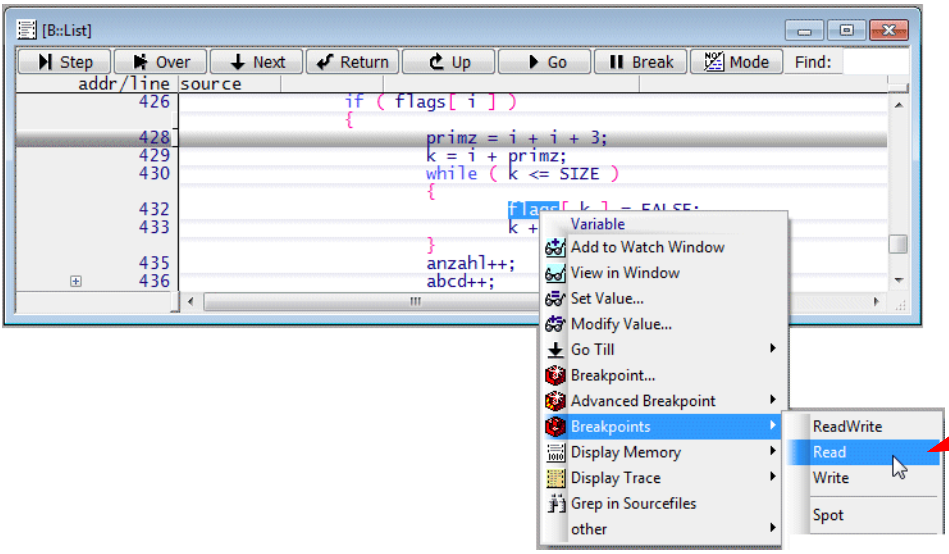
Disable the Program breakpoint by a left mouse double-click to the red program breakpoint indicator. The program breakpoint indicator becomes grey.



```
addr/line       code     label    mnemonic                    comment
                         {
            426                  if ( flags[ i ] )
NSR:4A326590    E7DC1002          ldrb     r1,[r12,+r2]
NSR:4A326594    E3510000          cmp      r1,#0x0            ; r1,#0
NSR:4A326598    0A000016          beq      0x4A3265F8
                                 {
            428                          primz = i + i + 3;
NSR:4A32659C    E1A01082          lsl      r1,r2,#0x1         ; r1,i,#1
NSR:4A3265A0    E2813003          add      r3,r1,#0x3         ; r3,r1,#3
            429                          k = i + primz;
NSR:4A3265A4    E0821003          add      r1,r2,r3           ; r1,i,primz
                         register int i, primz, k;
                         int anzahl;
```

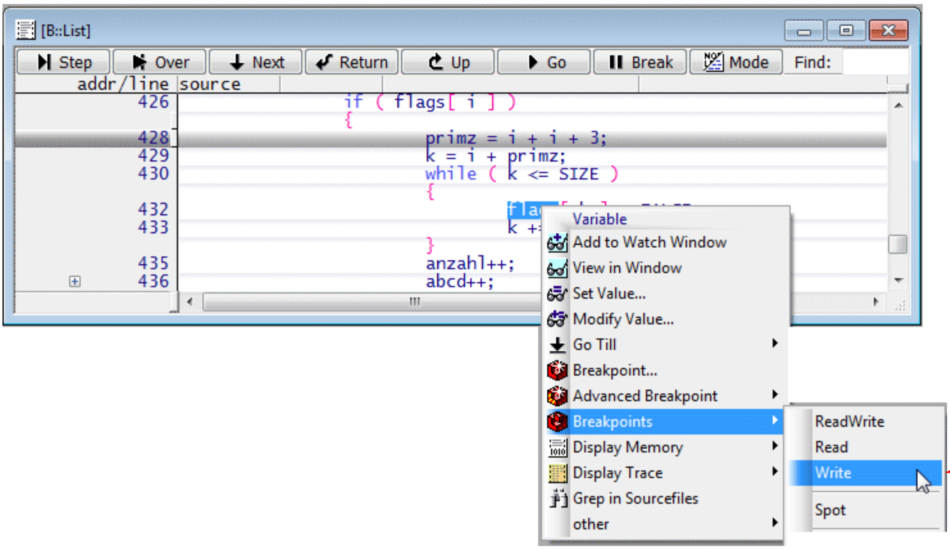
Break.Set *<address>* **/Program** [**/DISable**] Set a Program breakpoint to the specified address. The Program breakpoint can be disabled if required.

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

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

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

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

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

Break.List                            ; list all breakpoints
```



```
addr/line source
426                      if ( flags[ i ] )
                         {
428                              primz = i + i + 3;
429                              k = i + primz;
430                              while ( k <= SIZE )
                                 {
432                                       flags[ k ] = FALSE;
433                                       k +
                                 }
435                              anzahl++;
436                              abcd++;
```

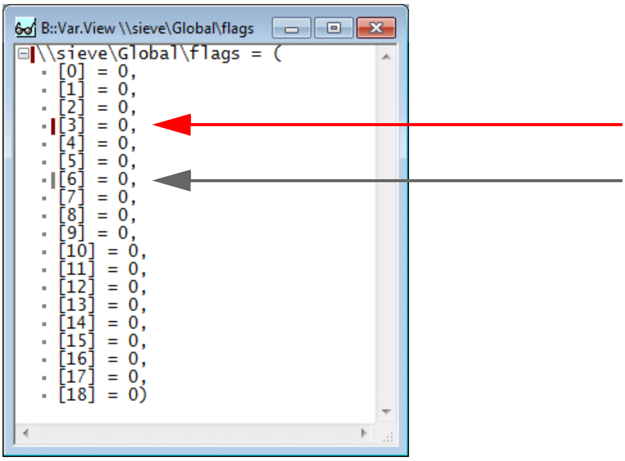
Core stops at a read access to the variable



```
addr/line source
426                      if ( flags[ i ] )
                         {
428                              primz = i + i + 3;
429                              k = i + primz;
430                              while ( k <= SIZE )
                                 {
432                                       flag
433                                       k +
                                 }
435                              anzahl++;
436                              abcd++;
```

Core stops at a write access to the variable

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



```
\\sieve\Global\flags = (
  [0] = 0,
  [1] = 0,
  [2] = 0,
  [3] = 0,
  [4] = 0,
  [5] = 0,
  [6] = 0,
  [7] = 0,
  [8] = 0,
  [9] = 0,
  [10] = 0,
  [11] = 0,
  [12] = 0,
  [13] = 0,
  [14] = 0,
  [15] = 0,
  [16] = 0,
  [17] = 0,
  [18] = 0)
```

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

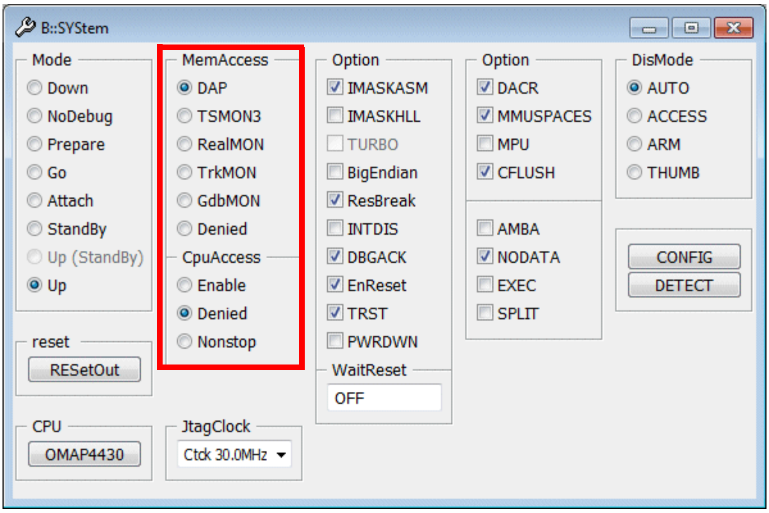
A small **grey breakpoint indicator**
marks a disabled Read/Write breakpoint.

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

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

```
Break.Set 0x0B56 /Read

Break.Set ast /Write

Break.Set vpchar+5 /ReadWrite /DISable

Var.Break.Set flags /Write

Var.Break.Set flags[3] /Read

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

Break.List
```

Breakpoint Setting at Run-time



B::SYStem

Mode
○ Down
○ NoDebug
○ Prepare
○ Go
○ Attach
○ StandBy
○ Up (StandBy)
● Up

MemAccess
● DAP
○ TSMON3
○ RealMON
○ TrkMON
○ GdbMON
○ Denied

CpuAccess
○ Enable
● Denied
○ Nonstop

Option
☑ IMASKASM
□ IMASKHLL
□ TURBO
□ BigEndian
☑ ResBreak
□ INTDIS
☑ DBGACK
☑ EnReset
☑ TRST
□ PWRDWN

WaitReset
OFF

Option
☑ DACR
☑ MMUSPACES
□ MPU
☑ CFLUSH
□ AMBA
☑ NODATA
□ EXEC
□ SPLIT

DisMode
● AUTO
○ ACCESS
○ ARM
○ THUMB

reset
RESetOut

CPU
OMAP4430

JtagClock
Ctck 30.0MHz

Software breakpoints

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

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

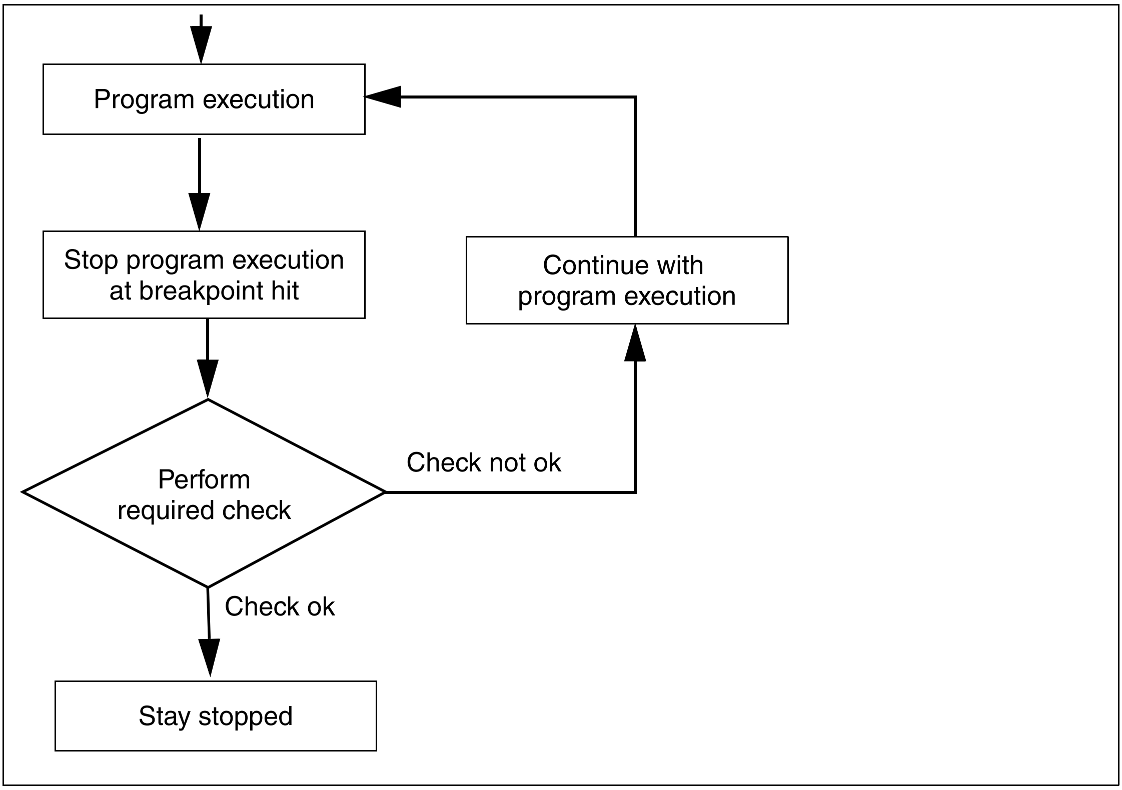
Real-time Breakpoints vs. Intrusive Breakpoints

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

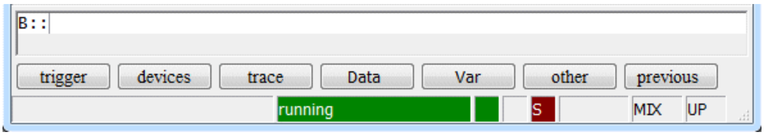
Real-time breakpoints do not disturb the real-time program execution on the core(s), but they require a complex on-chip break logic.

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

Intrusive breakpoint perform as follows:


Program execution
Stop program execution at breakpoint hit
Continue with program execution
Perform required check
Check not ok
Check ok
Stay stopped

Each stop to perform the check suspends the program execution for at least 1 ms. For details refer to **“StopAndGo Mode”** (glossary.pdf)

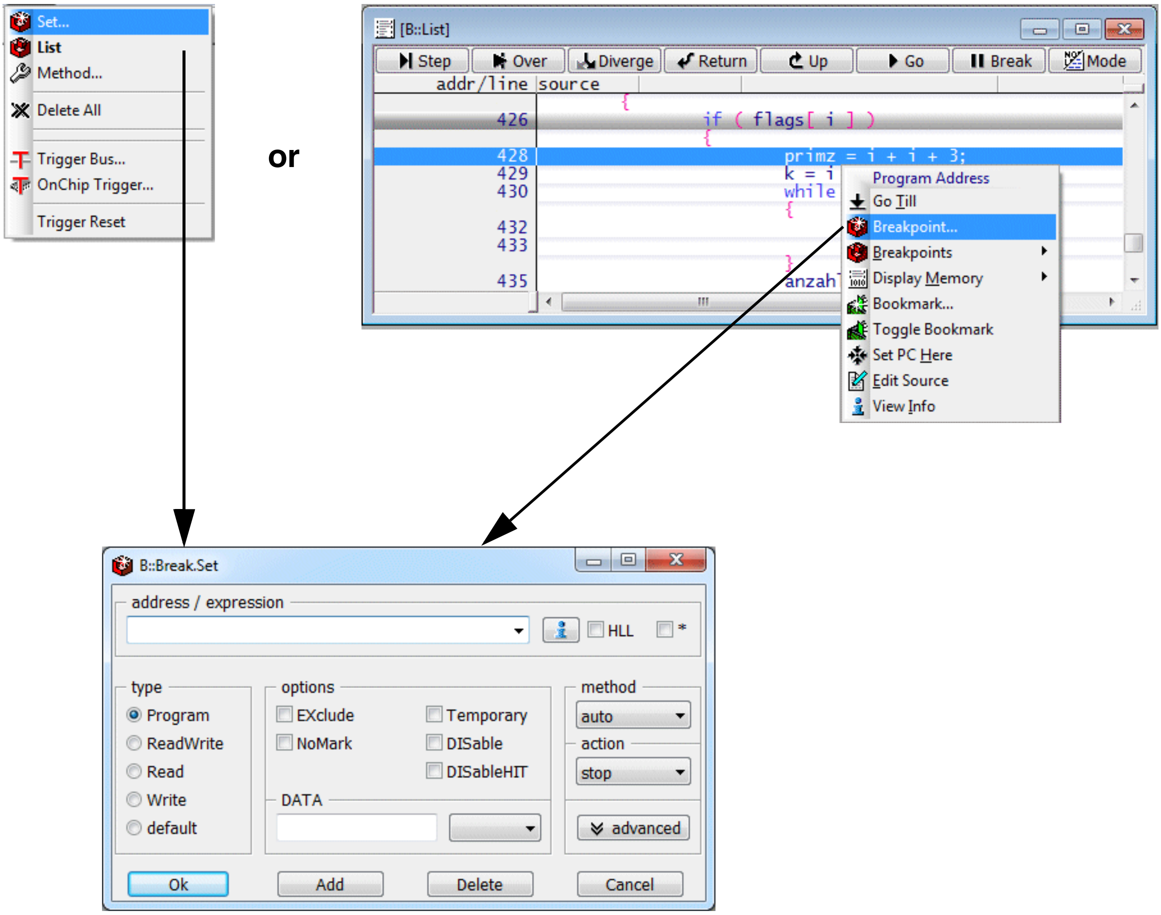

```
B::
```

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

Intrusive breakpoints are marked with a special breakpoint indicator: ▨

Break.Set Dialog Box

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



```
426            if ( flags[ i ] )
               {
428                primz = i + i + 3;
429                k = i
430                while
                   {
432
433
                   }
435                anzahl
```

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

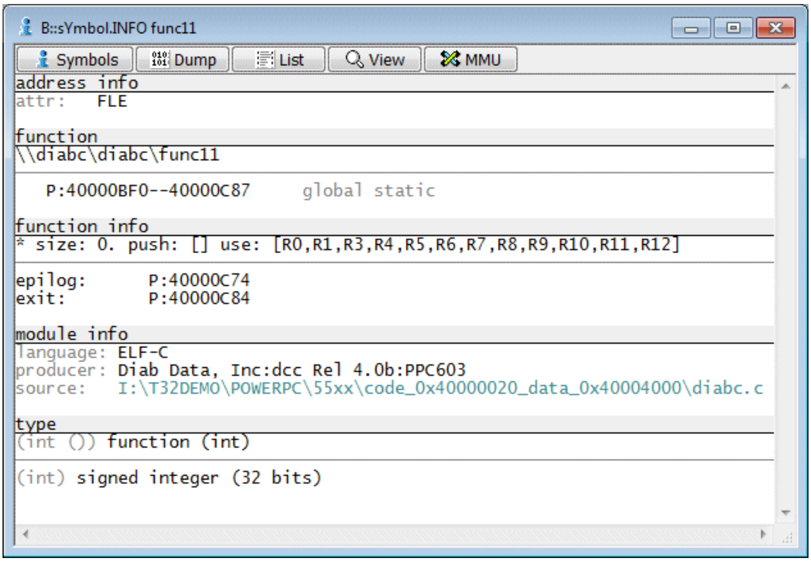
options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

method: auto

action: stop

The HLL Check Box - Function Name

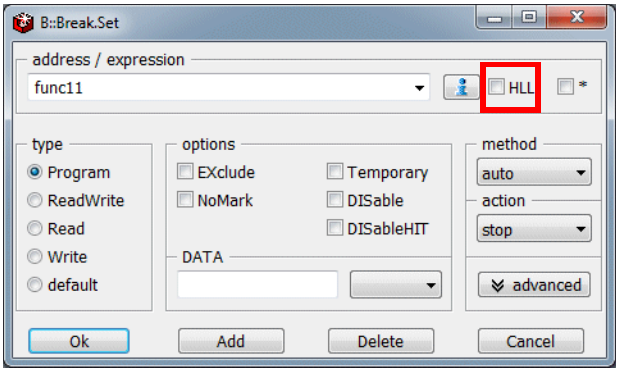
```
sYmbol.INFO func11                    ; display symbol information
                                      ; for function func11
```



```
address info
attr:    FLE

function
\\diabc\diabc\func11

   P:40000BF0--40000C87      global static

function info
* size: 0. push: [] use: [R0,R1,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12]

epilog:      P:40000C74
exit:        P:40000C84

module info
language: ELF-C
producer: Diab Data, Inc:dcc Rel 4.0b:PPC603
source:   I:\T32DEMO\POWERPC\55xx\code_0x40000020_data_0x40004000\diabc.c

type
(int ()) function (int)

(int) signed integer (32 bits)
```

Function Name/HLL Check Box OFF

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



address / expression: func11   □ HLL   □ *

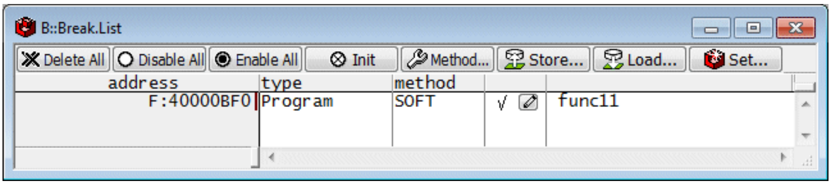
type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

DATA:

method: auto

action: stop

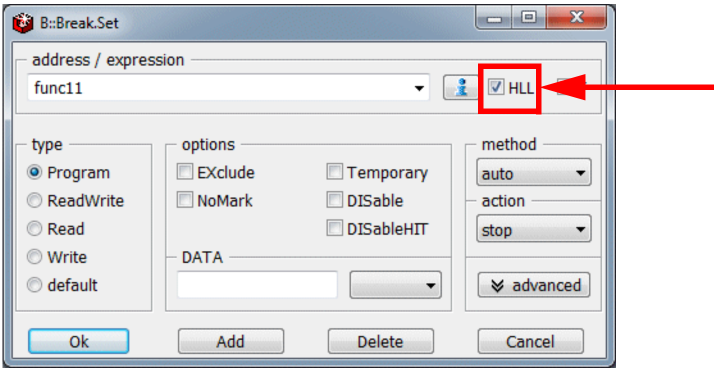


address	type	method		
F:40000BF0	Program	SOFT	√	func11

```
Break.Set func11
```

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

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



B::Break.Set

address / expression: func11 ☑ HLL

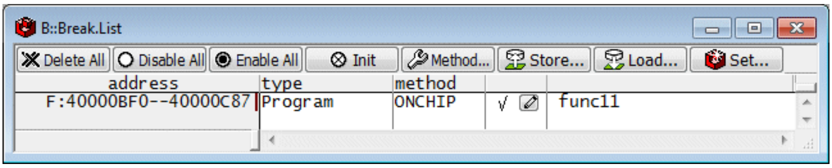
type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

DATA:

method: auto

action: stop



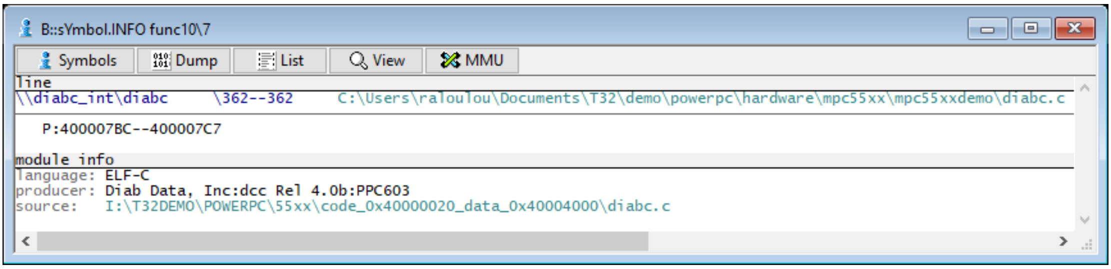
B::Break.List

address	type	method		
F:40000BF0--40000C87	Program	ONCHIP	√	func11

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

The HLL Check Box - Program Line Number

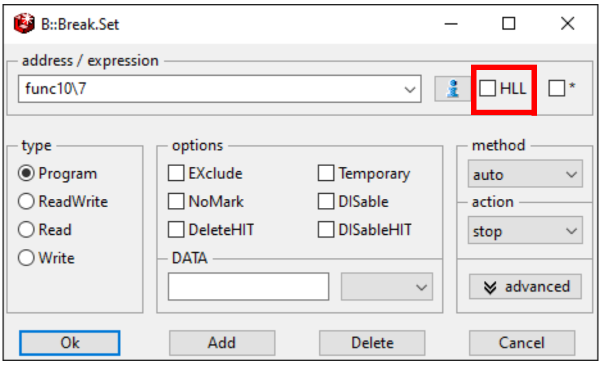
```
sYmbol.INFO func10\7                    ; display debug information
                                        ; for 7th program line in
                                        ; function func10
```



```
line
\\diabc_int\diabc     \362--362     C:\Users\raloulou\Documents\T32\demo\powerpc\hardware\mpc55xx\mpc55xxdemo\diabc.c

   P:400007BC--400007C7

module info
language: ELF-C
producer: Diab Data, Inc:dcc Rel 4.0b:PPC603
source:   I:\T32DEMO\POWERPC\55xx\code_0x40000020_data_0x40004000\diabc.c
```

Program Line Number/HLL Check Box OFF

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



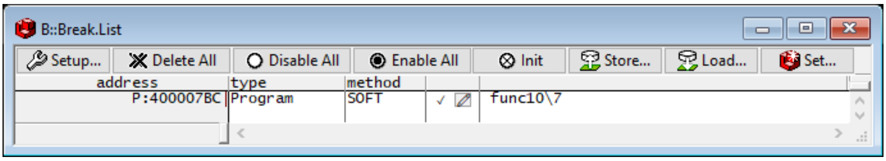
address / expression: func10\7 □ HLL □ *

type: ● Program ○ ReadWrite ○ Read ○ Write

options: □ EXclude □ Temporary □ NoMark □ DISable □ DeleteHIT □ DISableHIT

method: auto

action: stop



```
      address  type     method
   P:400007BC  Program  SOFT      ✓      func10\7
```

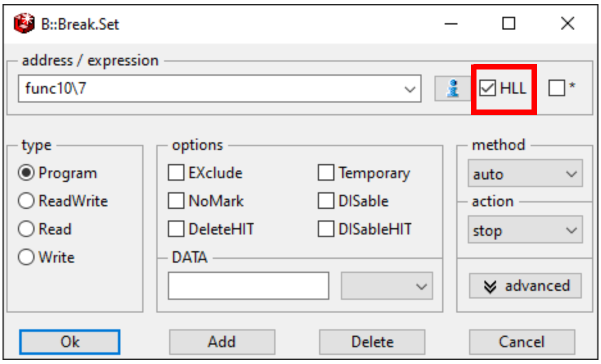
```
Break.Set func10\7
```

Program Line Number/HLL Check Box ON

- If the on-chip break logic supports ranges for Program breakpoints, a Program breakpoint implemented as Onchip is set to the full address range covered by all assembler instructions generated for the program line number.
- If the on-chip break logic provides only bitmasks to realizes breakpoints on instruction ranges, a Program breakpoint implemented as Onchip is set by using the smallest bitmask that covers the

complete address range of the program line.

- otherwise this breakpoint is rejected with an error message.



B::Break.Set

address / expression: func10\7 ☑ HLL □ *

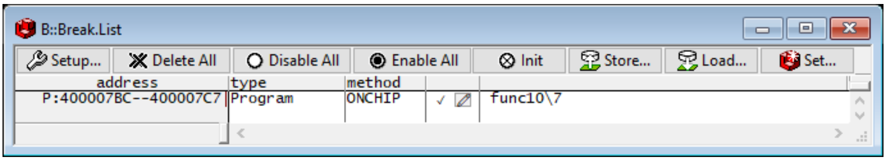
type: ● Program ○ ReadWrite ○ Read ○ Write

options: □ EXclude □ Temporary □ NoMark □ DISable □ DeleteHIT □ DISableHIT

DATA

method: auto

action: stop

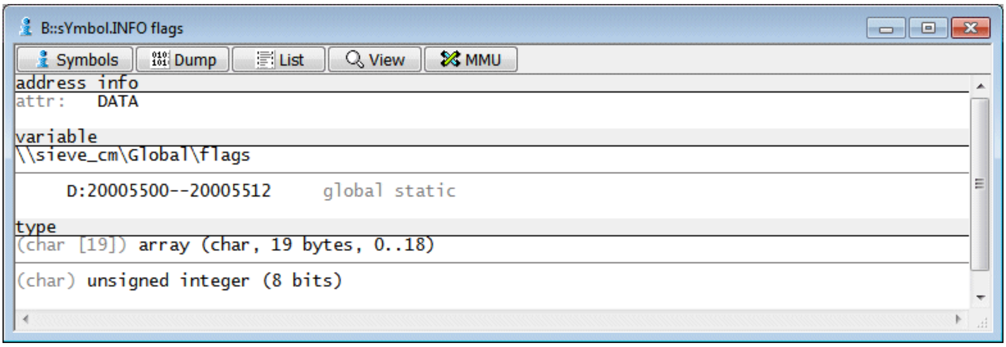


B::Break.List

address	type	method		
P:400007BC--400007C7	Program	ONCHIP	✓	func10\7

The HLL Check Box - Variable

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

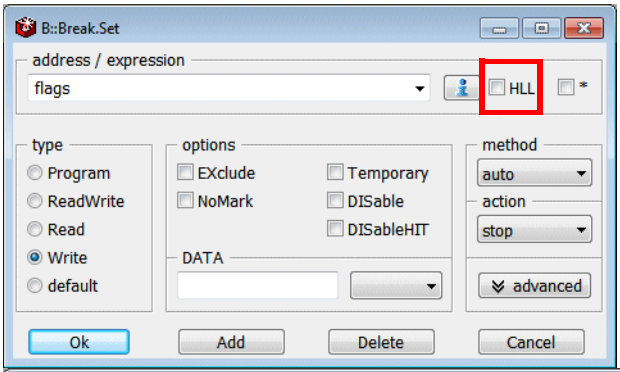


B::sYmbol.INFO flags

```
address info
attr:    DATA

variable
\\sieve_cm\Global\flags

     D:20005500--20005512      global static

type
(char [19]) array (char, 19 bytes, 0..18)

(char) unsigned integer (8 bits)
```

Variable/HLL Check Box OFF

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



B::Break.Set

address / expression: flags □ HLL □ *

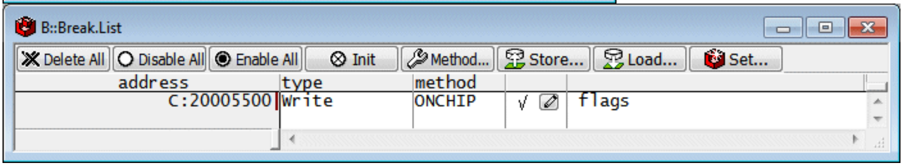
type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

DATA:

method: auto

action: stop



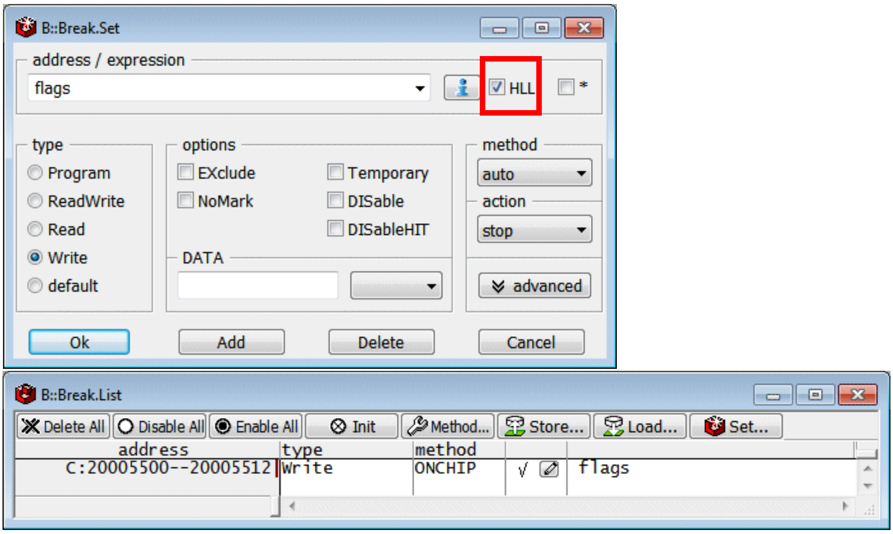
B::Break.List

address	type	method		
C:20005500	Write	ONCHIP	√	flags

```
Break.Set flags
```

Variable/HLL Check Box ON

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



B::Break.Set

address / expression: flags ☑ HLL □ *

type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ Temporary □ NoMark □ DISable □ DISableHIT

DATA

method: auto

action: stop

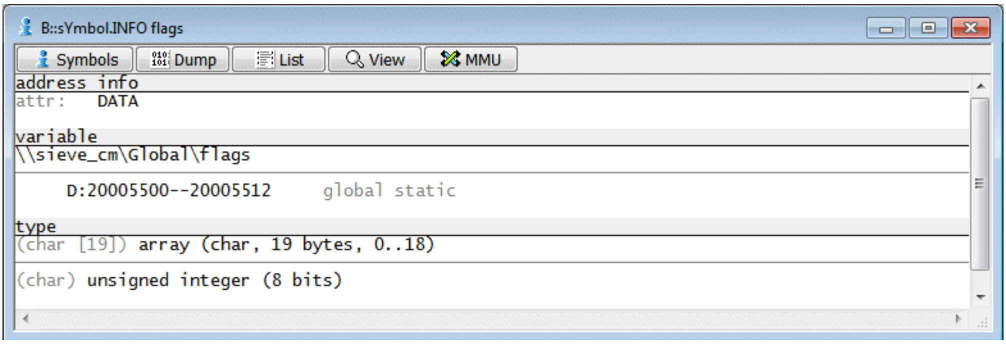
B::Break.List

```
       address        type          method
C:20005500--20005512 Write          ONCHIP    √     flags
```

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

The HLL Check Box - HLL Expression

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

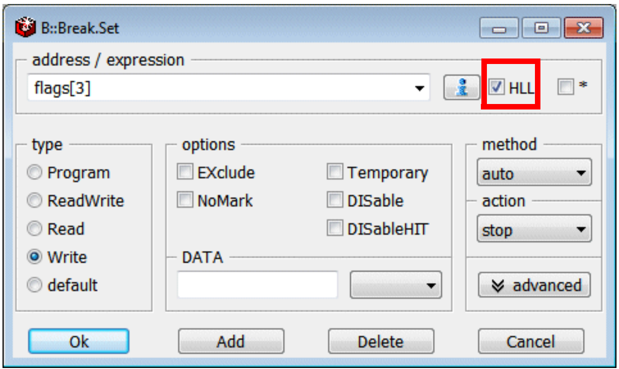


```
address info
attr:    DATA

variable
\\sieve_cm\Global\flags

     D:20005500--20005512      global static

type
(char [19]) array (char, 19 bytes, 0..18)

(char) unsigned integer (8 bits)
```

Variable/HLL Check Box Must Be ON

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

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



address / expression: flags[3] ☑ HLL □ *

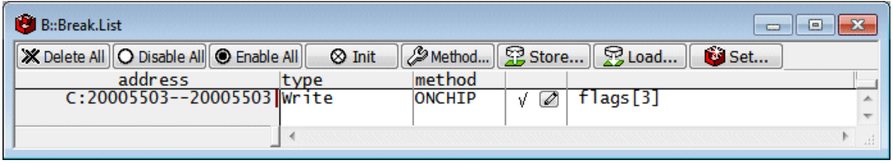
type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

DATA:

method: auto

action: stop

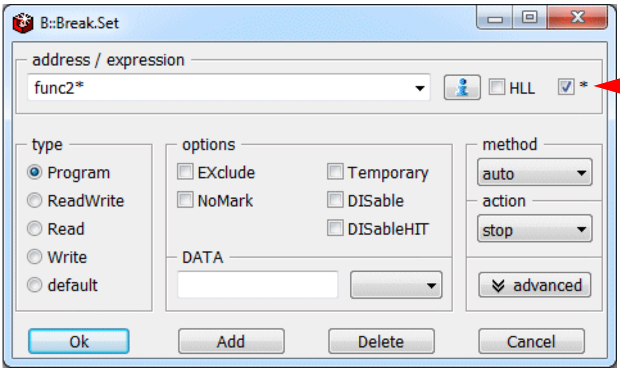


```
         address            type     method
 C:20005503--20005503       Write    ONCHIP     √     flags[3]
```

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

Allow Wildcards in address/expression

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



address / expression: func2*   ☐ HLL   ☑ *

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

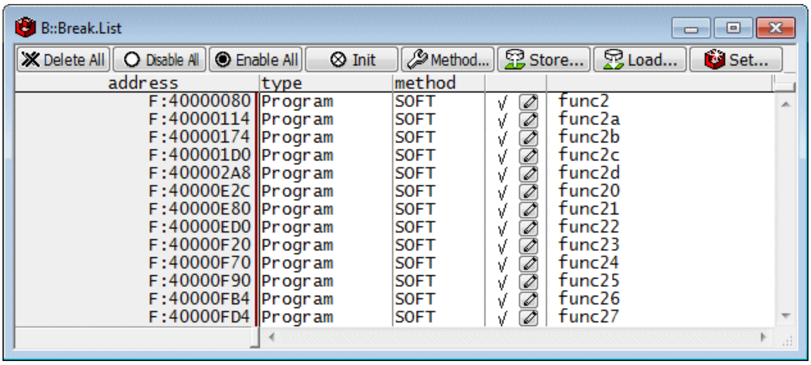
options: ☐ EXclude ☐ NoMark ☐ Temporary ☐ DISable ☐ DISableHIT

DATA:

method: auto

action: stop

Check * to enable wildcard usage

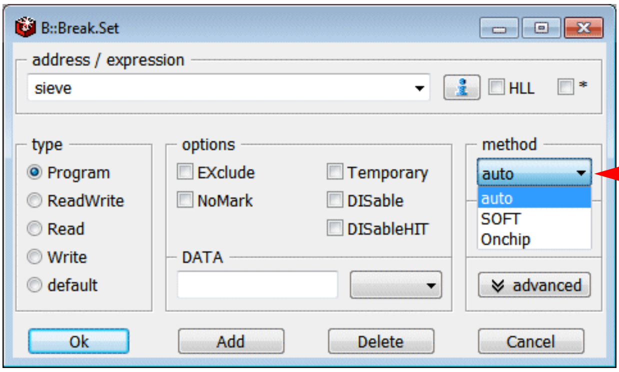


address	type	method		
F:40000080	Program	SOFT	√	func2
F:40000114	Program	SOFT	√	func2a
F:40000174	Program	SOFT	√	func2b
F:400001D0	Program	SOFT	√	func2c
F:400002A8	Program	SOFT	√	func2d
F:40000E2C	Program	SOFT	√	func20
F:40000E80	Program	SOFT	√	func21
F:40000ED0	Program	SOFT	√	func22
F:40000F20	Program	SOFT	√	func23
F:40000F70	Program	SOFT	√	func24
F:40000F90	Program	SOFT	√	func25
F:40000FB4	Program	SOFT	√	func26
F:40000FD4	Program	SOFT	√	func27

Requires sufficient resources if Onchip breakpoints are used.

```
Break.SetPATtern func2*
```

Implementations



B::Break.Set

address / expression: sieve □ HLL □ *

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

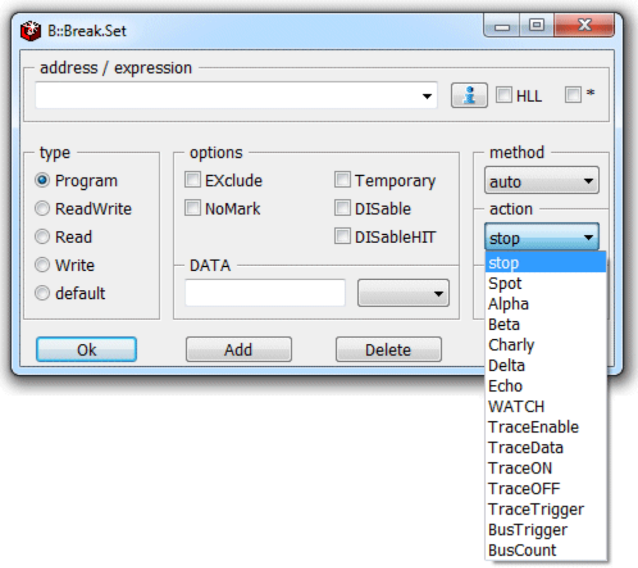
DATA

method: auto

Implementation

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

Actions



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

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

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

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

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

- **"Training Arm CoreSight ETM Tracing"** (training_arm_etm.pdf).
- **"Training Cortex-M Tracing"** (training_cortexm_etm.pdf).
- **"Training AURIX Tracing"** (training_aurix_trace.pdf).
- **"Training Hexagon ETM Tracing"** (training_hexagon_etm.pdf).
- **"Training Nexus Tracing"** (training_nexus.pdf).

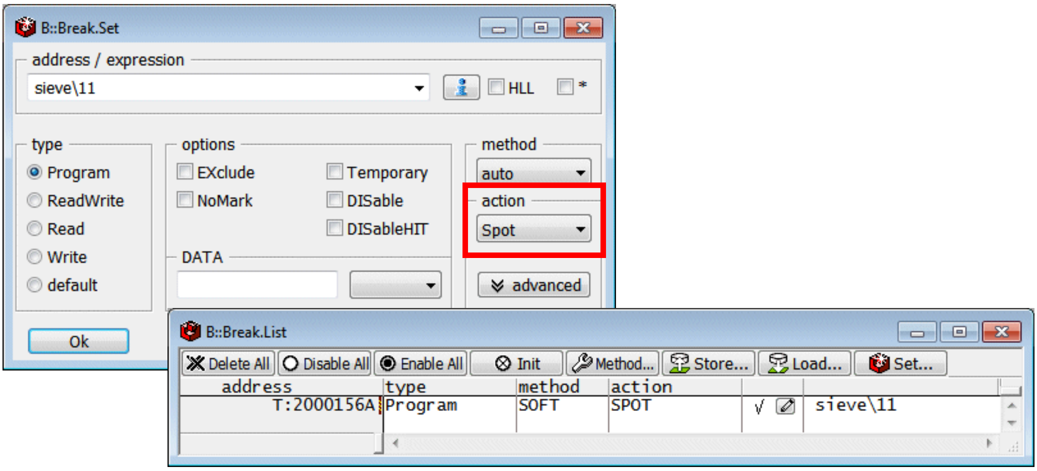
or with the description of the **Break.Set** command.

Example for the Action Spot

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

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

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



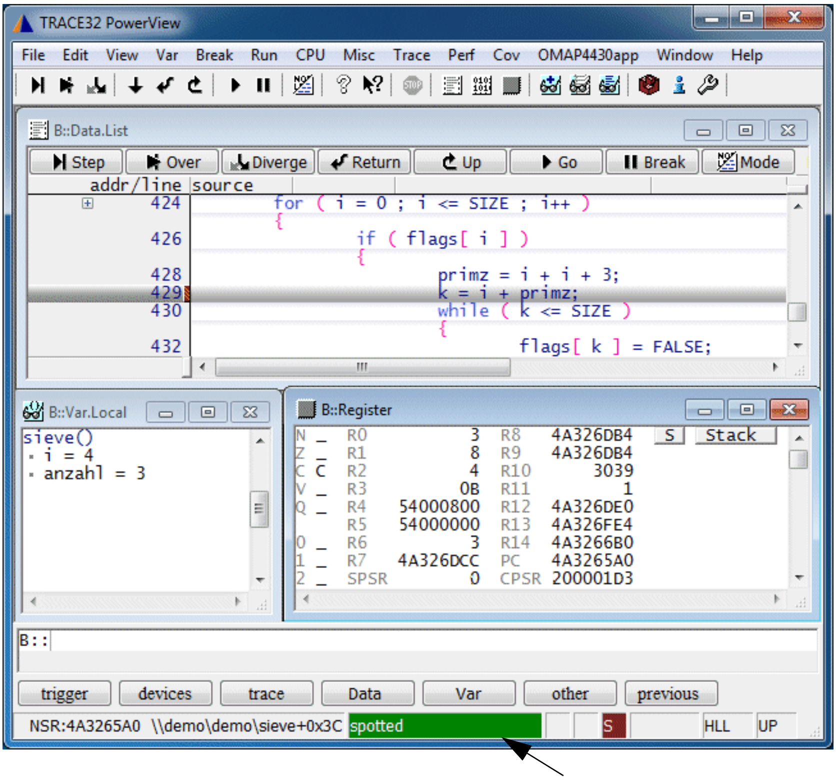
address / expression: sieve\11

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

method: auto

action: Spot

address	type	method	action		
T:2000156A	Program	SOFT	SPOT	√	sieve\11


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
B::Data.List
Step Over Diverge Return Up Go Break Mode
B::Var.Local
B::Register
S Stack
trigger devices trace Data Var other previous

```
addr/line source
     424        for ( i = 0 ; i <= SIZE ; i++ )
                {
     426                if ( flags[ i ] )
                        {
     428                        primz = i + i + 3;
     429                        k = i + primz;
     430                        while ( k <= SIZE )
                                {
     432                                flags[ k ] = FALSE;
```

```
sieve()
  i = 4
  anzahl = 3
```

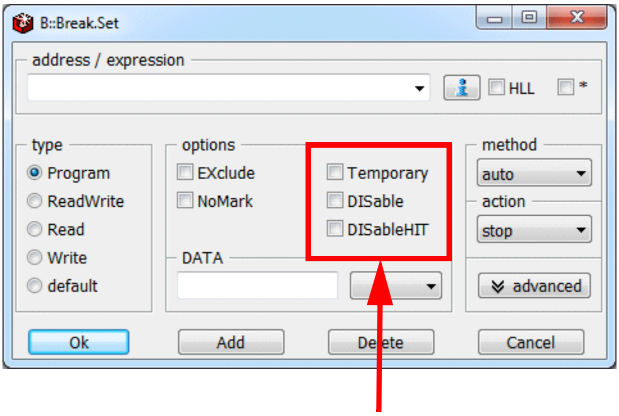
```
N _   R0          3   R8    4A326DB4
Z _   R1          8   R9    4A326DB4
C C   R2          4   R10       3039
V _   R3         0B   R11          1
Q _   R4   54000800   R12   4A326DE0
      R5   54000000   R13   4A326FE4
0 _   R6          3   R14   4A3266B0
1 _   R7   4A326DCC   PC    4A3265A0
2 _   SPSR        0   CPSR  200001D3
```

```
B::
```

```
NSR:4A3265A0  \\demo\demo\sieve+0x3C  spotted    S    HLL  UP
```

spotted indicates a breakpoint with the action Spot

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



Options

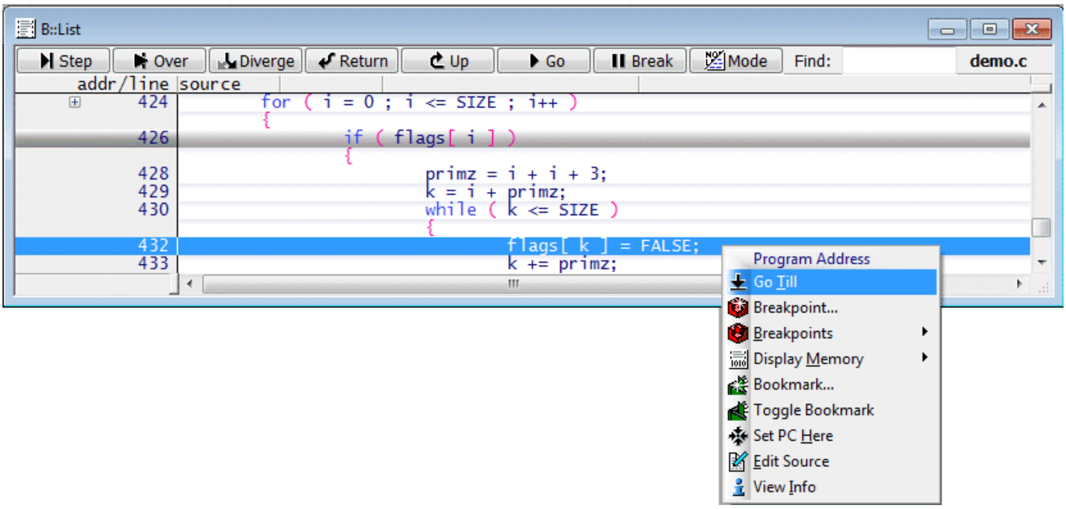
Temporary	**OFF:** Set a permanent breakpoint (default). **ON:** Set a temporary breakpoint. All temporary breakpoints are deleted the next time the core(s) stops the program execution.
DISable	**OFF:** Breakpoint is enabled (default). **ON:** Set breakpoint, but disabled.
DISableHIT	**ON:** Disable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

Example for the Option Temporary

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

Examples:

- **Go Till**

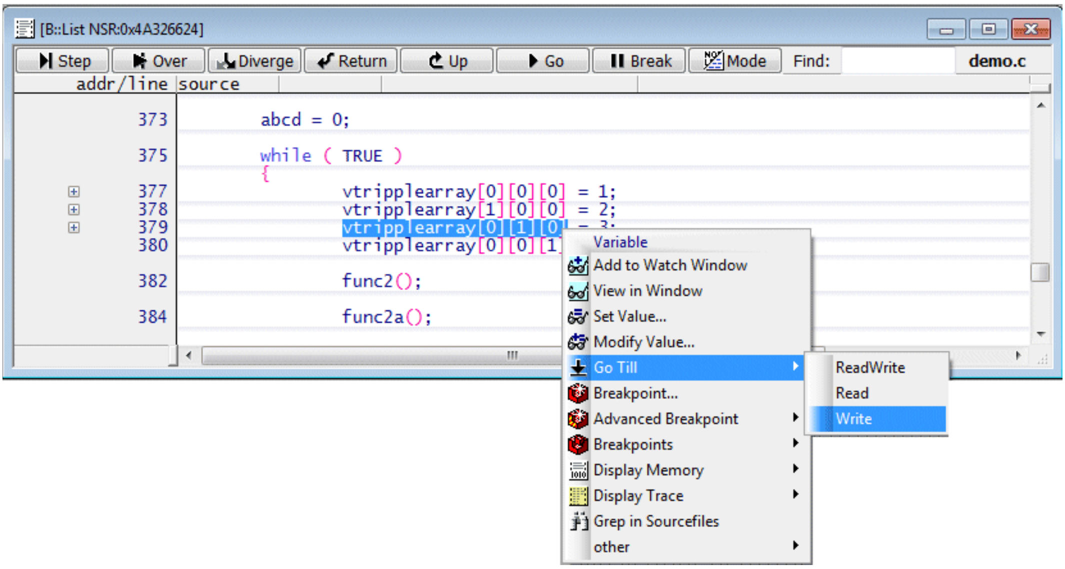


```
for ( i = 0 ; i <= SIZE ; i++ )
{
        if ( flags[ i ] )
        {
                primz = i + i + 3;
                k = i + primz;
                while ( k <= SIZE )
                {
                        flags[ k ] = FALSE;
                        k += primz;
```

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**

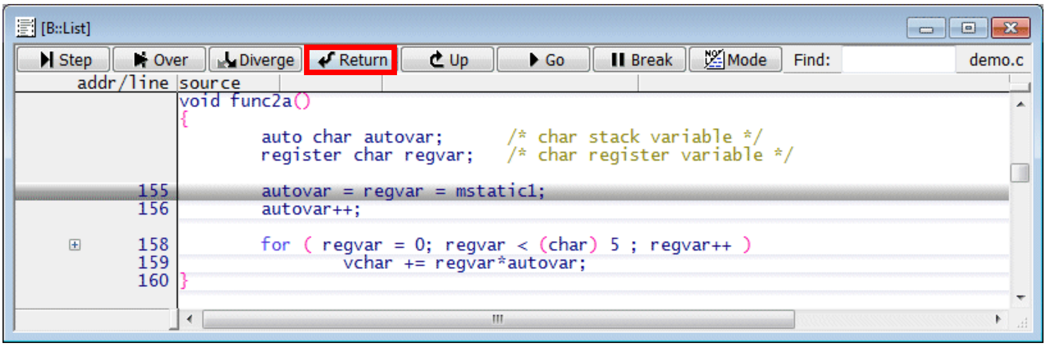


```
373         abcd = 0;

375         while ( TRUE )
            {
377                 vtripplearray[0][0][0] = 1;
378                 vtripplearray[1][0][0] = 2;
379                 vtripplearray[0][1][0] = 3;
380                 vtripplearray[0][0][1]

382                 func2();

384                 func2a();
```

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

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

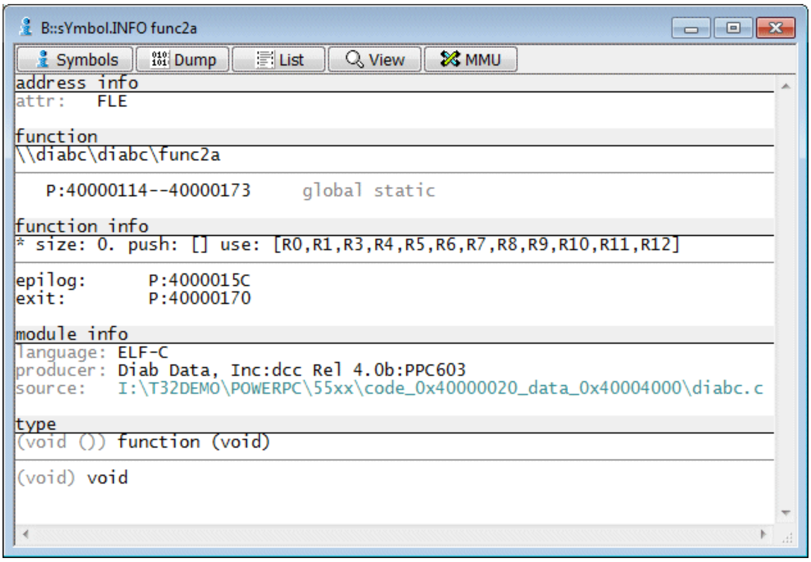
- **Go.Return and similar commands**



```
addr/line source
          void func2a()
          {
                  auto char autovar;        /* char stack variable */
                  register char regvar;     /* char register variable */

                  autovar = regvar = mstatic1;
                  autovar++;

                  for ( regvar = 0; regvar < (char) 5 ; regvar++ )
                          vchar += regvar*autovar;
          }
```

Go.Return

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

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



```
address info
attr:    FLE

function
\\diabc\diabc\func2a

   P:40000114--40000173      global static

function info
* size: 0. push: [] use: [R0,R1,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12]

epilog:       P:4000015C
exit:         P:40000170

module info
language: ELF-C
producer: Diab Data, Inc:dcc Rel 4.0b:PPC603
source:   I:\T32DEMO\POWERPC\55xx\code_0x40000020_data_0x40004000\diabc.c

type
(void ()) function (void)

(void) void
```

DATA Breakpoints

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

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

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

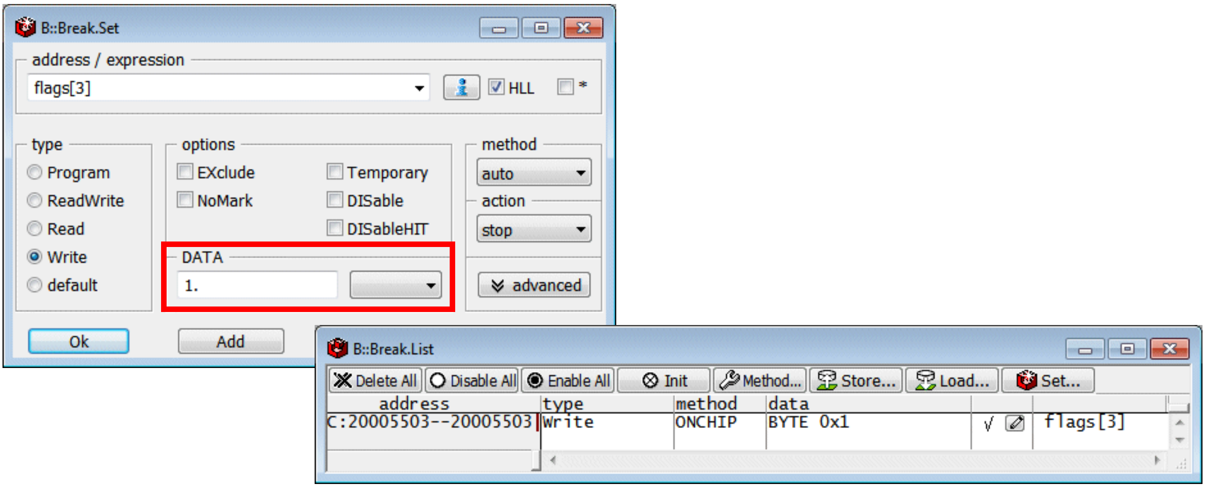
  TRACE32 PowerView allows inverted data values if this is supported by the on-chip break logic.

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

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

  TRACE32 PowerView allows inverted data values for intrusive DATA breakpoints.

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



B::Break.Set

address / expression: flags[3] ☑ HLL □ *

type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ Temporary □ NoMark □ DISable □ DISableHIT

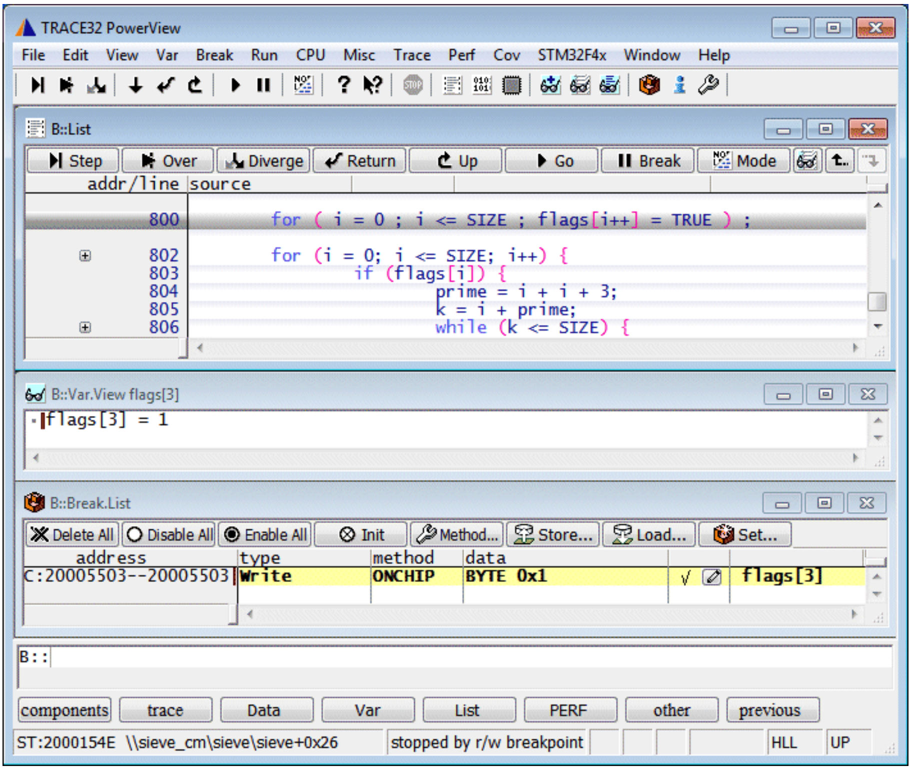
DATA: 1.

method: auto

action: stop

B::Break.List

address	type	method	data		
C:20005503--20005503	Write	ONCHIP	BYTE 0x1	√	flags[3]



TRACE32 PowerView

B::List

```
addr/line source
      800      for ( i = 0 ; i <= SIZE ; flags[i++] = TRUE ) ;

      802      for (i = 0; i <= SIZE; i++) {
      803          if (flags[i]) {
      804              prime = i + i + 3;
      805              k = i + prime;
      806              while (k <= SIZE) {
```

B::Var.View flags[3]

```
flags[3] = 1
```

B::Break.List

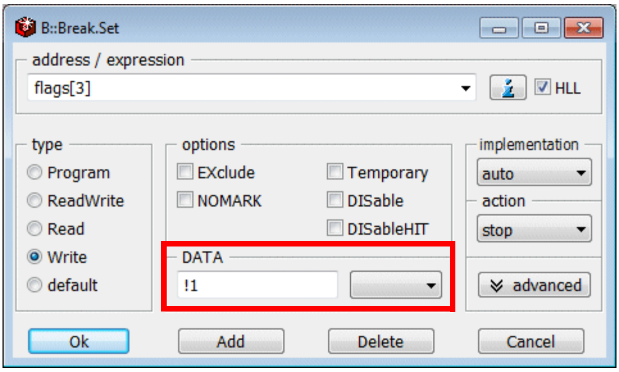
address	type	method	data		
C:20005503--20005503	Write	ONCHIP	BYTE 0x1	√	flags[3]

```
B::
```

ST:2000154E \\sieve_cm\sieve\sieve+0x26 | stopped by r/w breakpoint | HLL | UP

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

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



address / expression: flags[3] ☑ HLL

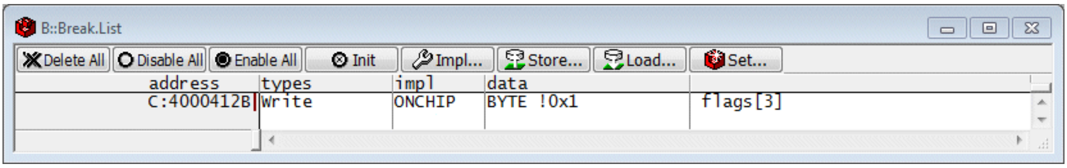
type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ Temporary □ NOMARK □ DISable □ DISableHIT

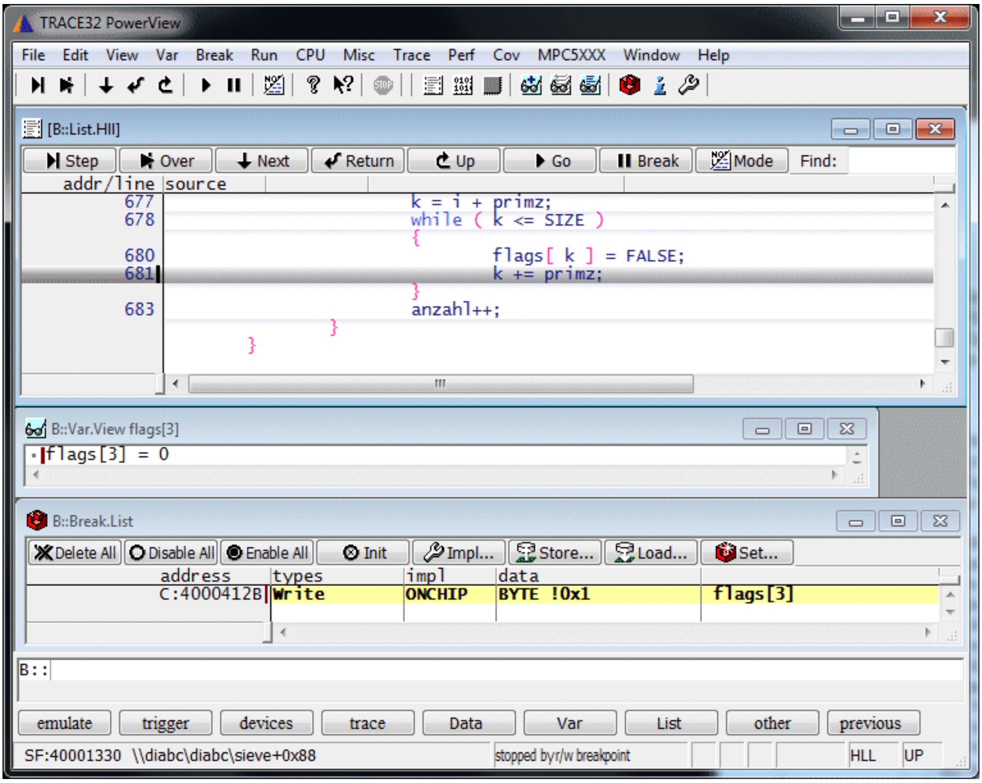
DATA: !1

implementation: auto

action: stop



address	types	impl	data	
C:4000412B	Write	ONCHIP	BYTE !0x1	flags[3]



```
addr/line source
  677                       k = i + primz;
  678                       while ( k <= SIZE )
                            {
  680                               flags[ k ] = FALSE;
  681                               k += primz;
                            }
  683                       anzahl++;
                        }
                    }
```

```
flags[3] = 0
```

address	types	impl	data	
C:4000412B	Write	ONCHIP	BYTE !0x1	flags[3]

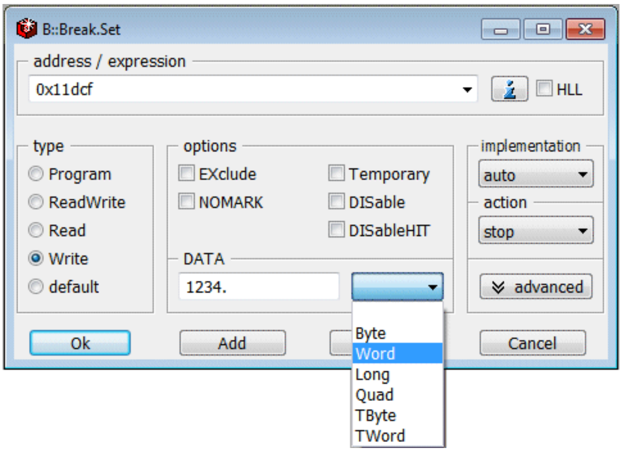
B::

SF:40001330 \\diabc\diabc\sieve+0x88 stopped by r/w breakpoint HLL UP

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

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

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



B::Break.Set

address / expression: 0x11dcf □ HLL

type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ Temporary □ NOMARK □ DISable □ DISableHIT

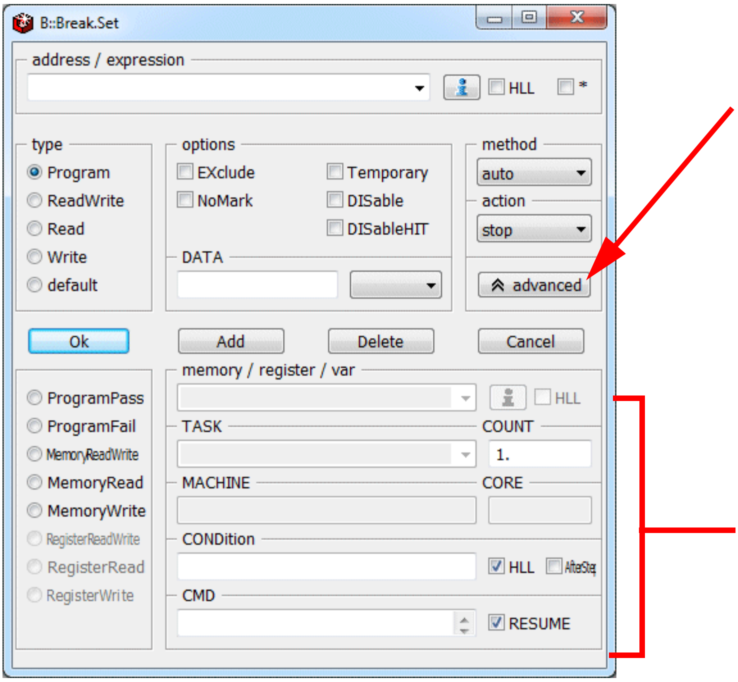
implementation: auto

action: stop

DATA: 1234.

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

Advanced Breakpoints



B::Break.Set

address / expression: □ HLL □ *

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

DATA

method: auto

action: stop

advanced

Ok | Add | Delete | Cancel

○ ProgramPass ○ ProgramFail ○ MemoryReadWrite ○ MemoryRead ○ MemoryWrite ○ RegisterReadWrite ○ RegisterRead ○ RegisterWrite

memory / register / var: □ HLL

TASK

COUNT: 1.

MACHINE

CORE

CONDition: ☑ HLL □ AfterStep

CMD: ☑ RESUME

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

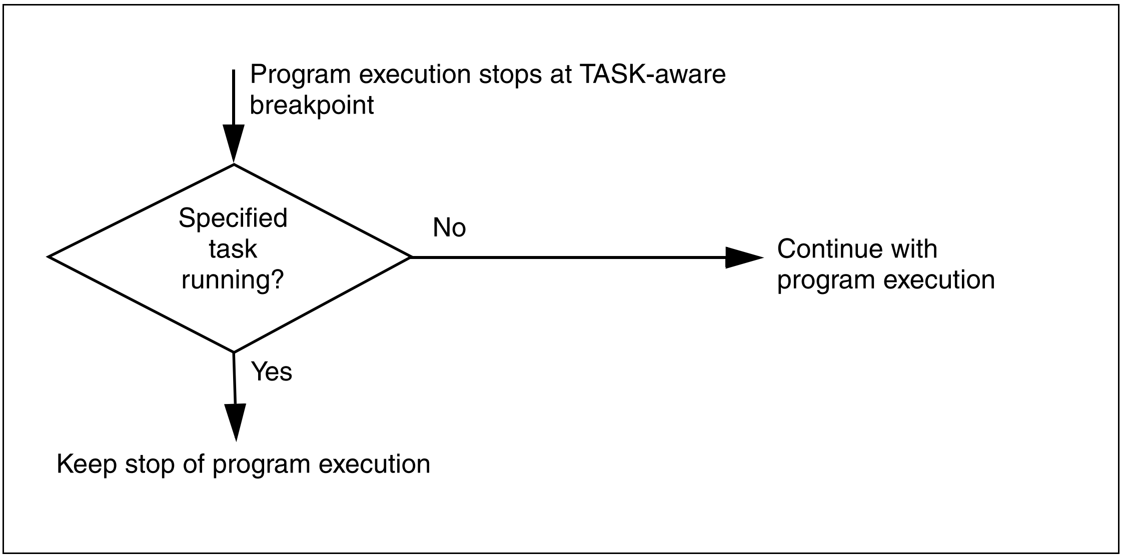
Advanced breakpoint input fields

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

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

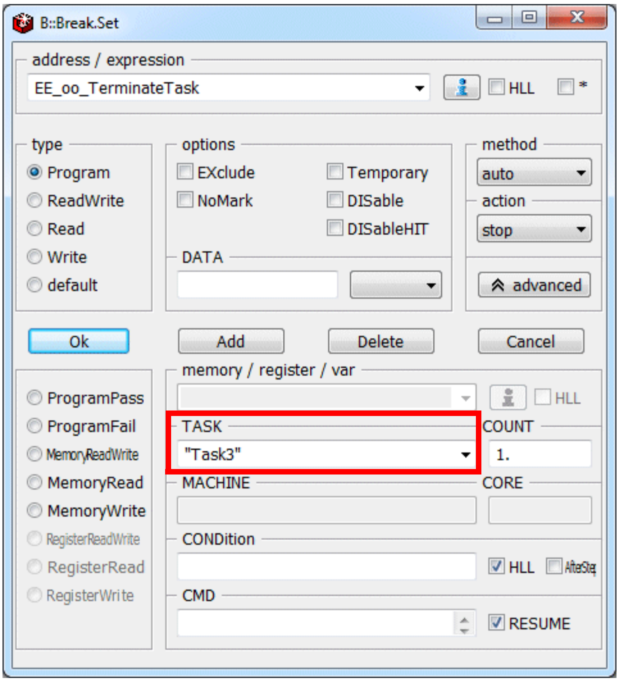
Intrusive TASK-aware Breakpoint

Processing:


Program execution stops at TASK-aware breakpoint
Specified task running?
No
Continue with program execution
Yes
Keep stop of program execution

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

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



address / expression: EE_oo_TerminateTask □ HLL □ *

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ Temporary □ NoMark □ DISable □ DISableHIT

method: auto

action: stop

DATA:

○ ProgramPass ○ ProgramFail ○ MemoryReadWrite ○ MemoryRead ○ MemoryWrite ○ RegisterReadWrite ○ RegisterRead ○ RegisterWrite

memory / register / var: □ HLL

TASK: "Task3"

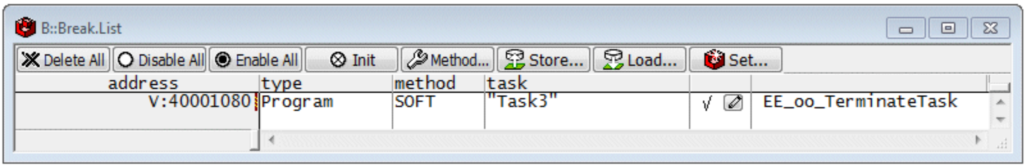
COUNT: 1.

MACHINE:

CORE:

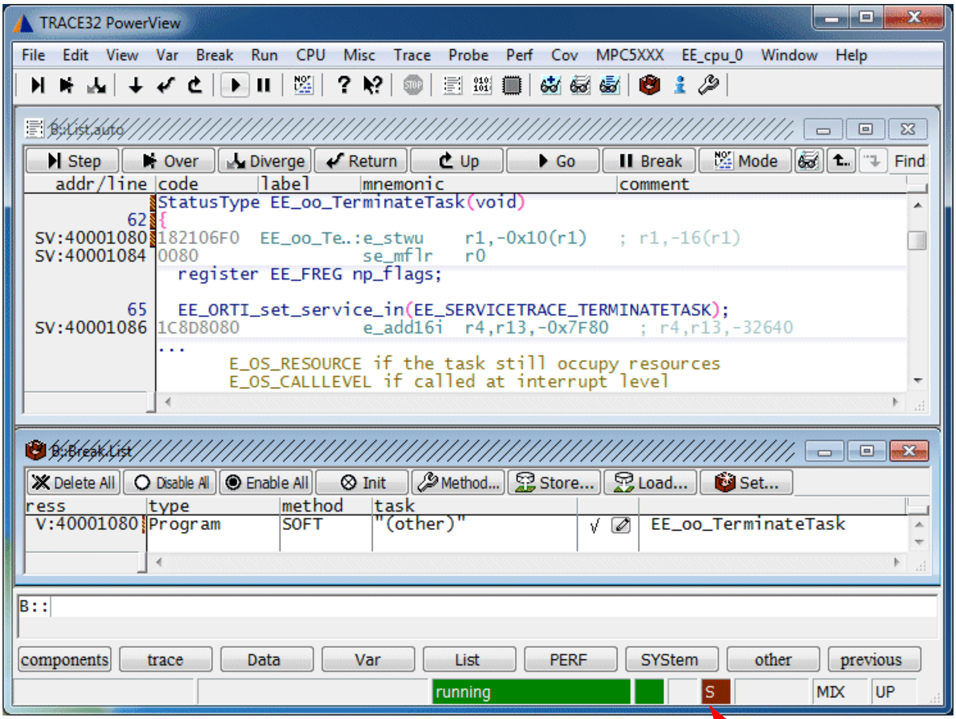
CONDition: ☑ HLL □ AfterStep

CMD: ☑ RESUME



```
         address  type     method  task
      V:40001080  Program  SOFT    "Task3"     √    EE_oo_TerminateTask
```

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

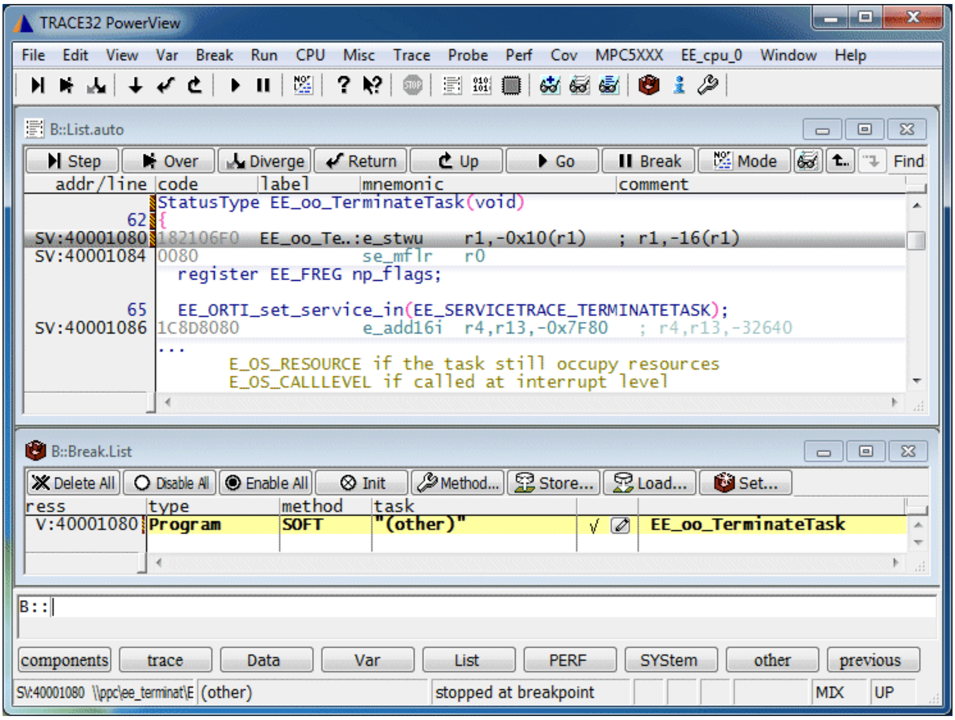

TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX EE_cpu_0 Window Help
B::List.auto
Step Over Diverge Return Up Go Break Mode Find
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
components trace Data Var List PERF SYStem other previous
running
S
MIX UP

```
  addr/line code     label         mnemonic                comment
                    StatusType EE_oo_TerminateTask(void)
          62        {
SV:40001080 182106F0 EE_oo_Te..:e_stwu    r1,-0x10(r1)     ; r1,-16(r1)
SV:40001084 0080                se_mflr   r0
                      register EE_FREG np_flags;

          65          EE_ORTI_set_service_in(EE_SERVICETRACE_TERMINATETASK);
SV:40001086 1C8D8080            e_add16i  r4,r13,-0x7F80    ; r4,r13,-32640
            ...
                           E_OS_RESOURCE if the task still occupy resources
                           E_OS_CALLLEVEL if called at interrupt level
```

```
ress         type      method  task
 V:40001080  Program   SOFT    "(other)"        √     EE_oo_TerminateTask
```

```
B::
```

The red S indicates that an intrusive breakpoint is used

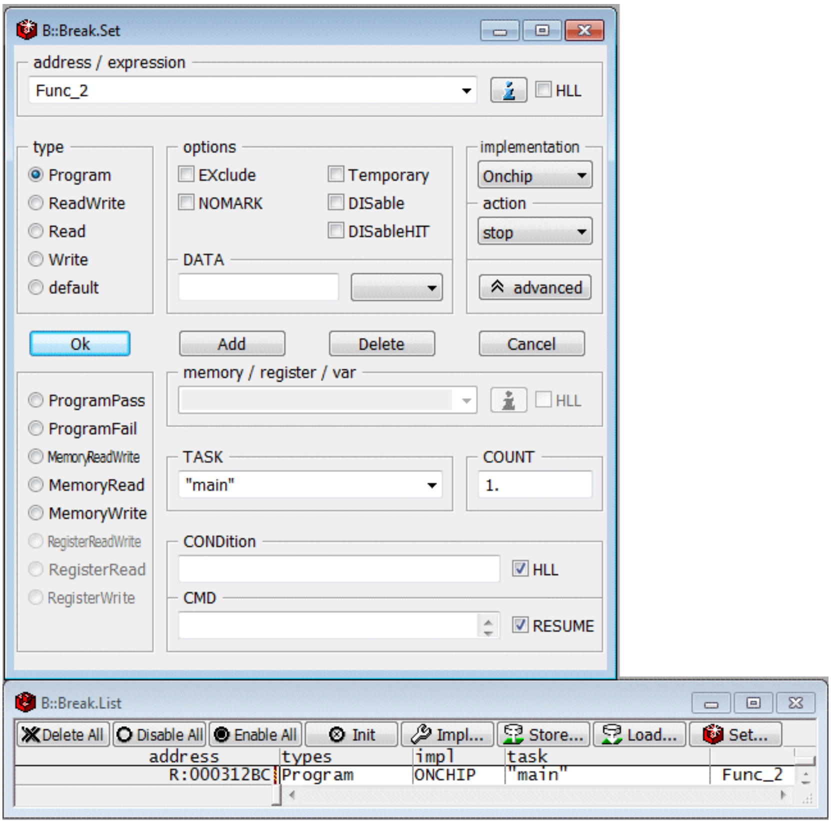

TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX EE_cpu_0 Window Help
B::List.auto
Step Over Diverge Return Up Go Break Mode Find
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
components trace Data Var List PERF SYStem other previous
MIX UP

```
  addr/line code     label         mnemonic                comment
                    StatusType EE_oo_TerminateTask(void)
          62        {
SV:40001080 182106F0 EE_oo_Te..:e_stwu    r1,-0x10(r1)     ; r1,-16(r1)
SV:40001084 0080                se_mflr   r0
                      register EE_FREG np_flags;

          65          EE_ORTI_set_service_in(EE_SERVICETRACE_TERMINATETASK);
SV:40001086 1C8D8080            e_add16i  r4,r13,-0x7F80    ; r4,r13,-32640
            ...
                           E_OS_RESOURCE if the task still occupy resources
                           E_OS_CALLLEVEL if called at interrupt level
```

```
ress         type      method  task
 V:40001080  Program   SOFT    "(other)"        √     EE_oo_TerminateTask
```

```
B::|
SV:40001080 \\ppc\ee_terminat\E (other)   stopped at breakpoint
```

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



B::Break.Set

address / expression: Func_2 □ HLL

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ Temporary □ NOMARK □ DISable □ DISableHIT

implementation: Onchip

action: stop

TASK: "main"

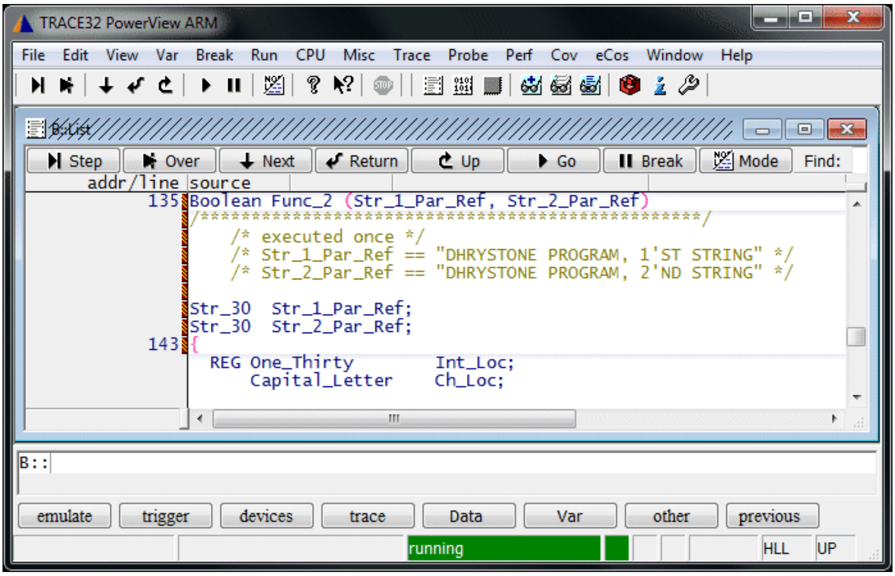
COUNT: 1.

CONDition: ☑ HLL

CMD: ☑ RESUME

B::Break.List

address	types	impl	task	
R:000312BC	Program	ONCHIP	"main"	Func_2



```
addr/line source
      135 Boolean Func_2 (Str_1_Par_Ref, Str_2_Par_Ref)
          /*************************************************/
              /* executed once */
              /* Str_1_Par_Ref == "DHRYSTONE PROGRAM, 1'ST STRING" */
              /* Str_2_Par_Ref == "DHRYSTONE PROGRAM, 2'ND STRING" */

          Str_30  Str_1_Par_Ref;
          Str_30  Str_2_Par_Ref;
      143 {
            REG One_Thirty        Int_Loc;
                Capital_Letter    Ch_Loc;
```

B::

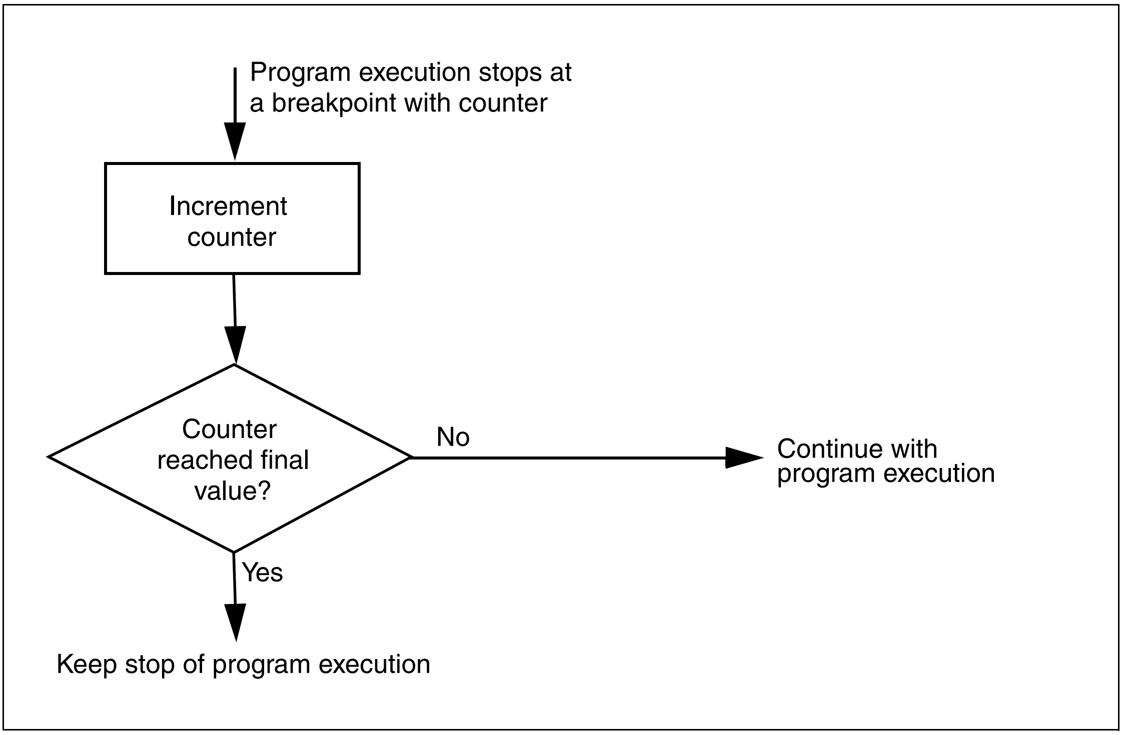
running HLL UP

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

Software Counter

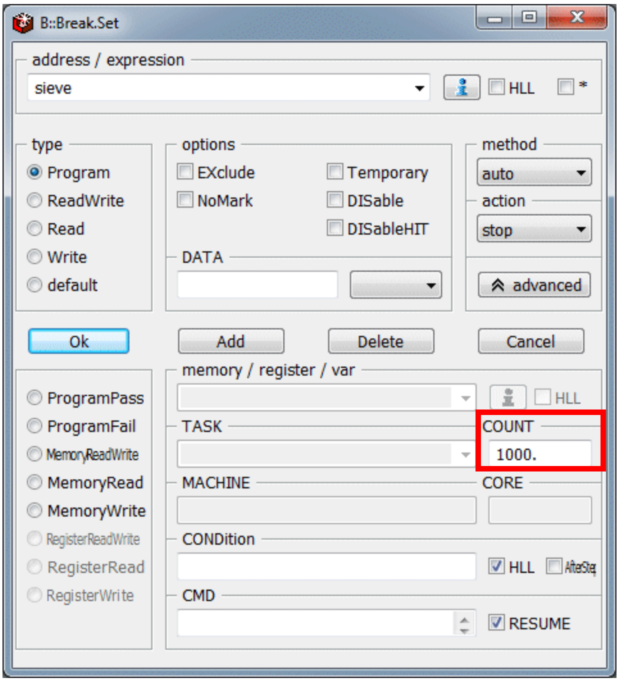
If the on-chip break logic of the core does not provide counters or if a Software breakpoint is used, counters are implemented as software counters.

Processing:


Program execution stops at a breakpoint with counter
Increment counter
Counter reached final value?
No
Continue with program execution
Yes
Keep stop of program execution

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

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



address / expression: sieve □ HLL □ *

type: ● Program

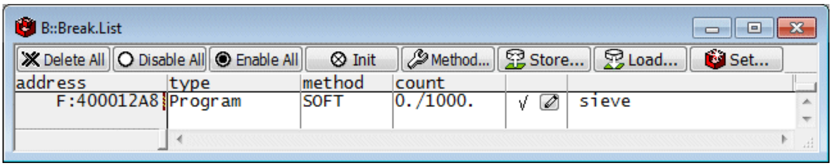
method: auto

action: stop

COUNT: 1000.

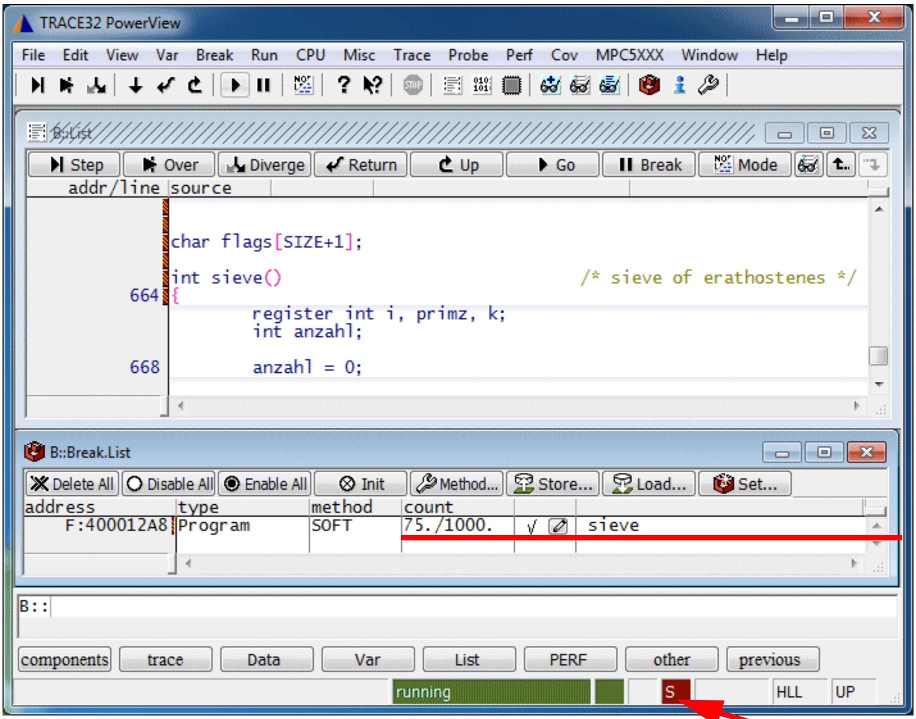
☑ HLL

☑ RESUME



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

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

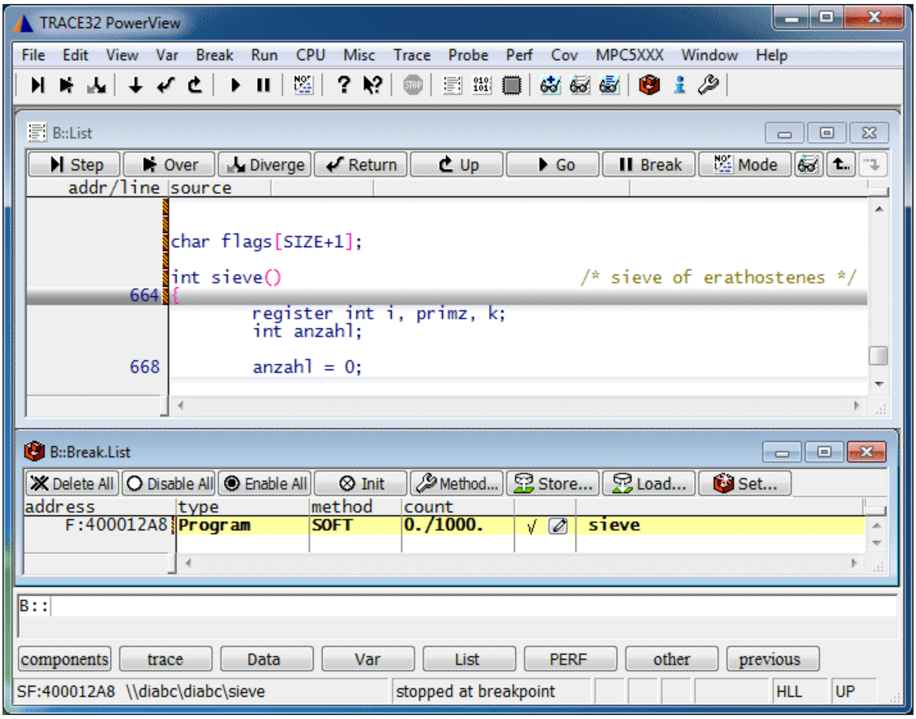

TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
B::
components trace Data Var List PERF other previous
running S HLL UP

```
      addr/line  source

                 char flags[SIZE+1];

                 int sieve()                              /* sieve of erathostenes */
            664  {
                         register int i, primz, k;
                         int anzahl;

            668          anzahl = 0;
```

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

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

The red S indicates an intrusive breakpoint


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
B::
components trace Data Var List PERF other previous
HLL UP

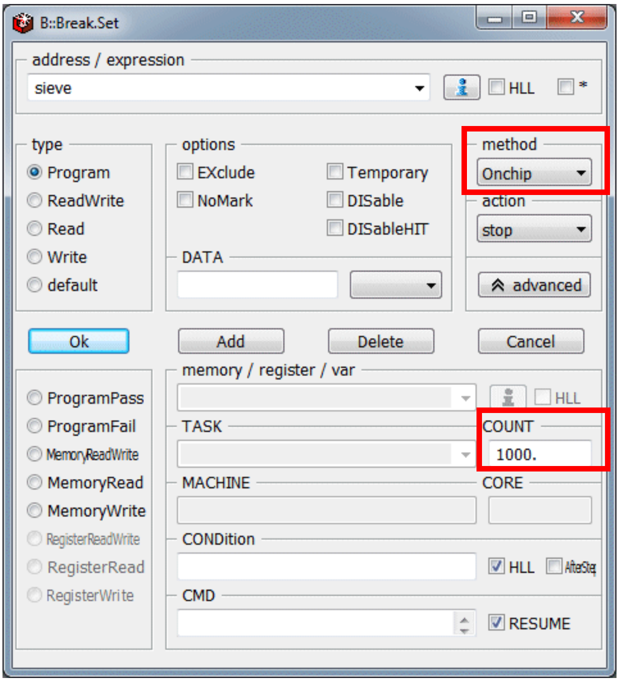
```
      addr/line  source

                 char flags[SIZE+1];

                 int sieve()                              /* sieve of erathostenes */
            664  {
                         register int i, primz, k;
                         int anzahl;

            668          anzahl = 0;
```

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

SF:400012A8 \\diabc\diabc\sieve   stopped at breakpoint

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

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



address / expression: sieve

type: ● Program

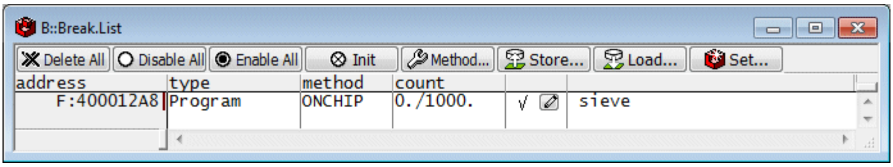
method: Onchip

action: stop

COUNT: 1000.

CONDition: ☑ HLL

☑ RESUME



address	type	method	count		
F:400012A8	Program	ONCHIP	0./1000.	√	sieve

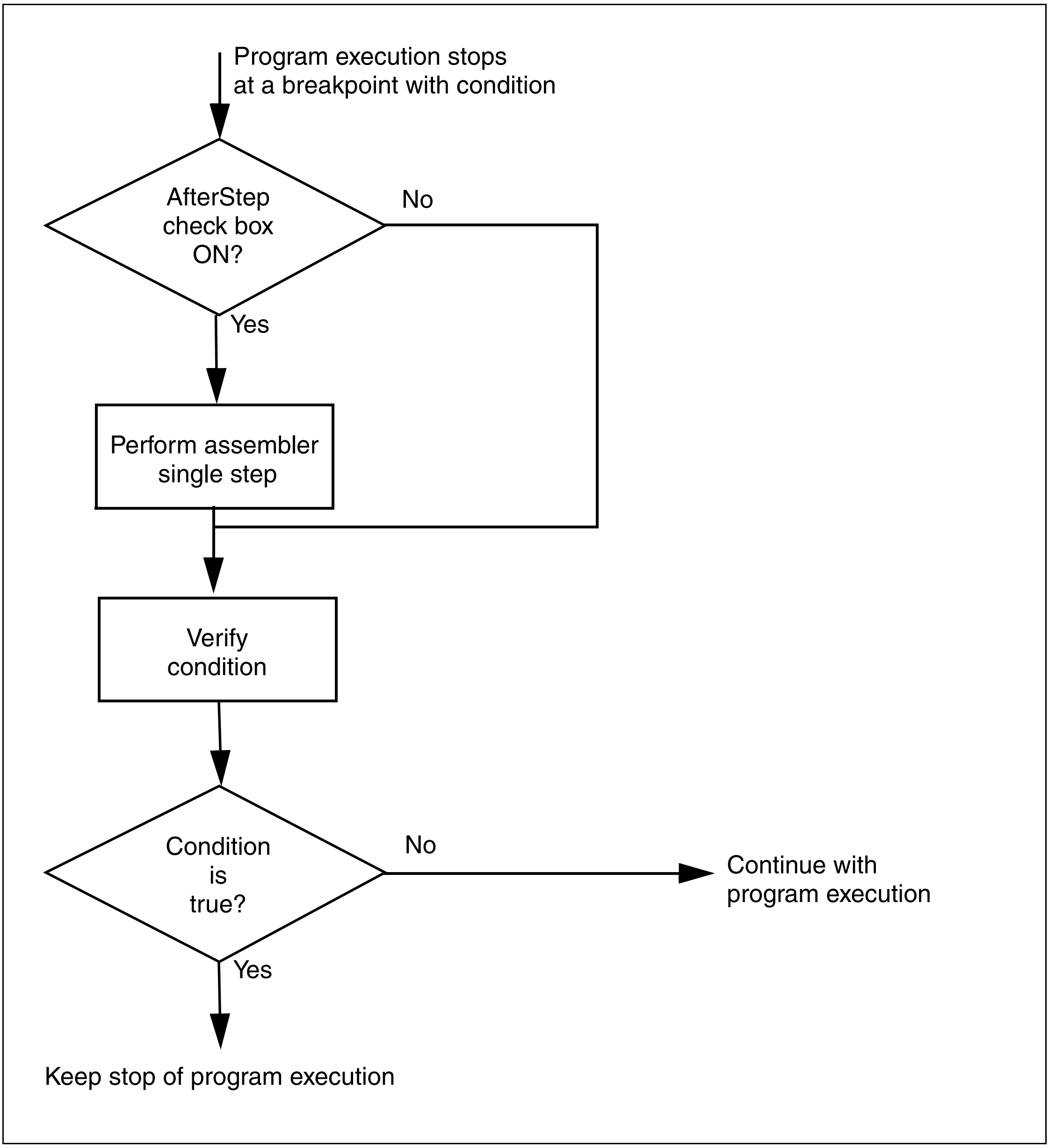
```
Break.Set sieve /COUNT 1000. /Onchip
```

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

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

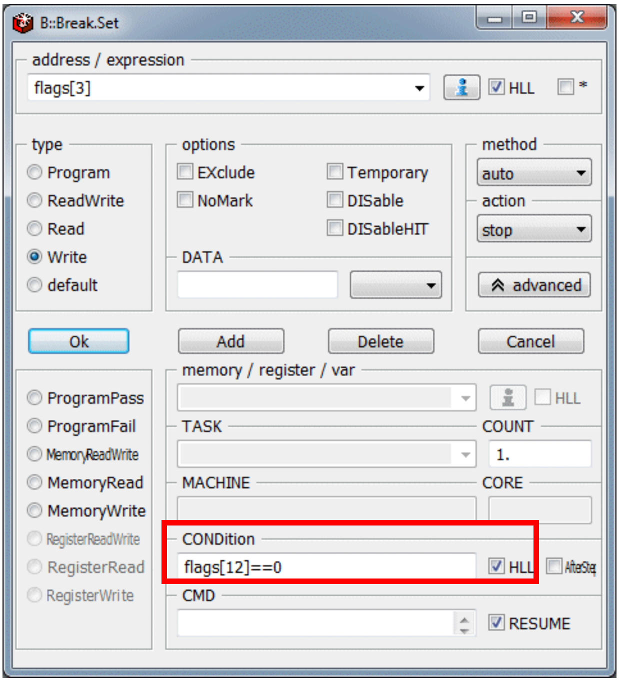
CONDition breakpoints are always intrusive.

Processing:



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

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



B::Break.Set

address / expression: flags[3] ☑ HLL □ *

type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ Temporary □ NoMark □ DISable □ DISableHIT

DATA:

method: auto

action: stop

memory / register / var: □ HLL

○ ProgramPass ○ ProgramFail ○ MemoryReadWrite ○ MemoryRead ○ MemoryWrite ○ RegisterReadWrite ○ RegisterRead ○ RegisterWrite

TASK:

COUNT: 1.

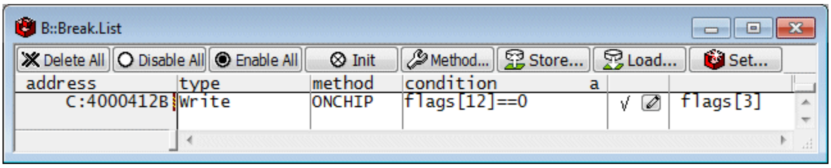
MACHINE:

CORE:

CONDition: flags[12]==0 ☑ HLL □ AfterStep

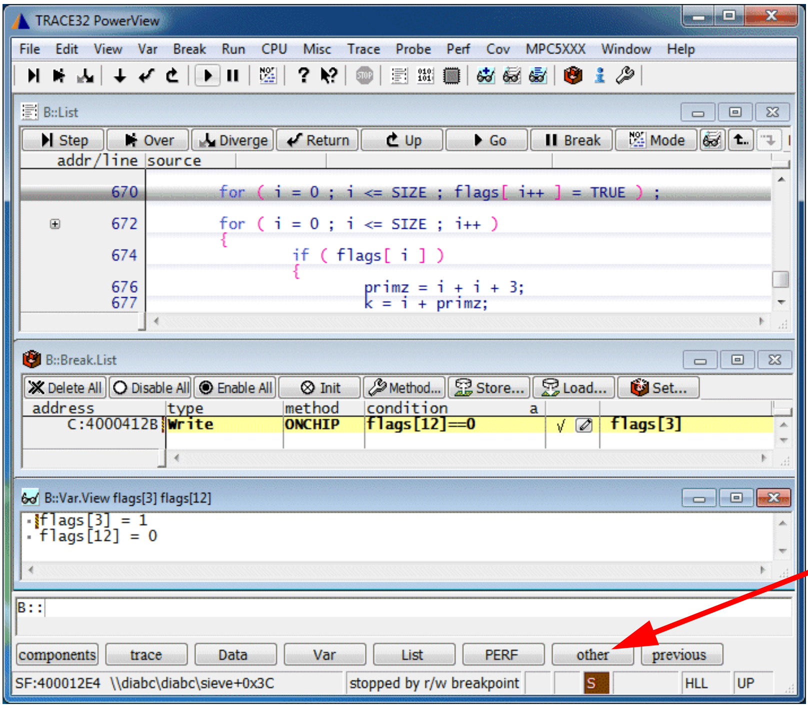
CMD:

☑ RESUME



B::Break.List

address	type	method	condition	a		
C:4000412B	Write	ONCHIP	flags[12]==0	√		flags[3]


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
B::Var.View flags[3] flags[12]
components trace Data Var List PERF other previous

```
addr/line source
      670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
      672         for ( i = 0 ; i <= SIZE ; i++ )
                  {
      674             if ( flags[ i ] )
                      {
      676                 primz = i + i + 3;
      677                 k = i + primz;
```

address	type	method	condition	a		
C:4000412B	Write	ONCHIP	flags[12]==0	√		flags[3]

```
flags[3] = 1
flags[12] = 0
```

```
B::
```

```
SF:400012E4  \\diabc\diabc\sieve+0x3C    stopped by r/w breakpoint    S    HLL  UP
```

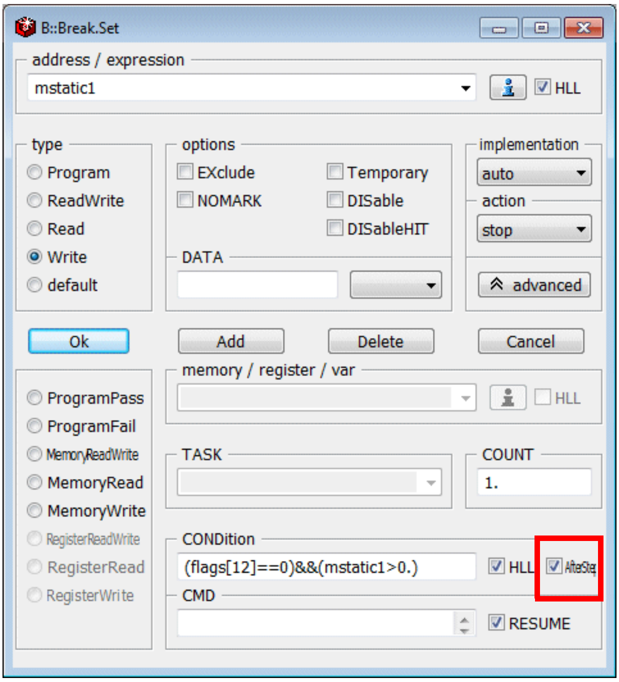
The red S indicates
an intrusive breakpoint

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

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

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

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



address / expression: mstatic1 ☑ HLL

type: ● Write

implementation: auto

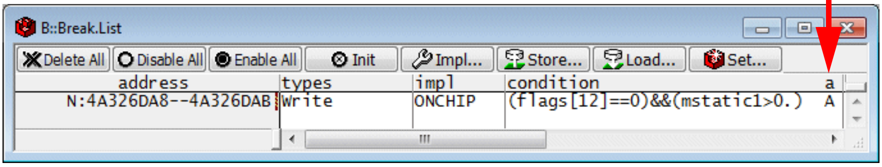
action: stop

COUNT: 1.

CONDition: (flags[12]==0)&&(mstatic1>0.) ☑ HLL ☑ AfterStep

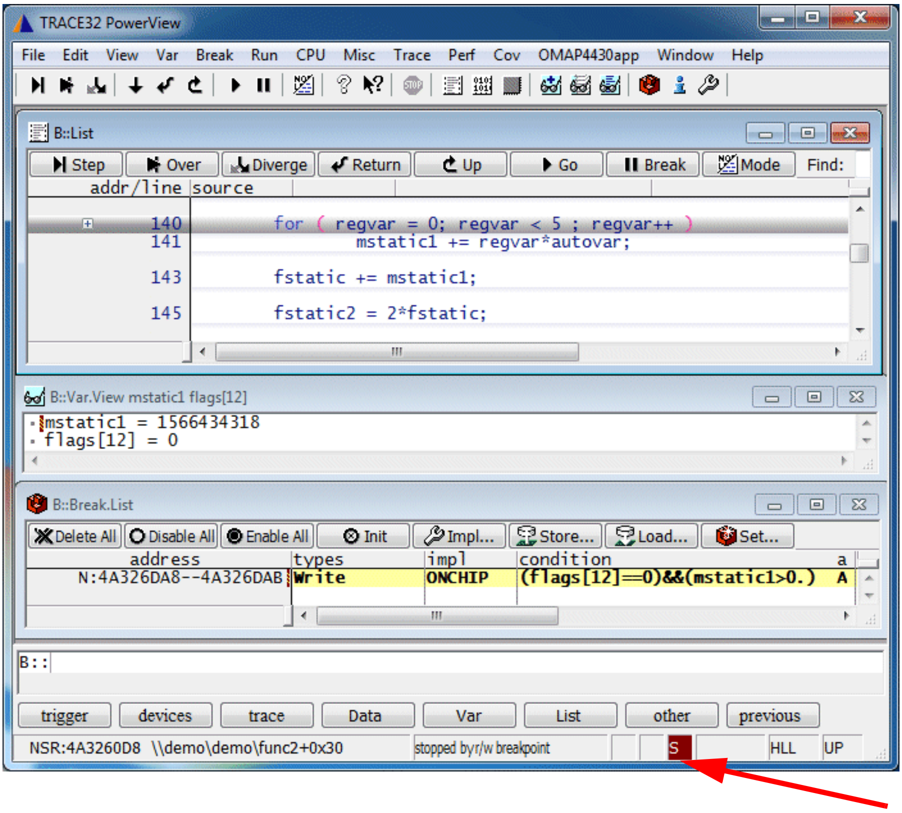
☑ RESUME

AfterStep checked



address	types	impl	condition	a
N:4A326DA8--4A326DAB	Write	ONCHIP	(flags[12]==0)&&(mstatic1>0.)	A

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


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov OMAP4430app Window Help
B::List
Step Over Diverge Return Up Go Break Mode Find:
B::Var.View mstatic1 flags[12]
B::Break.List
Delete All Disable All Enable All Init Impl... Store... Load... Set...
trigger devices trace Data Var List other previous

```
addr/line  source
   140         for ( regvar = 0; regvar < 5 ; regvar++ )
   141                 mstatic1 += regvar*autovar;

   143         fstatic += mstatic1;

   145         fstatic2 = 2*fstatic;
```

```
mstatic1 = 1566434318
flags[12] = 0
```

address	types	impl	condition	a
N:4A326DA8--4A326DAB	Write	ONCHIP	(flags[12]==0)&&(mstatic1>0.)	A

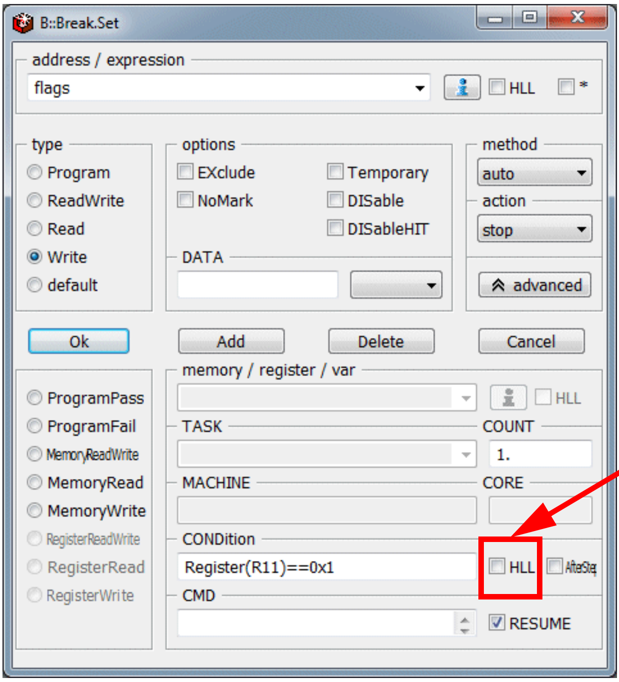
```
B::
```

NSR:4A3260D8 \\demo\demo\func2+0x30 | stopped by r/w breakpoint | S | HLL | UP

The red S indicates an intrusive breakpoint

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

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



B::Break.Set

address / expression: flags ☐ HLL ☐ *

type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: ☐ EXclude ☐ Temporary ☐ NoMark ☐ DISable ☐ DISableHIT

method: auto

action: stop

COUNT: 1.

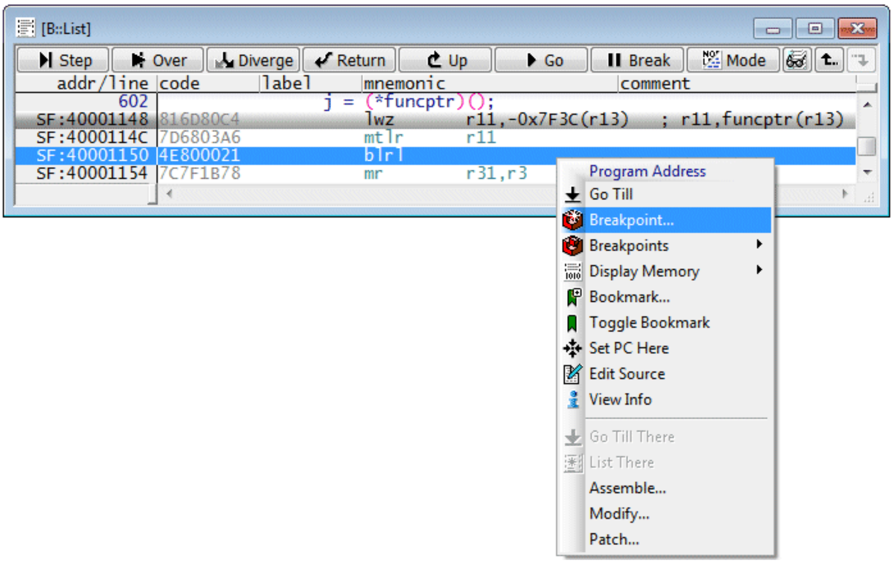
CONDition: Register(R11)==0x1 ☐ HLL ☐ AfterStep

☑ RESUME

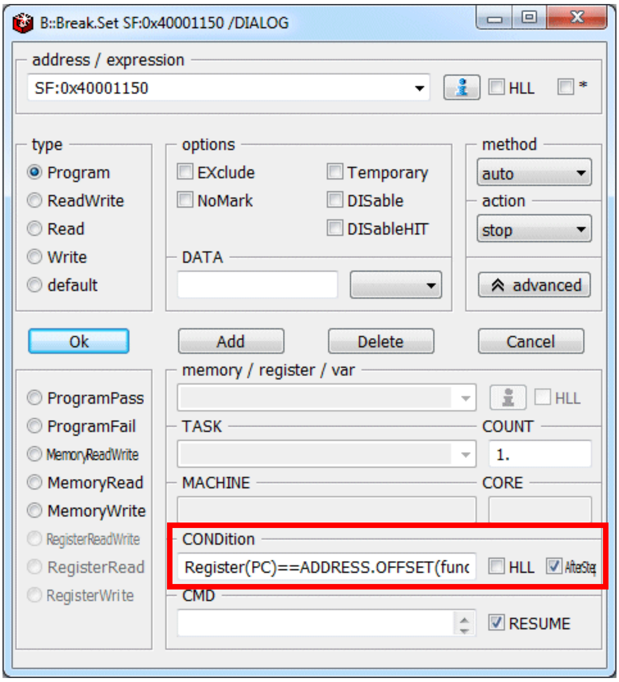
Switch HLL OFF ->
TRACE32 syntax can be used
to specify the condition

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

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

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



```
addr/line  code      label  mnemonic              comment
      602                   j = (*funcptr)();
SF:40001148 816D80C4        lwz     r11,-0x7F3C(r13)   ; r11,funcptr(r13)
SF:4000114C 7D6803A6        mtlr    r11
SF:40001150 4E800021        blrl
SF:40001154 7C7F1B78        mr      r31,r3
```



address / expression: SF:0x40001150 □ HLL □ *

type: ● Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

DATA:

method: auto

action: stop

○ ProgramPass ○ ProgramFail ○ MemoryReadWrite ○ MemoryRead ○ MemoryWrite ○ RegisterReadWrite ○ RegisterRead ○ RegisterWrite

memory / register / var: □ HLL

TASK:

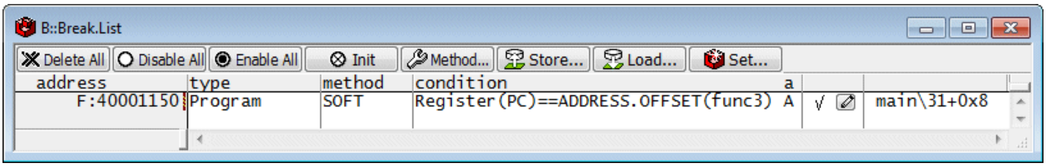
COUNT: 1.

MACHINE:

CORE:

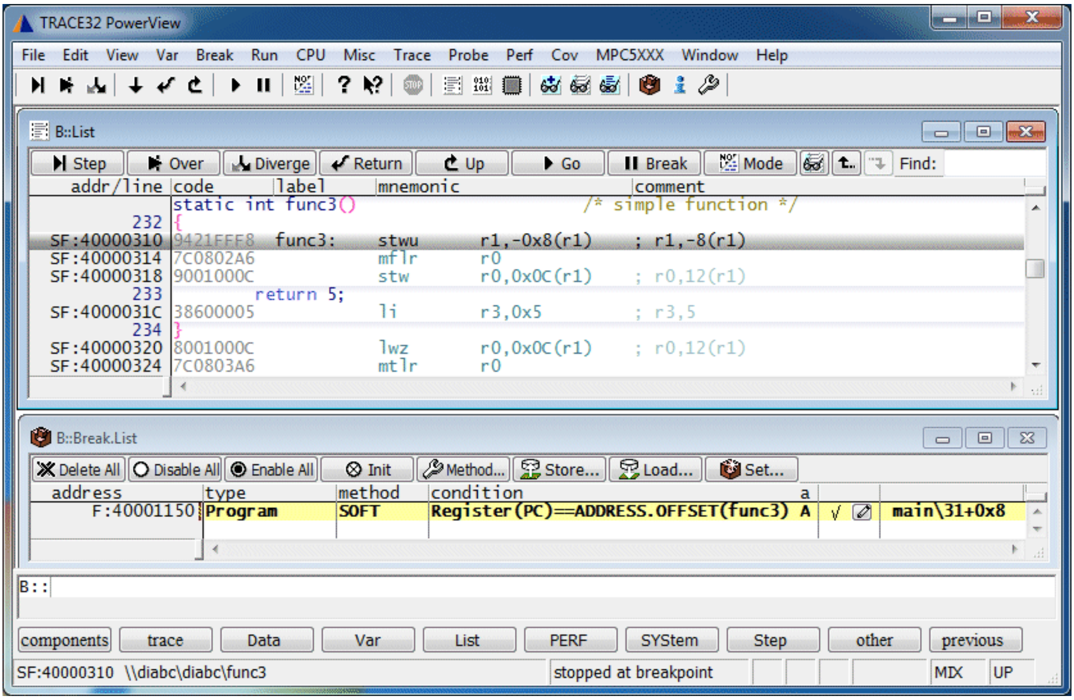
CONDition: Register(PC)==ADDRESS.OFFSET(func □ HLL ☑ AfterStep

CMD: ☑ RESUME



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

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



```
   addr/line code      label    mnemonic               comment
             static int func3()                        /* simple function */
         232 {
 SF:40000310 9421FFF8  func3:   stwu      r1,-0x8(r1)    ; r1,-8(r1)
 SF:40000314 7C0802A6           mflr      r0
 SF:40000318 9001000C           stw       r0,0x0C(r1)    ; r0,12(r1)
         233         return 5;
 SF:4000031C 38600005           li        r3,0x5         ; r3,5
         234 }
 SF:40000320 8001000C           lwz       r0,0x0C(r1)    ; r0,12(r1)
 SF:40000324 7C0803A6           mtlr      r0
```

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

```
B::
```

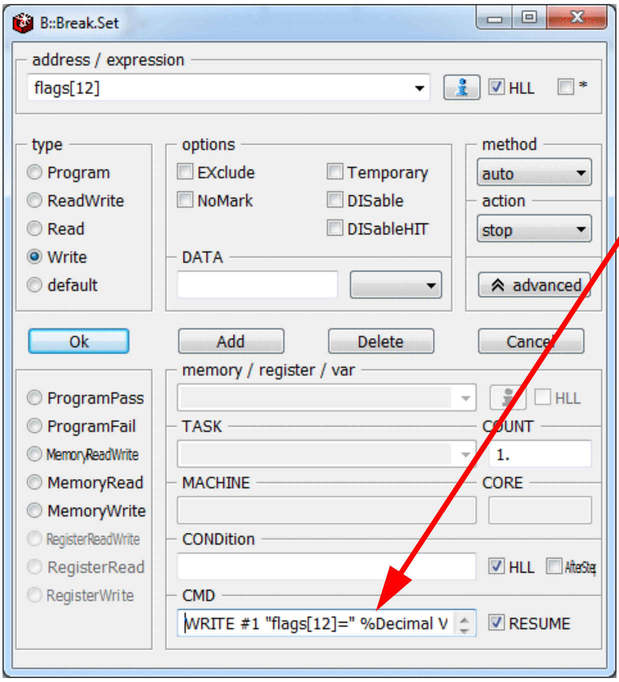
SF:40000310 \\diabc\diabc\func3 stopped at breakpoint MIX UP

CMD

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

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

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



B::Break.Set

address / expression: flags[12] ☑ HLL □ *

type: ○ Program ○ ReadWrite ○ Read ● Write ○ default

options: □ EXclude □ NoMark □ Temporary □ DISable □ DISableHIT

method: auto

action: stop

DATA:

memory / register / var: □ HLL

TASK:

COUNT: 1.

MACHINE:

CORE:

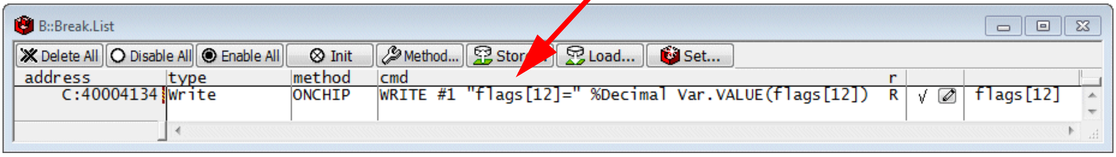
CONDition: ☑ HLL □ AfterStep

CMD: WRITE #1 "flags[12]=" %Decimal V

☑ RESUME

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.



B::Break.List

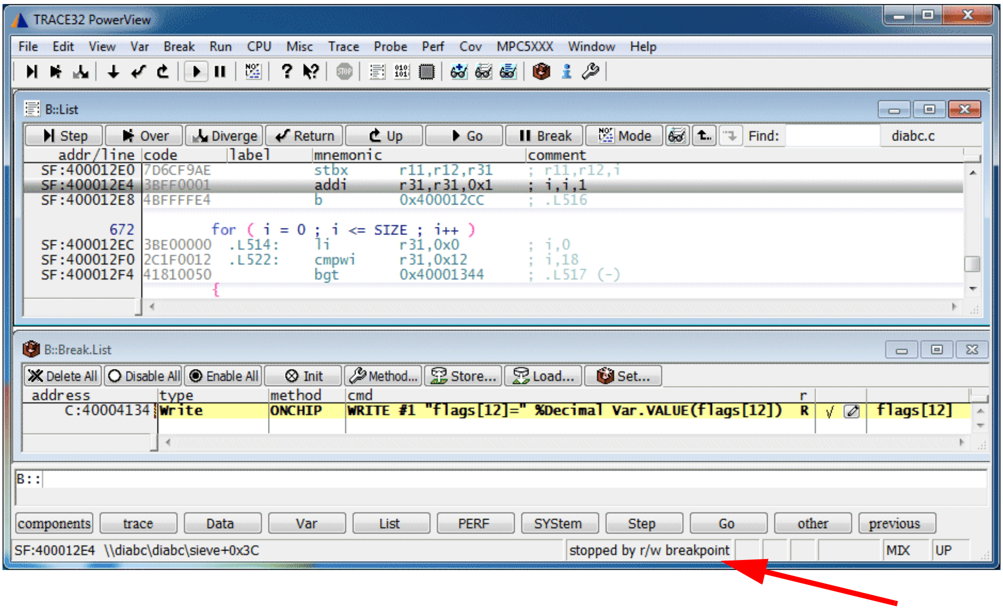
address	type	method	cmd	r		
C:40004134	Write	ONCHIP	WRITE #1 "flags[12]=" %Decimal Var.VALUE(flags[12])	R	√	flags[12]

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

It is recommended to set RESUME to OFF, if CMD

- starts a PRACTICE script with the command DO
- commands are used that open processing windows like Trace.STATistic.Func, Trace.Chart.sYmbol or CTS.List

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


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode Find: diabc.c
B::Break.List
Delete All Disable All Enable All Init Method... Store... Load... Set...
components trace Data Var List PERF SYStem Step Go other previous

```
   addr/line code      label    mnemonic              comment
SF:400012E0 7D6CF9AE          stbx     r11,r12,r31  ; r11,r12,i
SF:400012E4 3BFF0001          addi     r31,r31,0x1  ; i,i,1
SF:400012E8 4BFFFFE4          b        0x400012CC   ; .L516

        672         for ( i = 0 ; i <= SIZE ; i++ )
SF:400012EC 3BE00000  .L514:  li       r31,0x0      ; i,0
SF:400012F0 2C1F0012  .L522:  cmpwi    r31,0x12     ; i,18
SF:400012F4 41810050          bgt      0x40001344   ; .L517 (-)
                    {
```

address	type	method	cmd	r		
C:40004134	Write	ONCHIP	WRITE #1 "flags[12]=" %Decimal Var.VALUE(flags[12])	R	√	flags[12]

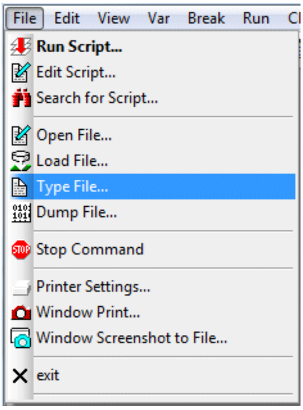
```
B::
```

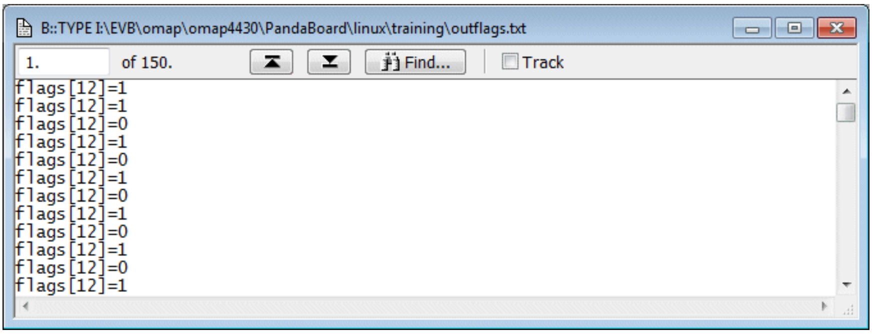
SF:400012E4 \\diabc\diabc\sieve+0x3C | stopped by r/w breakpoint | MIX | UP

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

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

Display the result:




B::TYPE I:\EVB\omap\omap4430\PandaBoard\linux\training\outflags.txt
1.
of 150.
Find...
□ Track

```
flags[12]=1
flags[12]=1
flags[12]=0
flags[12]=1
flags[12]=0
flags[12]=1
flags[12]=0
flags[12]=1
flags[12]=0
flags[12]=1
flags[12]=0
flags[12]=1
```

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

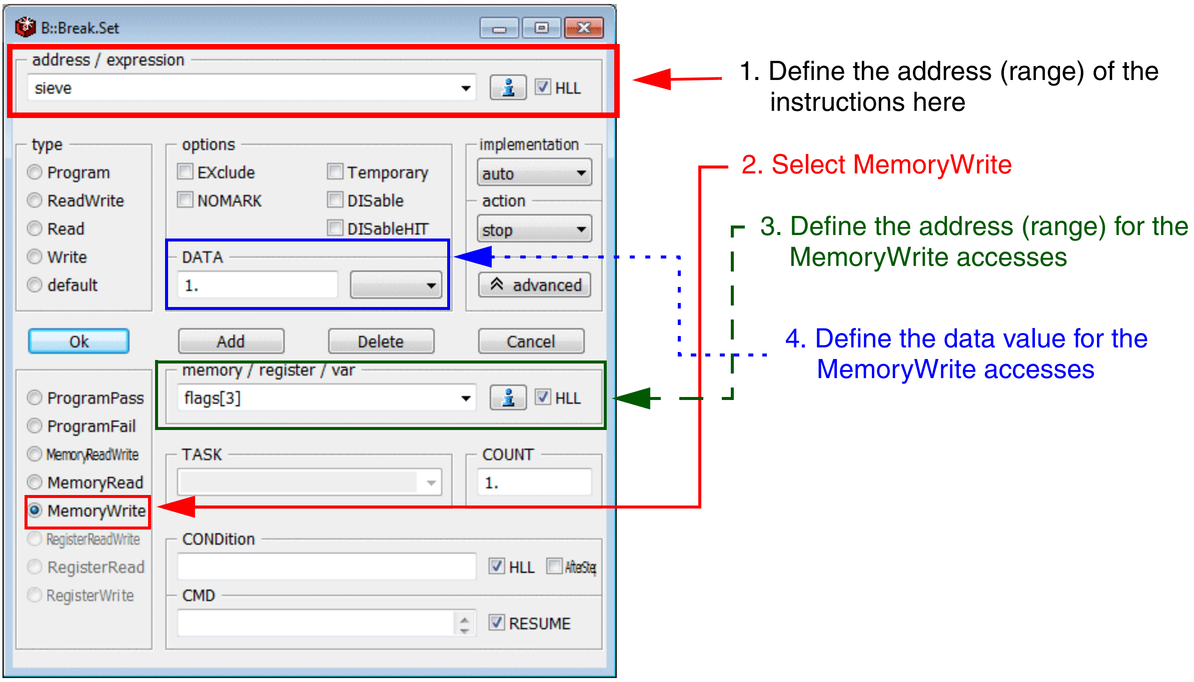
Preconditions

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

Advantageous

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

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



B::Break.Set

address / expression: sieve ☑ HLL

type: ○ Program ○ ReadWrite ○ Read ○ Write ○ default

options: □ EXclude □ Temporary □ NOMARK □ DISable □ DISableHIT

implementation: auto

action: stop

DATA: 1.

○ ProgramPass ○ ProgramFail ○ MemoryReadWrite ○ MemoryRead ● MemoryWrite ○ RegisterReadWrite ○ RegisterRead ○ RegisterWrite

memory / register / var: flags[3] ☑ HLL

TASK:

COUNT: 1.

CONDition: ☑ HLL □ AfterStep

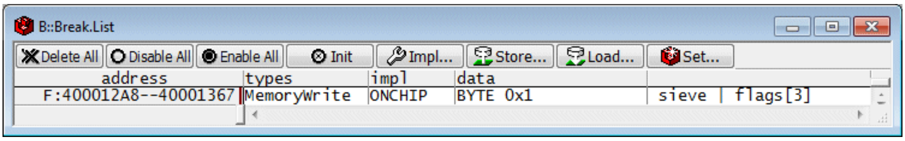
CMD: ☑ RESUME

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

2. Select MemoryWrite

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

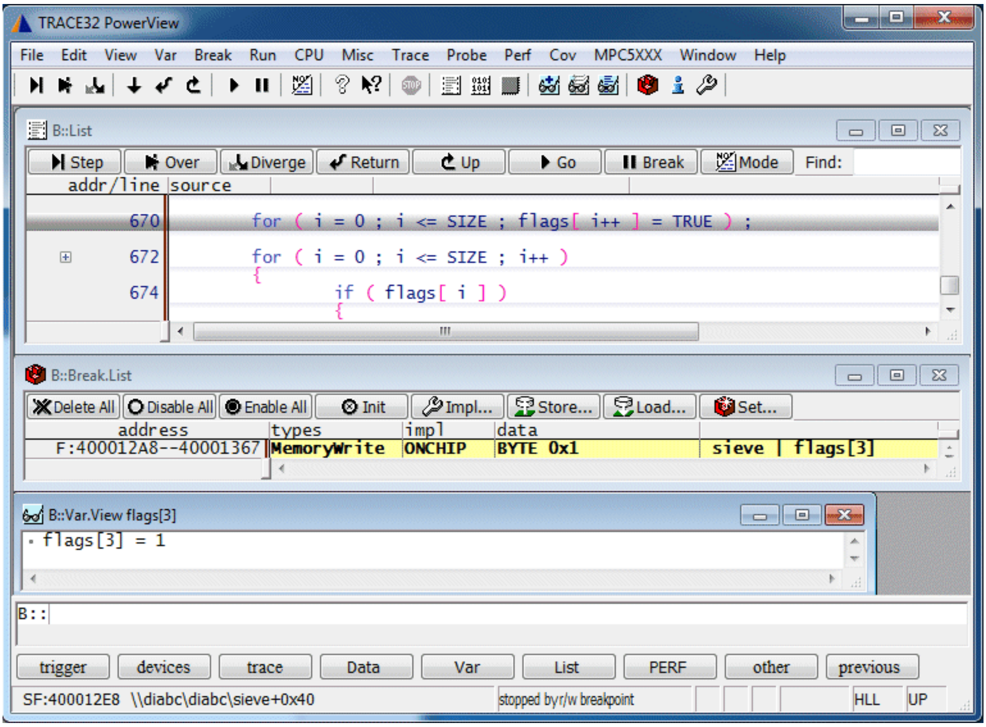
4. Define the data value for the MemoryWrite accesses



B::Break.List

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

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



TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Probe Perf Cov MPC5XXX Window Help
B::List
Step Over Diverge Return Up Go Break Mode Find:
B::Break.List
Delete All Disable All Enable All Init Impl... Store... Load... Set...
B::Var.View flags[3]
trigger devices trace Data Var List PERF other previous
HLL UP

```
addr/line source
      670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

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

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

```
flags[3] = 1
```

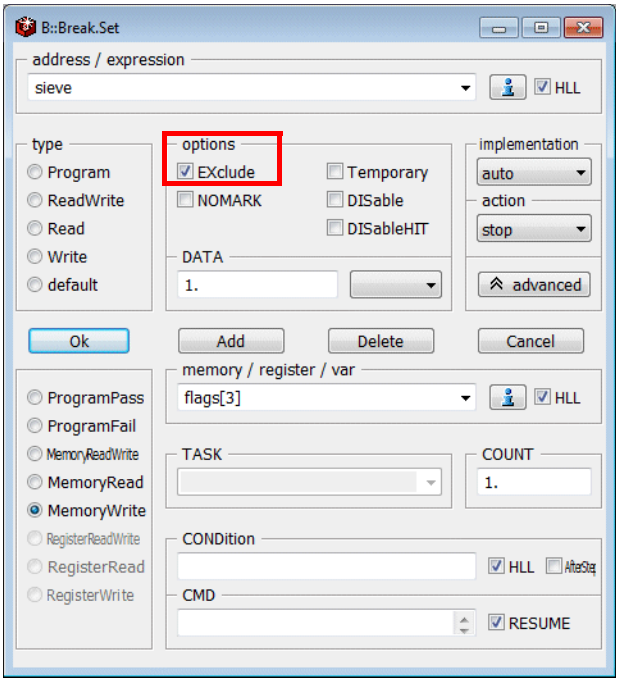
```
B::
```

SF:400012E8 \\diabc\diabc\sieve+0x40 stopped by r/w breakpoint

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

Example for the Option EXclude

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



B::Break.Set

address / expression: sieve ☑ HLL

type: ○ Program ○ ReadWrite ○ Read ○ Write ○ default

options: ☑ EXclude □ NOMARK □ Temporary □ DISable □ DISableHIT

DATA: 1.

implementation: auto

action: stop

○ ProgramPass ○ ProgramFail ○ MemoryReadWrite ○ MemoryRead ● MemoryWrite ○ RegisterReadWrite ○ RegisterRead ○ RegisterWrite

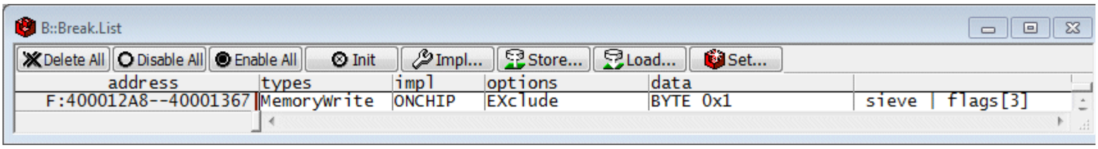
memory / register / var: flags[3] ☑ HLL

TASK:

COUNT: 1.

CONDition: ☑ HLL □ AfterStep

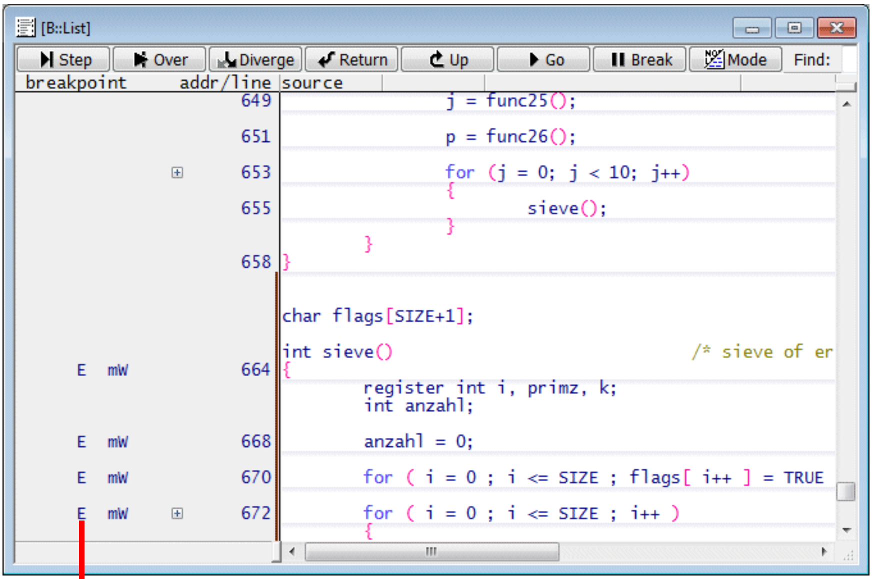
CMD: ☑ RESUME



B::Break.List

```
           address         types          impl      options         data
 F:400012A8--40001367 MemoryWrite    ONCHIP    EXclude         BYTE 0x1               sieve | flags[3]
```

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



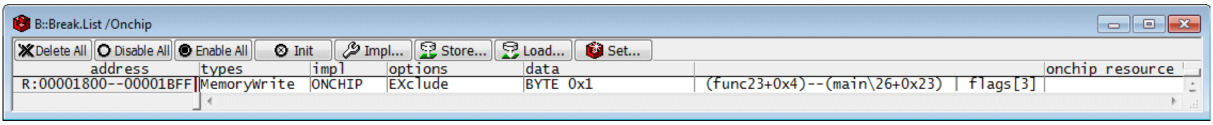
```
breakpoint       addr/line source
                   649                j = func25();

                   651                p = func26();

                   653                for (j = 0; j < 10; j++)
                                      {
                   655                        sieve();
                                      }
                                }
                   658 }


                       char flags[SIZE+1];

                       int sieve()                         /* sieve of er
    E   mW         664 {
                               register int i, primz, k;
                               int anzahl;

    E   mW         668         anzahl = 0;

    E   mW         670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE

    E   mW         672         for ( i = 0 ; i <= SIZE ; i++ )
                               {
```

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

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

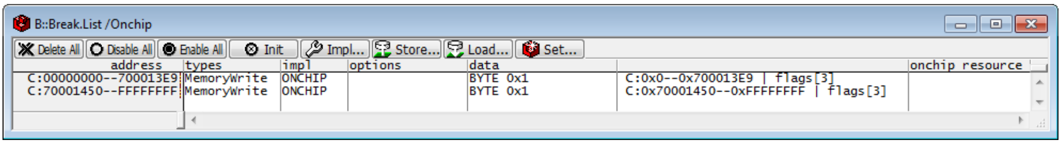
```
Break.List /Onchip
```

Inverting logic of on-chip break logic:



```
           address  types        impl    options    data                                         onchip resource
R:00001800--00001BFF MemoryWrite ONCHIP  EXclude    BYTE 0x1   (func23+0x4)--(main\26+0x23) | flags[3]
```

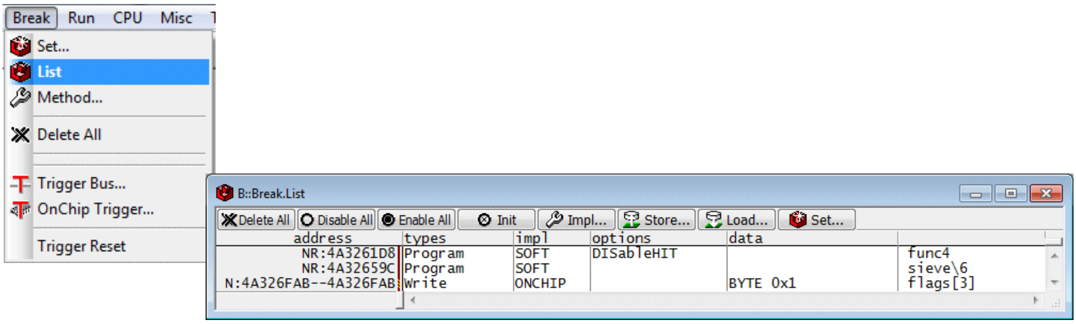
Two address range breakpoints:



```
           address  types        impl    options    data                                     onchip resource
 C:00000000--700013E9 MemoryWrite ONCHIP             BYTE 0x1   C:0x0--0x700013E9 | flags[3]
 C:70001450--FFFFFFFF MemoryWrite ONCHIP             BYTE 0x1   C:0x70001450--0xFFFFFFFF | flags[3]
```

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

Display a List of all Set Breakpoints

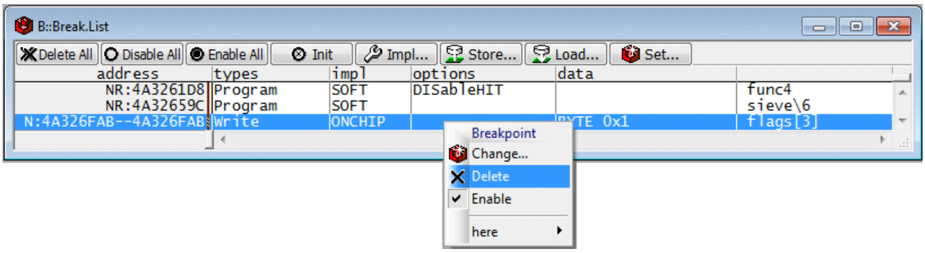


```
          address  types     impl     options       data
      NR:4A3261D8  Program   SOFT     DISableHIT                        func4
      NR:4A32659C  Program   SOFT                                       sieve\6
N:4A326FAB--4A326FAB Write   ONCHIP                 BYTE 0x1            flags[3]
```

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

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

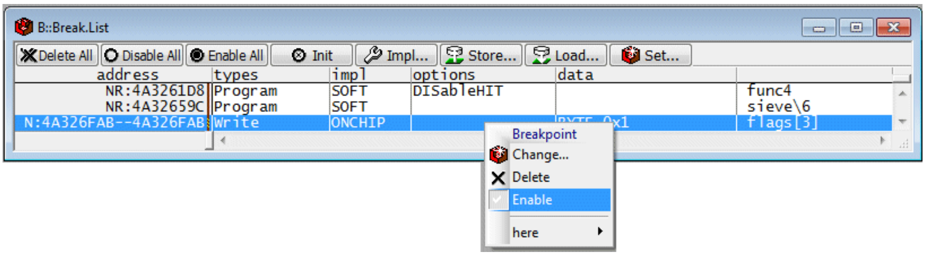
Delete Breakpoints



address	types	impl	options	data	
NR:4A3261D8	Program	SOFT	DISableHIT		func4
NR:4A32659C	Program	SOFT			sieve\6
N:4A326FAB--4A326FAB	Write	ONCHIP		BYTE 0x1	flags[3]

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

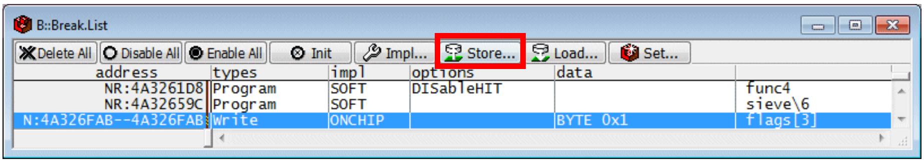
Enable/Disable Breakpoints



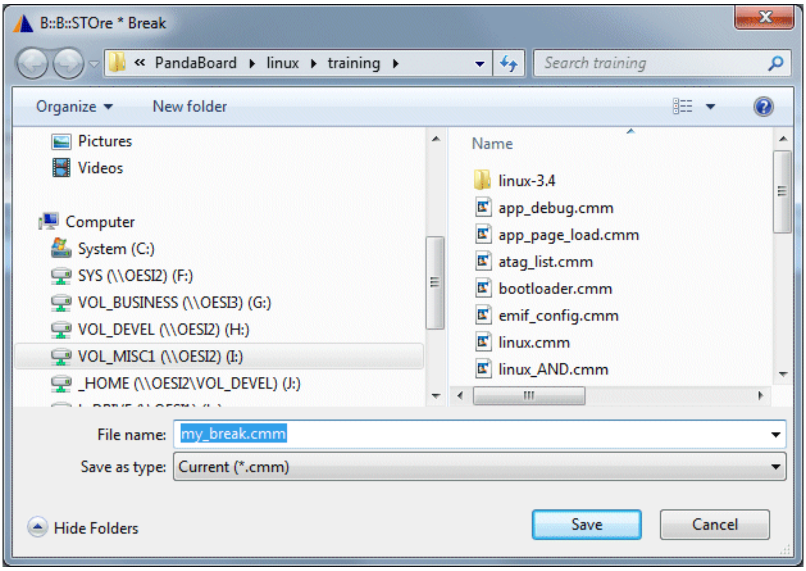
address	types	impl	options	data	
NR:4A3261D8	Program	SOFT	DISableHIT		func4
NR:4A32659C	Program	SOFT			sieve\6
N:4A326FAB--4A326FAB	Write	ONCHIP		BYTE 0x1	flags[3]

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

Store Breakpoint Settings



address	types	impl	options	data	
NR:4A3261D8	Program	SOFT	DISableHIT		func4
NR:4A32659C	Program	SOFT			sieve\6
N:4A326FAB--4A326FAB	Write	ONCHIP		BYTE 0x1	flags[3]



File name: my_break.cmm

Save as type: Current (*.cmm)

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

 B::

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

 ENDDO
```

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

Debugging of Optimized Code

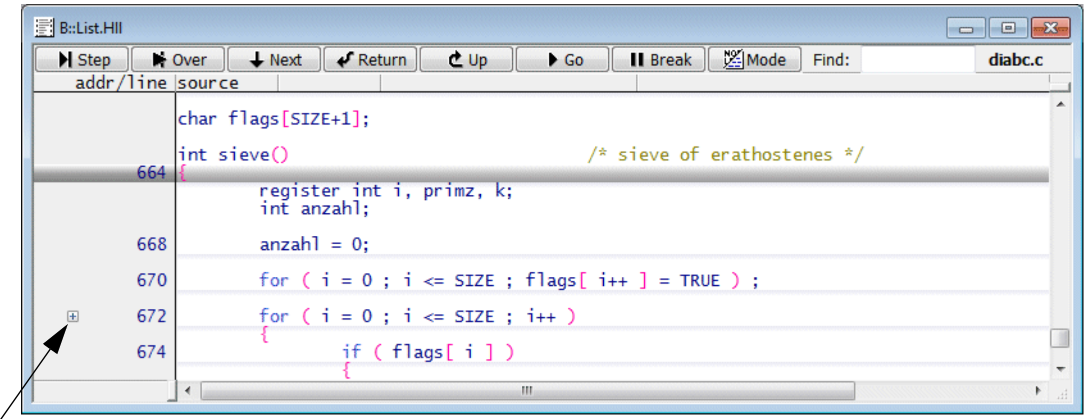
A video tutorial about debugging optimized code can be found here:

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

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

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

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



```
 addr/line source

          char flags[SIZE+1];

          int sieve()                          /* sieve of erathostenes */
       664 {
                 register int i, primz, k;
                 int anzahl;

       668       anzahl = 0;

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

  ⊞    672       for ( i = 0 ; i <= SIZE ; i++ )
                 {
       674             if ( flags[ i ] )
                       {
```

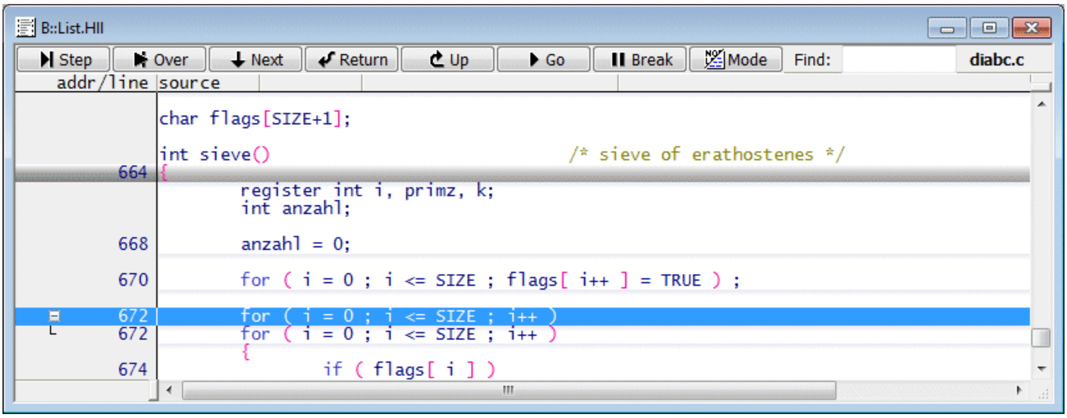
tree button

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

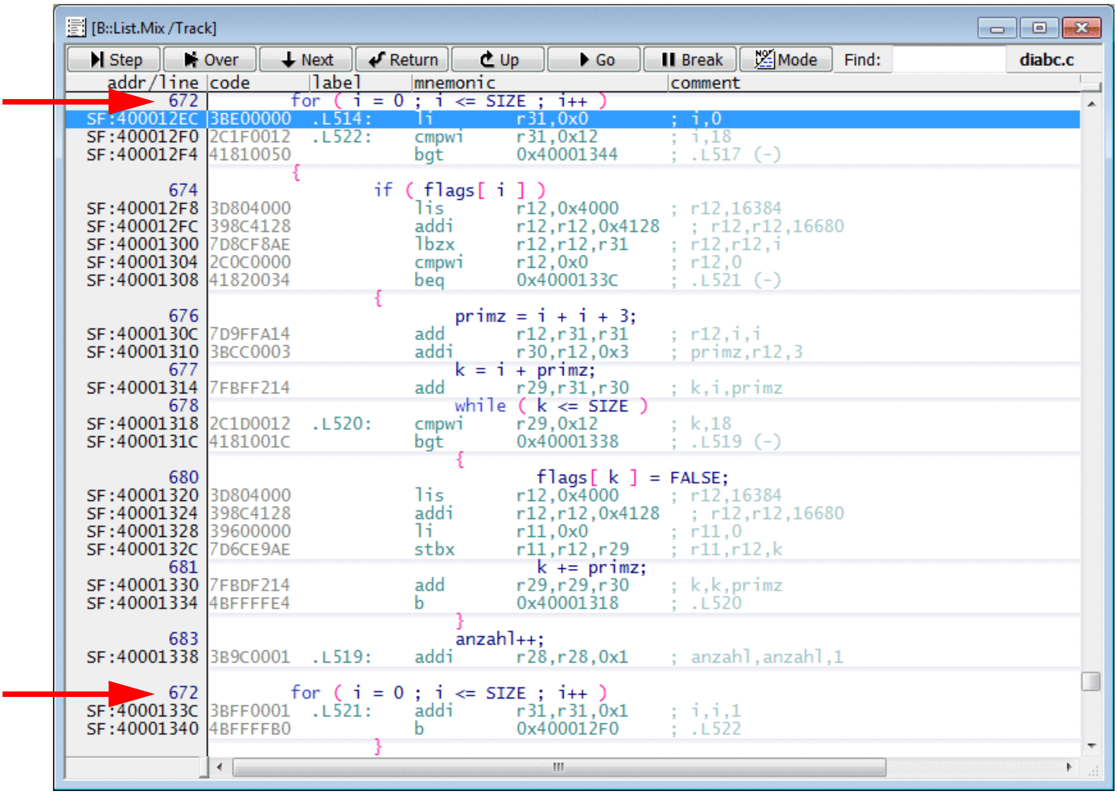
- In HLL debug mode, the HLL code lines are displayed as written in the compiled program (source line order).
- In MIX debug mode, the target code is disassembled and the HLL code lines are displayed together with their assembler code blocks (target line order). This means if two or more detached blocks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



```
      addr/line source

                char flags[SIZE+1];

                int sieve()                       /* sieve of erathostenes */
            664 {
                        register int i, primz, k;
                        int anzahl;

            668         anzahl = 0;

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

   ⊟        672         for ( i = 0 ; i <= SIZE ; i++ )
   └        672         for ( i = 0 ; i <= SIZE ; i++ )
                        {
            674                 if ( flags[ i ] )
```



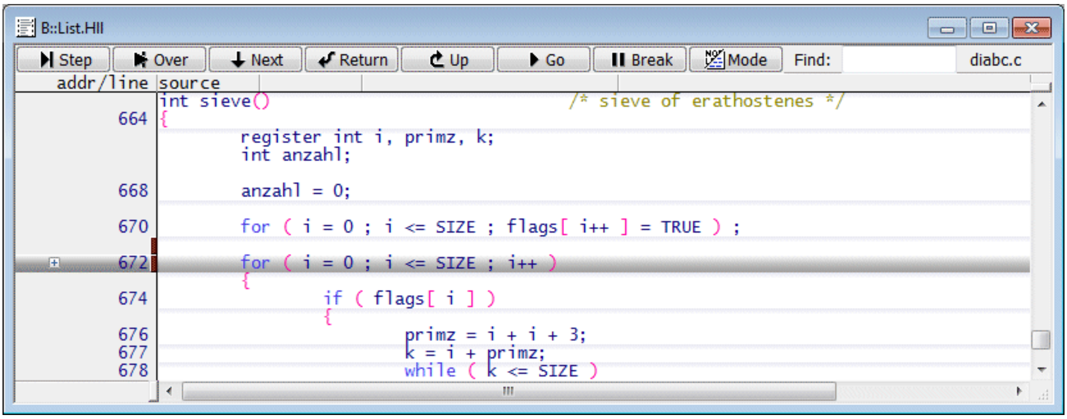
```
    addr/line code     label     mnemonic             comment
          672         for ( i = 0 ; i <= SIZE ; i++ )
 SF:400012EC 3BE00000  .L514:    li        r31,0x0          ; i,0
 SF:400012F0 2C1F0012  .L522:    cmpwi     r31,0x12         ; i,18
 SF:400012F4 41810050            bgt       0x40001344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

          672         for ( i = 0 ; i <= SIZE ; i++ )
 SF:4000133C 3BFF0001  .L521:    addi      r31,r31,0x1      ; i,i,1
 SF:40001340 4BFFFFB0            b         0x400012F0       ; .L522
                              }
```

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

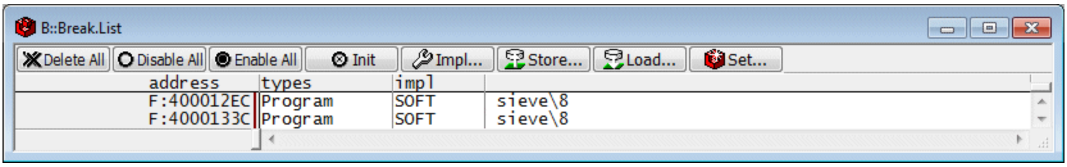
```
List.Hll

List.Mix
```

Please be aware of the following:

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



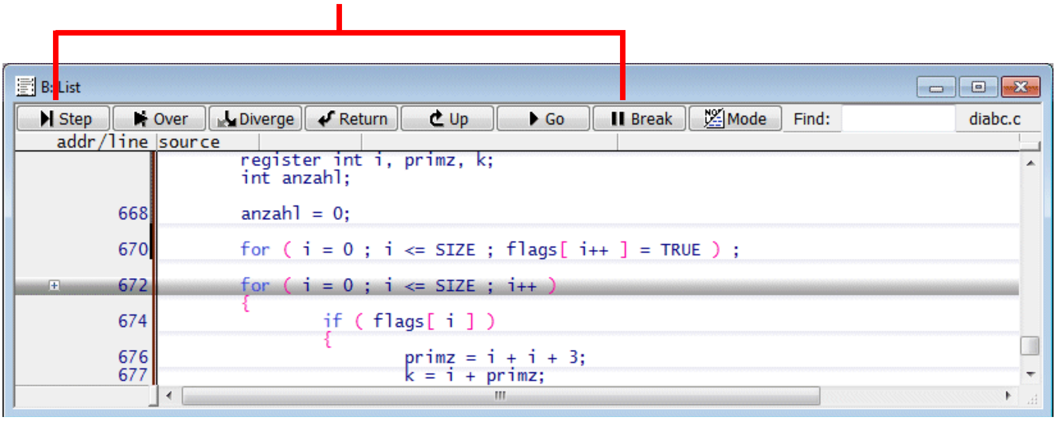
```
int sieve()                                        /* sieve of erathostenes */
{
        register int i, primz, k;
        int anzahl;

        anzahl = 0;

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

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



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

Basic Debug Control

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

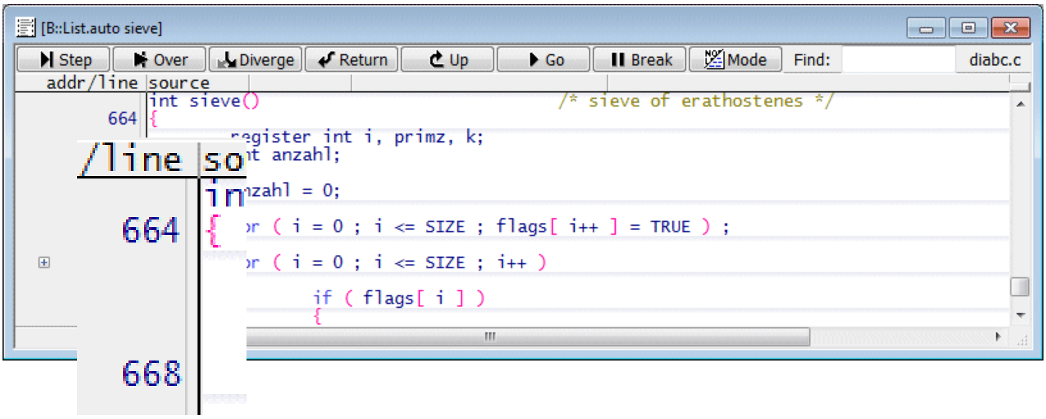


```
        register int i, primz, k;
        int anzahl;

        anzahl = 0;

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

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

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

More details on Step.Diverge

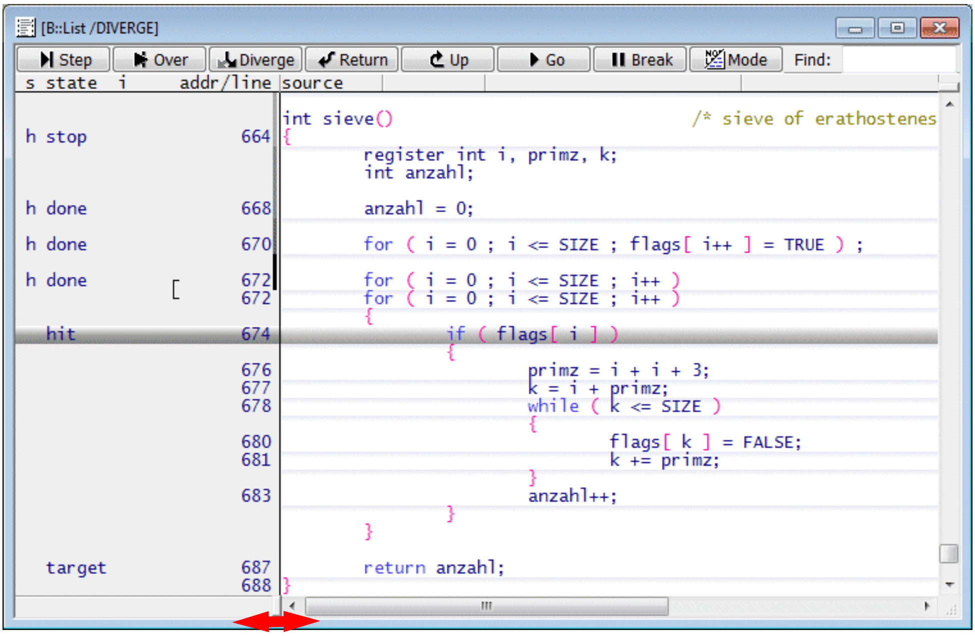
TRACE32 maintains a list of all assembler/HLL lines which were already reached by a Step. These reached lines are marked with a slim grey line in the List window.



```
      int sieve()                              /* sieve of erathostenes */
664   {
          register int i, primz, k;
          int anzahl;

          anzahl = 0;

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

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

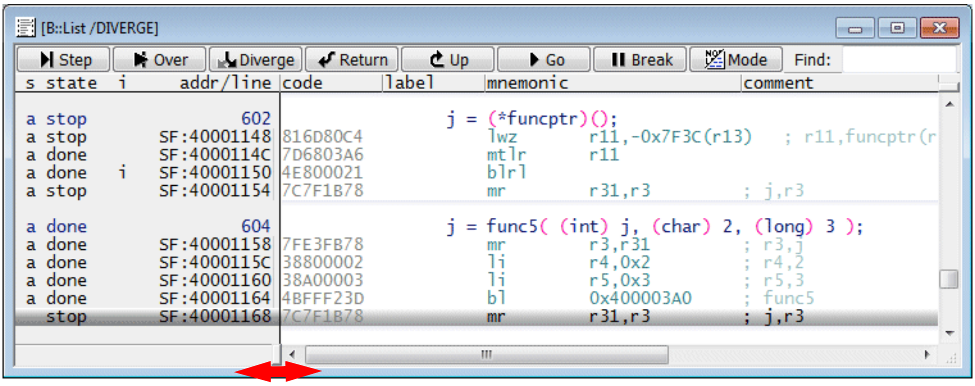
The following command allows you to get more details:

```
List.auto /DIVERGE
```



```
s state  i      addr/line source

                          int sieve()                         /* sieve of erathostenes
h stop              664   {
                                  register int i, primz, k;
                                  int anzahl;

h done              668           anzahl = 0;

h done              670           for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

h done          [   672           for ( i = 0 ; i <= SIZE ; i++ )
                    672           for ( i = 0 ; i <= SIZE ; i++ )
                                  {
  hit               674                   if ( flags[ i ] )
                                          {
                    676                           primz = i + i + 3;
                    677                           k = i + primz;
                    678                           while ( k <= SIZE )
                                                  {
                    680                                   flags[ k ] = FALSE;
                    681                                   k += primz;
                                                  }
                    683                           anzahl++;
                                          }
                                  }

  target            687           return anzahl;
                    688   }
```

Drag this handle to see the DIVERGE details

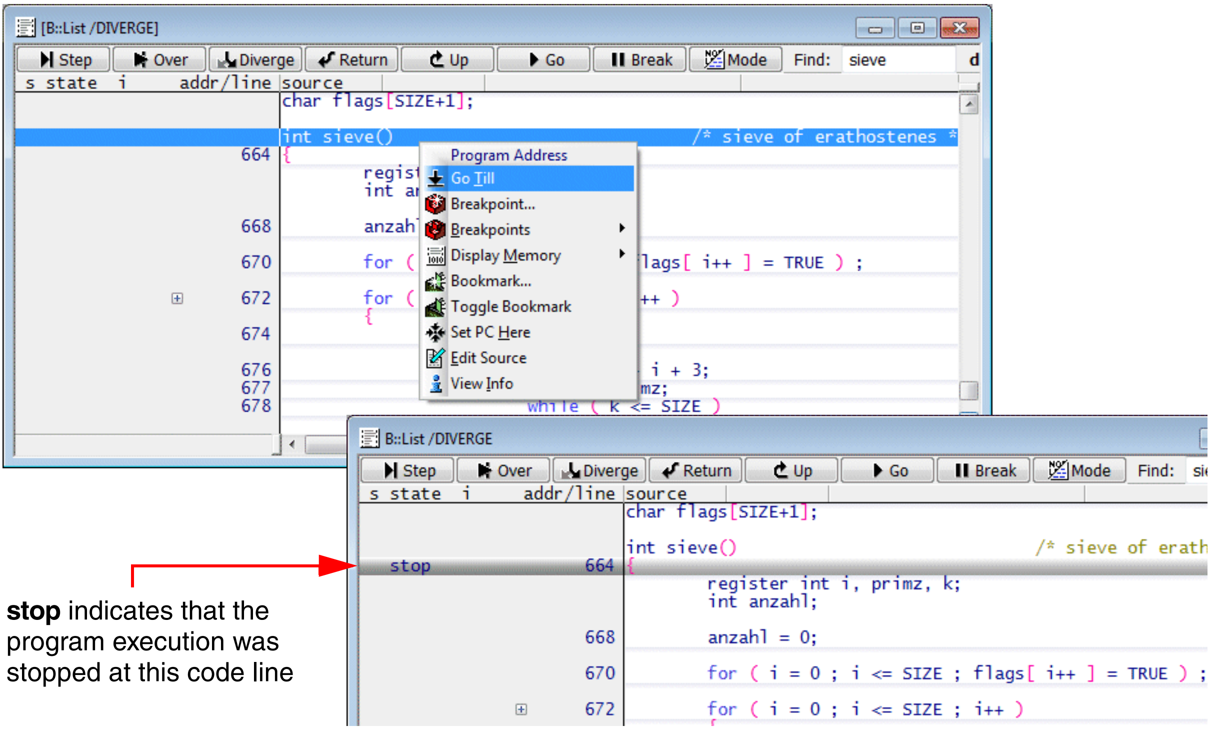


```
s state  i      addr/line code     |label     |mnemonic                |comment

a stop                602                  j = (*funcptr)();
a stop      SF:40001148 816D80C4           lwz        r11,-0x7F3C(r13)    ; r11,funcptr(r
a done      SF:4000114C 7D6803A6           mtlr       r11
a done   i  SF:40001150 4E800021           blrl
a stop      SF:40001154 7C7F1B78           mr         r31,r3              ; j,r3

a done                604                  j = func5( (int) j, (char) 2, (long) 3 );
a done      SF:40001158 7FE3FB78           mr         r3,r31              ; r3,j
a done      SF:4000115C 38800002           li         r4,0x2              ; r4,2
a done      SF:40001160 38A00003           li         r5,0x3              ; r5,3
a done      SF:40001164 4BFFF23D           bl         0x400003A0          ; func5
  stop      SF:40001168 7C7F1B78           mr         r31,r3              ; j,r3
```

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

Example 1: Diverge through function sieve.

1. Run program execution until entry to function sieve.



```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                     /* sieve of erathostenes *
                     664  {
                                  regis
                                  int ar

                     668          anzah

                     670          for (                 lags[ i++ ] = TRUE ) ;

                     672          for (                 ++ )
                                  {
                     674

                     676                                 i + 3;
                     677                                mz;
                     678                  while ( k <= SIZE )
```

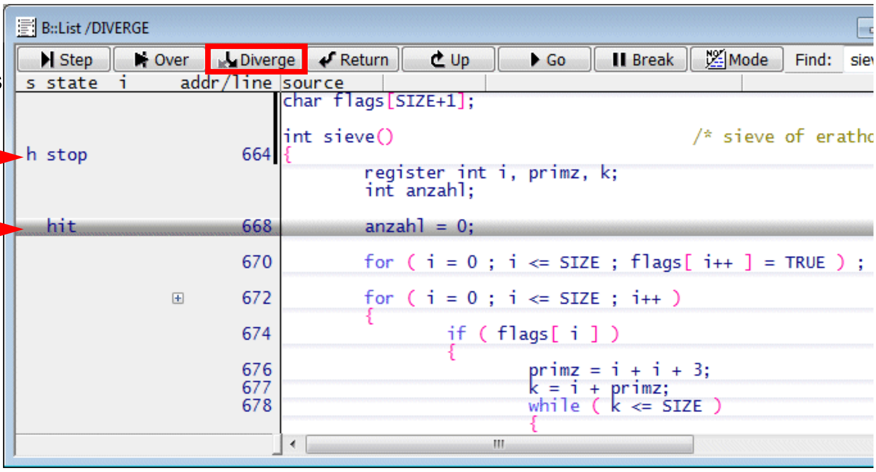
```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                        /* sieve of erath
   stop              664  {
                                  register int i, primz, k;
                                  int anzahl;

                     668          anzahl = 0;

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

                     672          for ( i = 0 ; i <= SIZE ; i++ )
```

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

2. Start a Step.Diverge command.

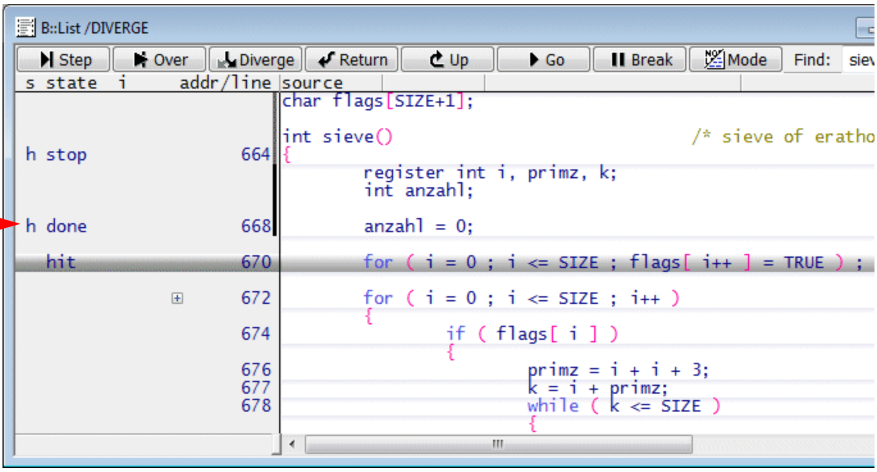
h indicates that a Step command in HLL mode was started in this line

hit indicates that this code line was reached by Step command



```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                        /* sieve of eratho
h stop               664  {
                                  register int i, primz, k;
                                  int anzahl;

  hit                668          anzahl = 0;

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

                     672          for ( i = 0 ; i <= SIZE ; i++ )
                                  {
                     674                  if ( flags[ i ] )
                                          {
                     676                          primz = i + i + 3;
                     677                          k = i + primz;
                     678                          while ( k <= SIZE )
                                                  {
```

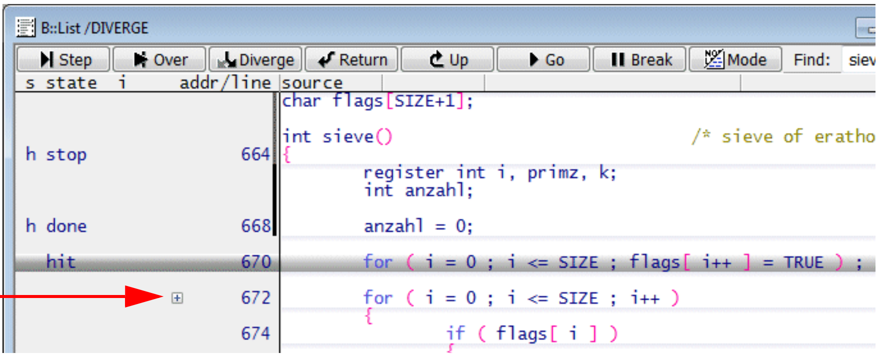
3. Continue with Step.Diverge.



```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                          /* sieve of eratho
h stop               664  {
                                  register int i, primz, k;
                                  int anzahl;

h done               668          anzahl = 0;
  hit                670          for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

              ⊞      672          for ( i = 0 ; i <= SIZE ; i++ )
                                  {
                     674              if ( flags[ i ] )
                                      {
                     676                  primz = i + i + 3;
                     677                  k = i + primz;
                     678                  while ( k <= SIZE )
                                          {
```

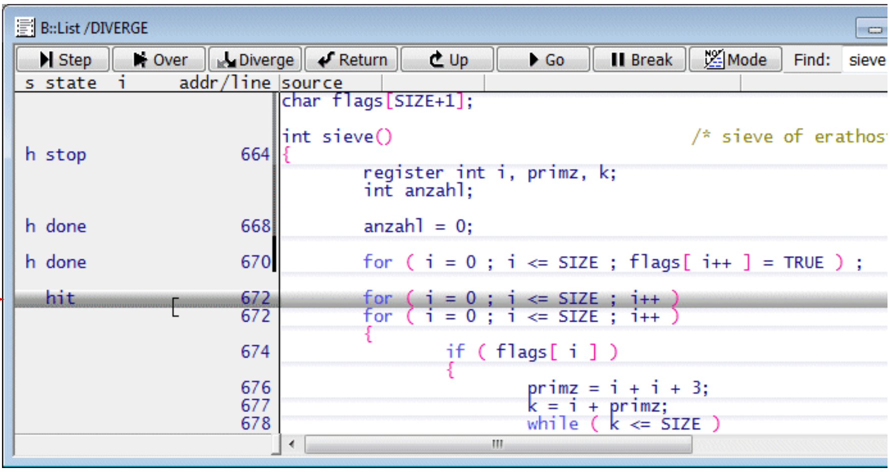
done indicates that the code line was reached by a Step command and that a Step command was started from this code line

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



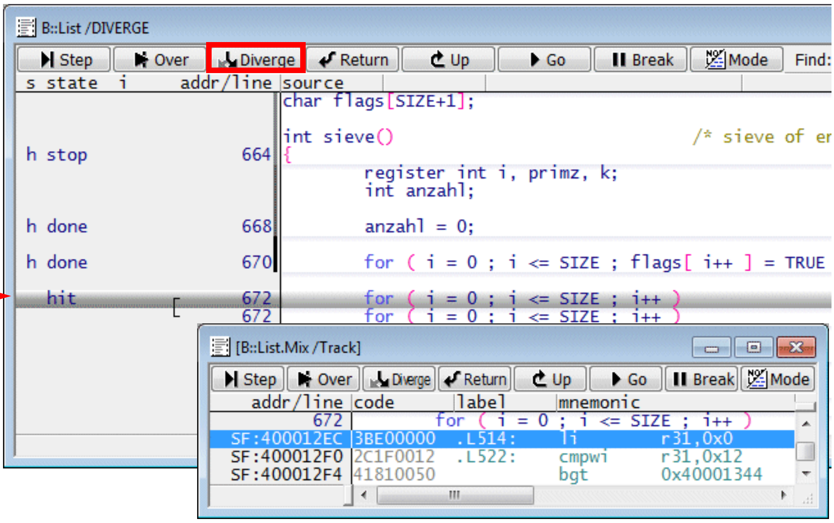
```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                          /* sieve of eratho
h stop               664  {
                                  register int i, primz, k;
                                  int anzahl;

h done               668          anzahl = 0;

  hit                670          for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

                 ⊞   672          for ( i = 0 ; i <= SIZE ; i++ )
                                  {
                     674                  if ( flags[ i ] )
```

4. Continue with Step.Diverge.



```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                          /* sieve of erathos
h stop               664  {
                                  register int i, primz, k;
                                  int anzahl;

h done               668          anzahl = 0;

h done               670          for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

  hit           [    672          for ( i = 0 ; i <= SIZE ; i++ )
                     672          for ( i = 0 ; i <= SIZE ; i++ )
                                  {
                     674                  if ( flags[ i ] )
                                          {
                     676                          primz = i + i + 3;
                     677                          k = i + primz;
                     678                          while ( k <= SIZE )
```

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

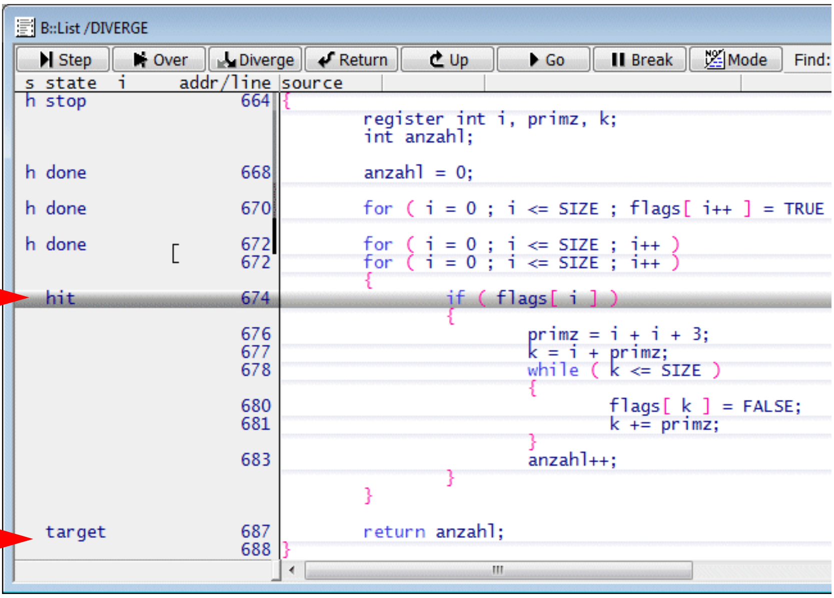


```
s state  i     addr/line source
                         char flags[SIZE+1];

                         int sieve()                     /* sieve of er
h stop               664 {
                                 register int i, primz, k;
                                 int anzahl;

h done               668         anzahl = 0;

h done               670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE

  hit          [     672         for ( i = 0 ; i <= SIZE ; i++ )
                     672         for ( i = 0 ; i <= SIZE ; i++ )
```

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

This HLL code line includes a conditional branch

5. Continue with Step.Diverge.



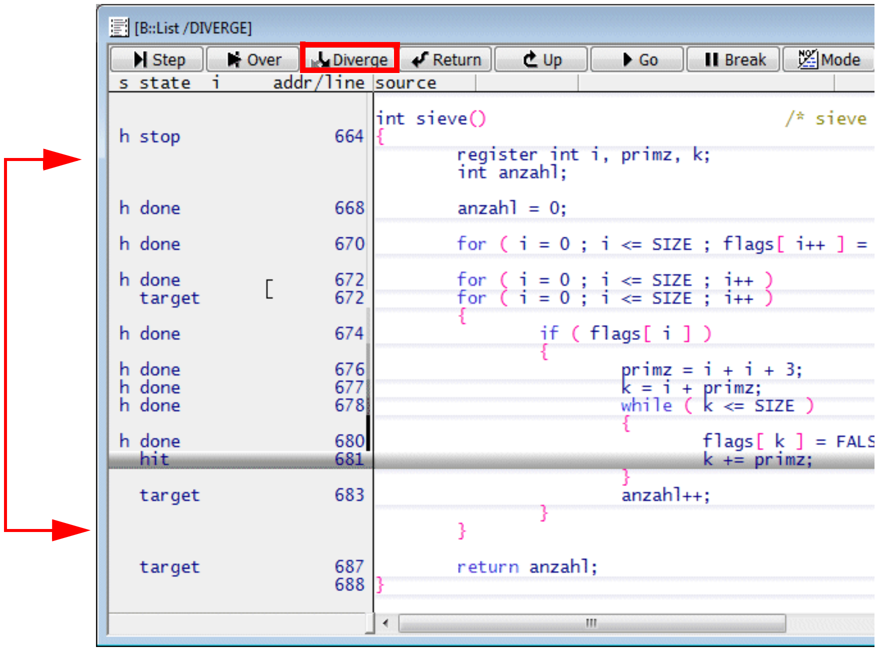
```
s state  i     addr/line source
h stop               664 {
                                 register int i, primz, k;
                                 int anzahl;

h done               668         anzahl = 0;

h done               670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE

h done         [     672         for ( i = 0 ; i <= SIZE ; i++ )
                     672         for ( i = 0 ; i <= SIZE ; i++ )
                                 {
  hit                674                 if ( flags[ i ] )
                                         {
                     676                         primz = i + i + 3;
                     677                         k = i + primz;
                     678                         while ( k <= SIZE )
                                                 {
                     680                                 flags[ k ] = FALSE;
                     681                                 k += primz;
                                                 }
                     683                         anzahl++;
                                         }
                                 }

  target             687         return anzahl;
                     688 }
```

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

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

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

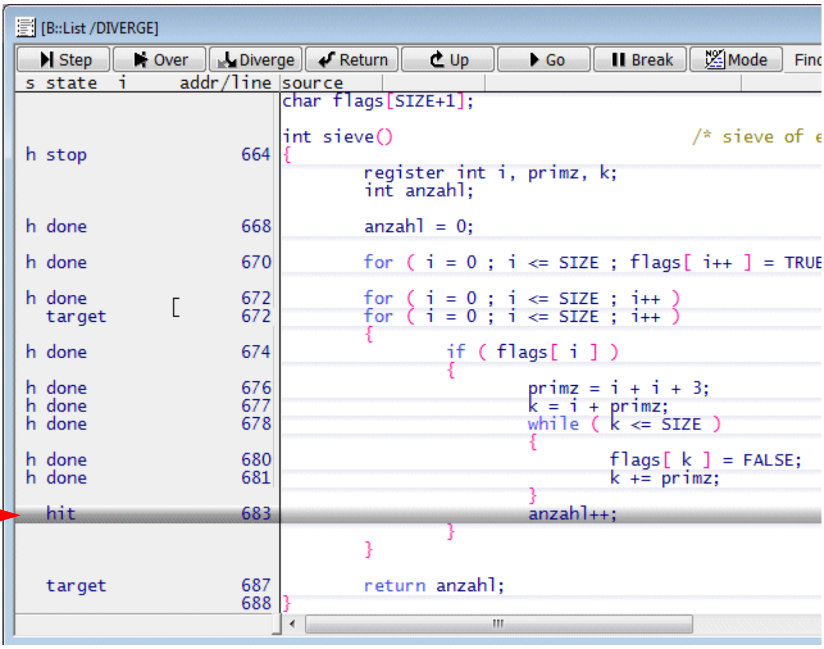
All code lines are now either marked as **done, hit** or **target**



```
s state  i      addr/line source
                          int sieve()                          /* sieve
h stop              664   {
                                  register int i, primz, k;
                                  int anzahl;

h done              668           anzahl = 0;

h done              670           for ( i = 0 ; i <= SIZE ; flags[ i++ ] =

h done        [     672           for ( i = 0 ; i <= SIZE ; i++ )
  target            672           for ( i = 0 ; i <= SIZE ; i++ )
                                  {
h done              674                   if ( flags[ i ] )
                                          {
h done              676                           primz = i + i + 3;
h done              677                           k = i + primz;
h done              678                           while ( k <= SIZE )
                                                  {
h done              680                                   flags[ k ] = FALS
  hit               681                                   k += primz;
                                                  }
  target            683                           anzahl++;
                                          }
                                  }

  target            687           return anzahl;
                    688   }
```

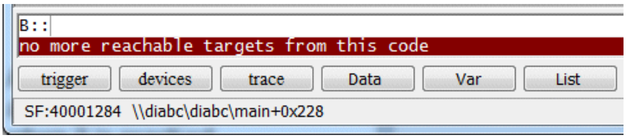
7. **Continue with Step.Diverge.**

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



```
s state  i      addr/line source
                          char flags[SIZE+1];

                          int sieve()                          /* sieve of e
h stop              664   {
                                  register int i, primz, k;
                                  int anzahl;

h done              668           anzahl = 0;

h done              670           for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE

h done        [     672           for ( i = 0 ; i <= SIZE ; i++ )
  target            672           for ( i = 0 ; i <= SIZE ; i++ )
                                  {
h done              674                   if ( flags[ i ] )
                                          {
h done              676                           primz = i + i + 3;
h done              677                           k = i + primz;
h done              678                           while ( k <= SIZE )
                                                  {
h done              680                                   flags[ k ] = FALSE;
h done              681                                   k += primz;
                                                  }
  hit               683                           anzahl++;
                                          }
                                  }

  target            687           return anzahl;
                    688   }
```

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



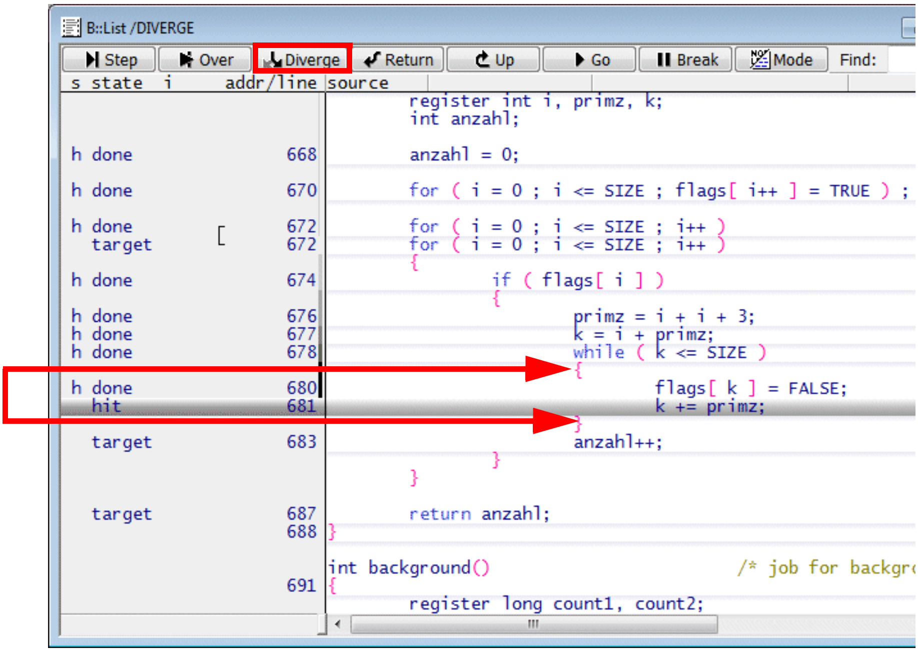
```
B::
no more reachable targets from this code
SF:40001284  \\diabc\diabc\main+0x228
```

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

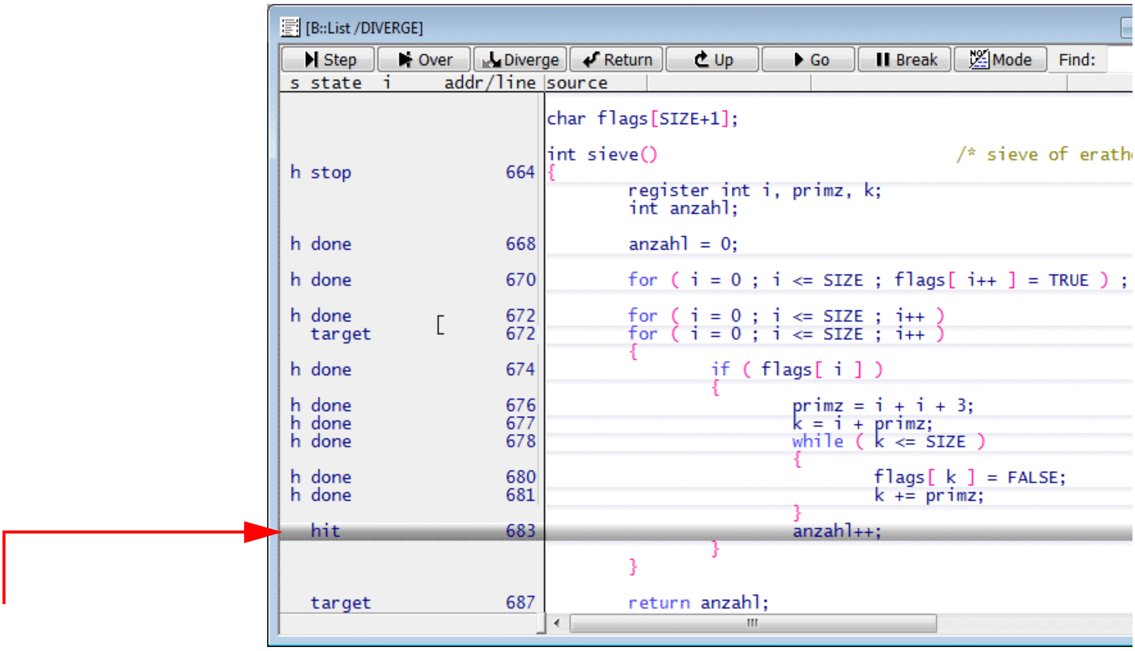
Example 2: Exit a loop.

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



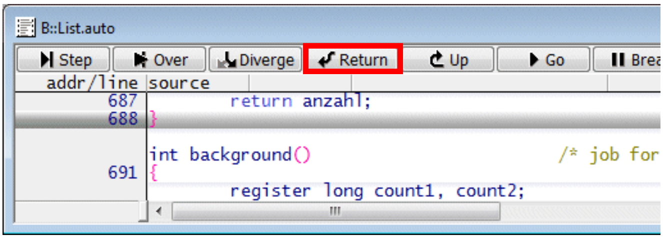
```
s state   i      addr/line source
                                   register int i, primz, k;
                                   int anzahl;

h done                 668         anzahl = 0;

h done                 670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

h done            [    672         for ( i = 0 ; i <= SIZE ; i++ )
  target               672         for ( i = 0 ; i <= SIZE ; i++ )
                                   {
h done                 674                 if ( flags[ i ] )
                                           {
h done                 676                         primz = i + i + 3;
h done                 677                         k = i + primz;
h done                 678                         while ( k <= SIZE )
                                                   {
h done                 680                                 flags[ k ] = FALSE;
  hit                  681                                 k += primz;
                                                   }
  target               683                         anzahl++;
                                           }
                                   }

  target               687         return anzahl;
                       688 }

                           int background()                         /* job for backgr
                       691 {
                                   register long count1, count2;
```



```
s state   i      addr/line source

                           char flags[SIZE+1];

                           int sieve()                              /* sieve of erath
h stop                 664 {
                                   register int i, primz, k;
                                   int anzahl;

h done                 668         anzahl = 0;

h done                 670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;

h done            [    672         for ( i = 0 ; i <= SIZE ; i++ )
  target               672         for ( i = 0 ; i <= SIZE ; i++ )
                                   {
h done                 674                 if ( flags[ i ] )
                                           {
h done                 676                         primz = i + i + 3;
h done                 677                         k = i + primz;
h done                 678                         while ( k <= SIZE )
                                                   {
h done                 680                                 flags[ k ] = FALSE;
h done                 681                                 k += primz;
                                                   }
  hit                  683                         anzahl++;
                                           }
                                   }

  target               687         return anzahl;
```

<table>
<tr>
<td><b>Return</b></td>
<td>

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

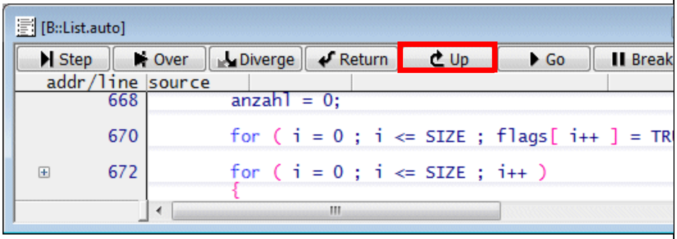


B::List.auto

```
addr/line source
      687         return anzahl;
      688 }

          int background()                    /* job for
      691 {
                  register long count1, count2;
```

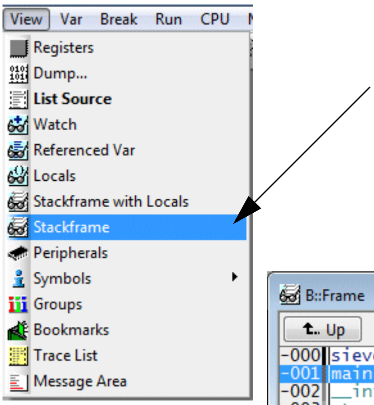
</td>
</tr>
<tr>
<td><b>Up</b></td>
<td>

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

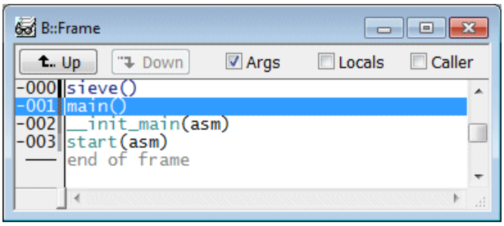


[B::List.auto]

```
addr/line source
      668         anzahl = 0;

      670         for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TR

      672         for ( i = 0 ; i <= SIZE ; i++ )
                  {
```



Display the HLL stack to check the function nesting



B::Frame

```
-000 sieve()
-001 main()
-002 __init_main(asm)
-003 start(asm)
 ——  end of frame
```

</td>
</tr>
</table>

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

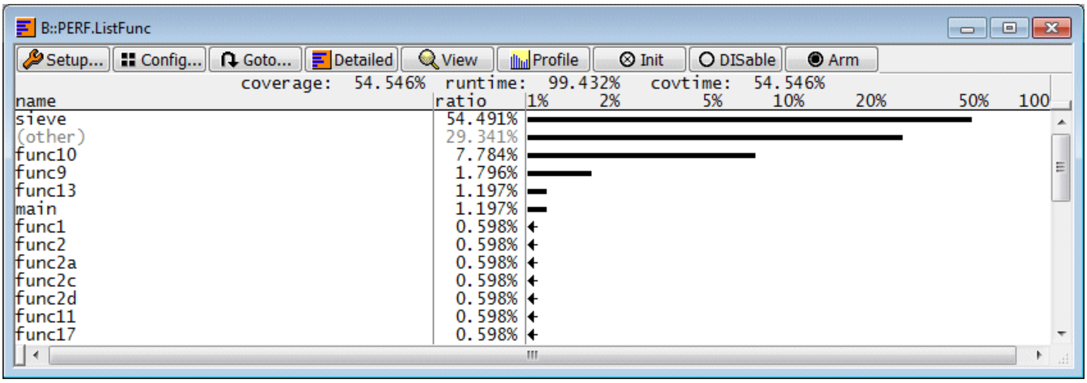
```
Step 10.

Step.Change Register(R11)

Step.Till Register(R11)>0xAA

Var.Step.Change flags[3]

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

Step.Over	Step over call

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

Program Counter Sampling

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



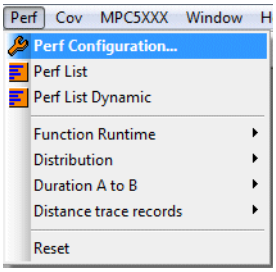
```
                              coverage:   54.546%  runtime:   99.432%   covtime:   54.546%
name                                    ratio     1%       2%       5%      10%      20%       50%    100
sieve                                   54.491%
(other)                                 29.341%
func10                                   7.784%
func9                                    1.796%
func13                                   1.197%
main                                     1.197%
func1                                    0.598%
func2                                    0.598%
func2a                                   0.598%
func2c                                   0.598%
func2d                                   0.598%
func11                                   0.598%
func17                                   0.598%
```

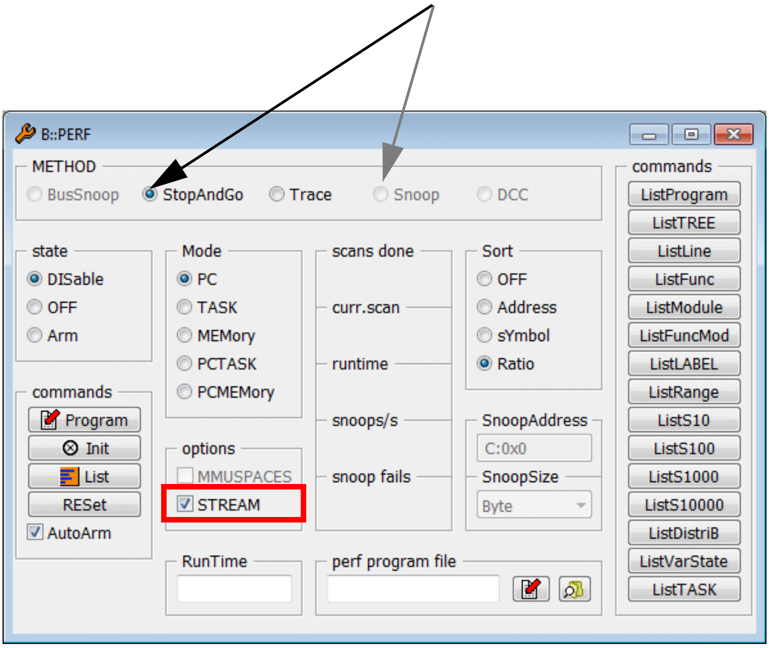
Measurement procedure: The Program Counter is sampled periodically. This is implemented in two ways.

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

Steps to be taken:

1. **Open the PERF configuration window.**





METHOD: ○ BusSnoop ● StopAndGo ○ Trace ○ Snoop ○ DCC

state: ● DISable ○ OFF ○ Arm

Mode: ● PC ○ TASK ○ MEMory ○ PCTASK ○ PCMEMory

Sort: ○ OFF ○ Address ○ sYmbol ● Ratio

options: □ MMUSPACES ☑ STREAM

☑ AutoArm

SnoopAddress: C:0x0

SnoopSize: Byte

PERF.state	Display PERF configuration window

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

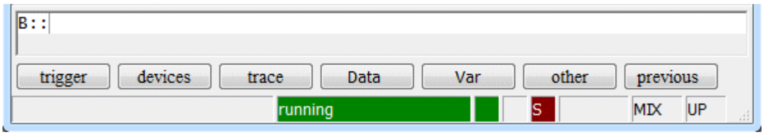
The default METHOD for all other processor architectures is **StopAndGo**.

Remarks on the StopAndGo method

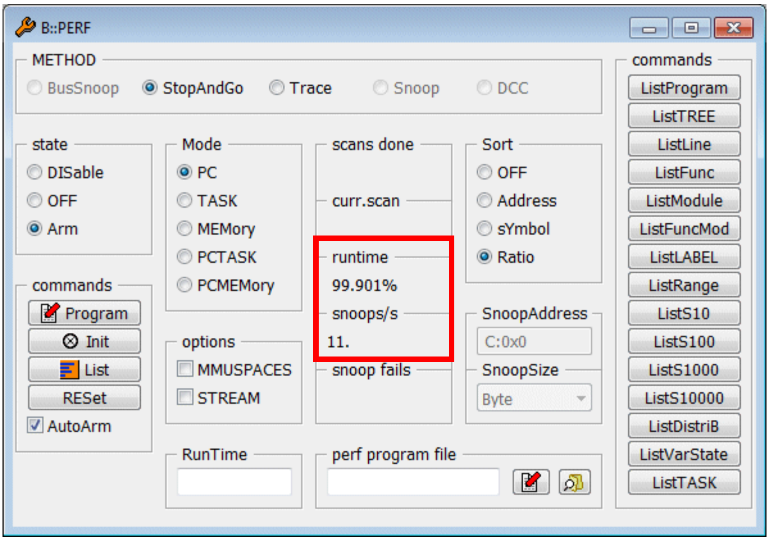
StopAnd Go means that the core is stopped periodically in order to get the actual Program Counter.

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

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



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


B::PERF

METHOD: ○ BusSnoop ● StopAndGo ○ Trace ○ Snoop ○ DCC

state: ○ DISable ○ OFF ● Arm

Mode: ● PC ○ TASK ○ MEMory ○ PCTASK ○ PCMEMory

scans done

curr.scan

runtime
99.901%

snoops/s
11.

snoop fails

Sort: ○ OFF ○ Address ○ sYmbol ● Ratio

SnoopAddress: C:0x0

SnoopSize: Byte

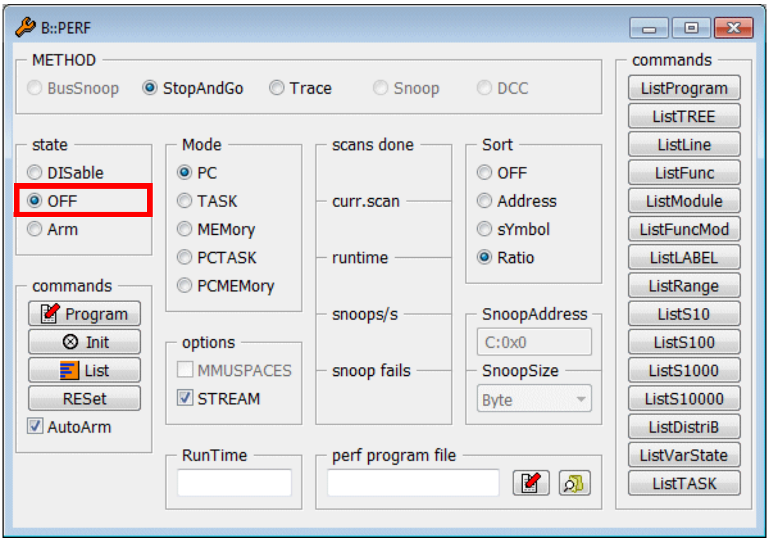
options: □ MMUSPACES □ STREAM

☑ AutoArm

RunTime

perf program file

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



METHOD: ○ BusSnoop ● StopAndGo ○ Trace ○ Snoop ○ DCC

state: ○ DISable ● OFF ○ Arm

Mode: ● PC ○ TASK ○ MEMory ○ PCTASK ○ PCMEMory

Sort: ○ OFF ○ Address ○ sYmbol ● Ratio

options: □ MMUSPACES ☑ STREAM

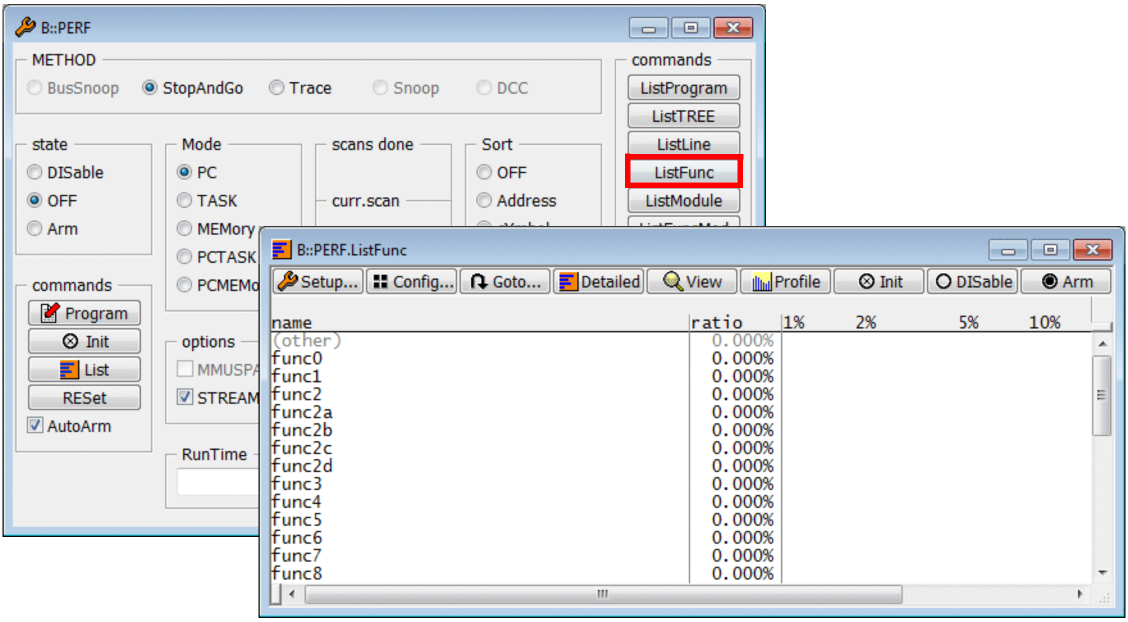
commands: ☑ AutoArm

SnoopAddress: C:0x0

SnoopSize: Byte

PERF.OFF	Enable the sample-based profiling

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



METHOD: ○ BusSnoop ● StopAndGo ○ Trace ○ Snoop ○ DCC

state: ○ DISable ● OFF ○ Arm

Mode: ● PC ○ TASK ○ MEMory ○ PCTASK ○ PCMEMory

Sort: ○ OFF ○ Address

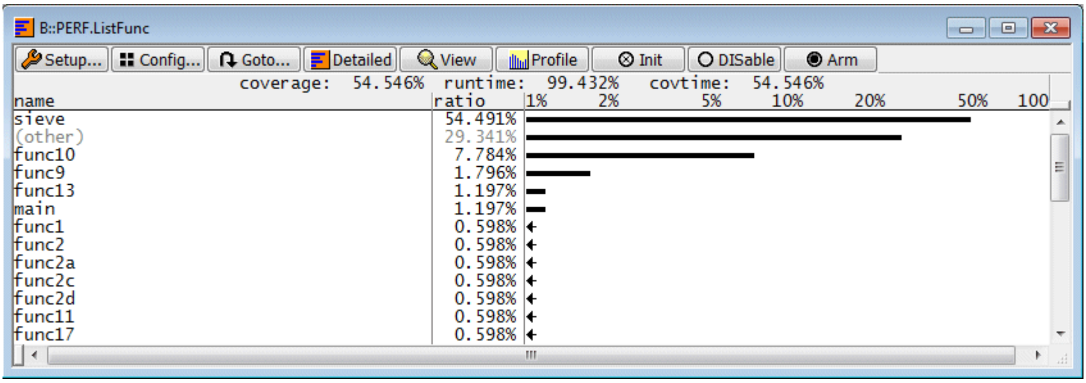
options: □ MMUSPA ☑ STREAM

commands: ☑ AutoArm

```
name                  ratio     1%    2%       5%     10%
(other)               0.000%
func0                 0.000%
func1                 0.000%
func2                 0.000%
func2a                0.000%
func2b                0.000%
func2c                0.000%
func2d                0.000%
func3                 0.000%
func4                 0.000%
func5                 0.000%
func6                 0.000%
func7                 0.000%
func8                 0.000%
```

PERF.ListFunc	Open an HLL function profiling window

4. Start the program execution and the sampling.

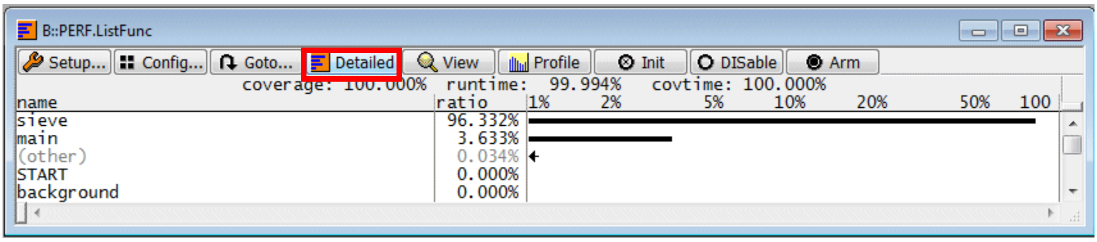


```
                     coverage:  54.546%  runtime:  99.432%   covtime:  54.546%
name                                    ratio
sieve                                    54.491%
(other)                                  29.341%
func10                                    7.784%
func9                                     1.796%
func13                                    1.197%
main                                      1.197%
func1                                     0.598%
func2                                     0.598%
func2a                                    0.598%
func2c                                    0.598%
func2d                                    0.598%
func11                                    0.598%
func17                                    0.598%
```

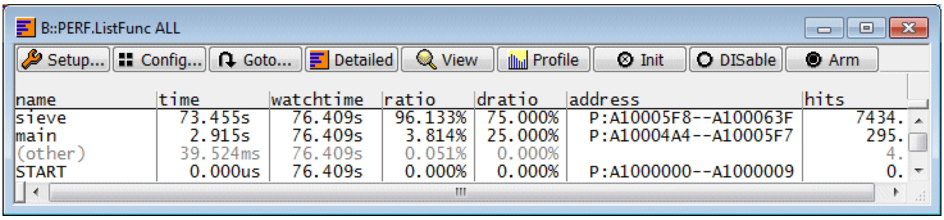
Details

In-depth Result

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



```
                              coverage: 100.000%  runtime:   99.994%   covtime: 100.000%
name                                   ratio     1%      2%        5%     10%      20%        50%   100
sieve                                  96.332%
main                                    3.633%
(other)                                 0.034%
START                                   0.000%
background                              0.000%
```

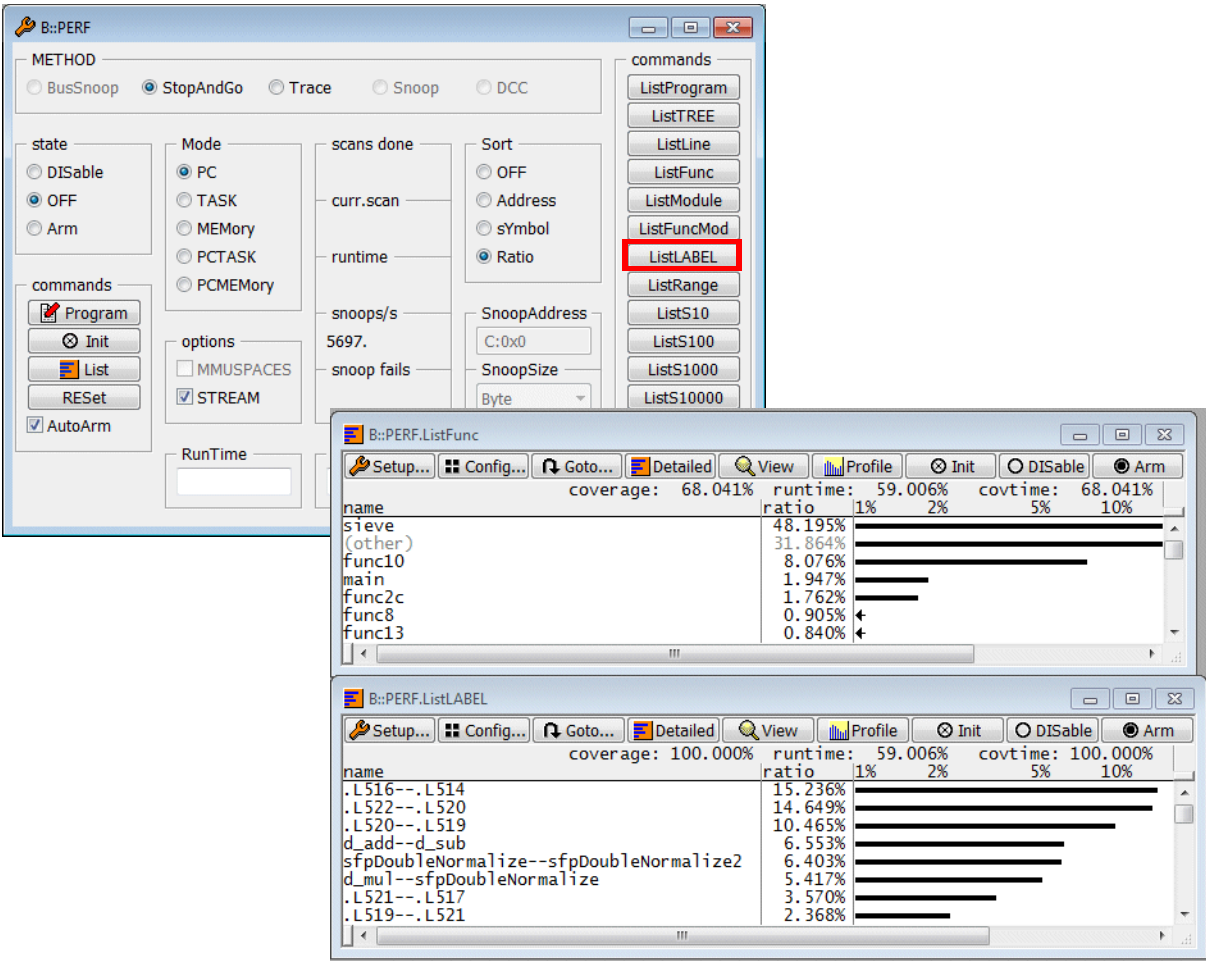


```
name          time      watchtime  ratio    dratio   address                  hits
sieve         73.455s   76.409s    96.133%  75.000%  P:A10005F8--A100063F    7434.
main           2.915s   76.409s     3.814%  25.000%  P:A10004A4--A10005F7     295.
(other)       39.524ms  76.409s     0.051%   0.000%                             4.
START          0.000us  76.409s     0.000%   0.000%  P:A1000000--A1000009       0.
```

PERF.ListFunc ALL Open a detailed HLL function profiling window

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

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



METHOD: ○ BusSnoop ● StopAndGo ○ Trace ○ Snoop ○ DCC

state: ○ DISable ● OFF ○ Arm

Mode: ● PC ○ TASK ○ MEMory ○ PCTASK ○ PCMEMory

scans done

curr.scan

runtime

snoops/s 5697.

snoop fails

Sort: ○ OFF ○ Address ○ sYmbol ● Ratio

SnoopAddress C:0x0

SnoopSize Byte

options: □ MMUSPACES ☑ STREAM

☑ AutoArm

RunTime

```
                     coverage:  68.041%  runtime:  59.006%   covtime:  68.041%
name                                    ratio   1%     2%          5%      10%
sieve                                   48.195%
(other)                                 31.864%
func10                                   8.076%
main                                     1.947%
func2c                                   1.762%
func8                                    0.905%
func13                                   0.840%
```

```
                     coverage: 100.000%  runtime:  59.006%   covtime: 100.000%
name                                    ratio   1%     2%          5%      10%
.L516--.L514                            15.236%
.L522--.L520                            14.649%
.L520--.L519                            10.465%
d_add--d_sub                             6.553%
sfpDoubleNormalize--sfpDoubleNormalize2  6.403%
d_mul--sfpDoubleNormalize                5.417%
.L521--.L517                             3.570%
.L519--.L521                             2.368%
```

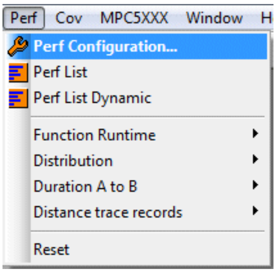
PERF.ListLABEL	Open a window for label-based profiling

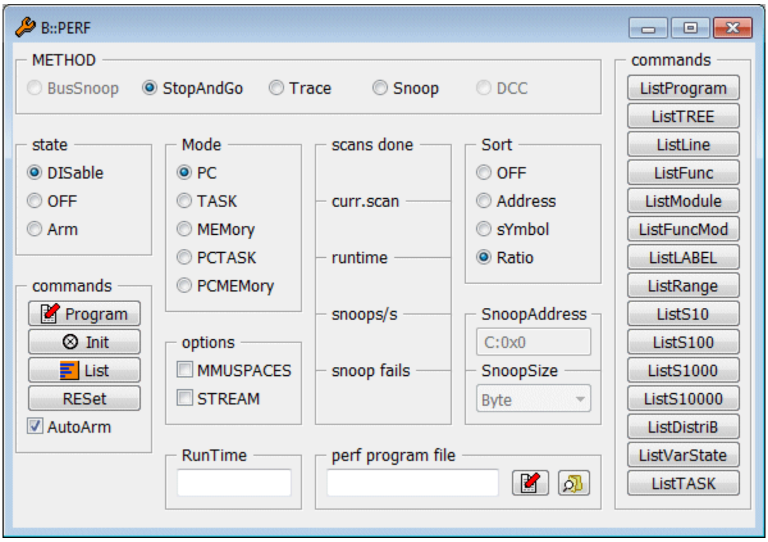
TASK Sampling

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

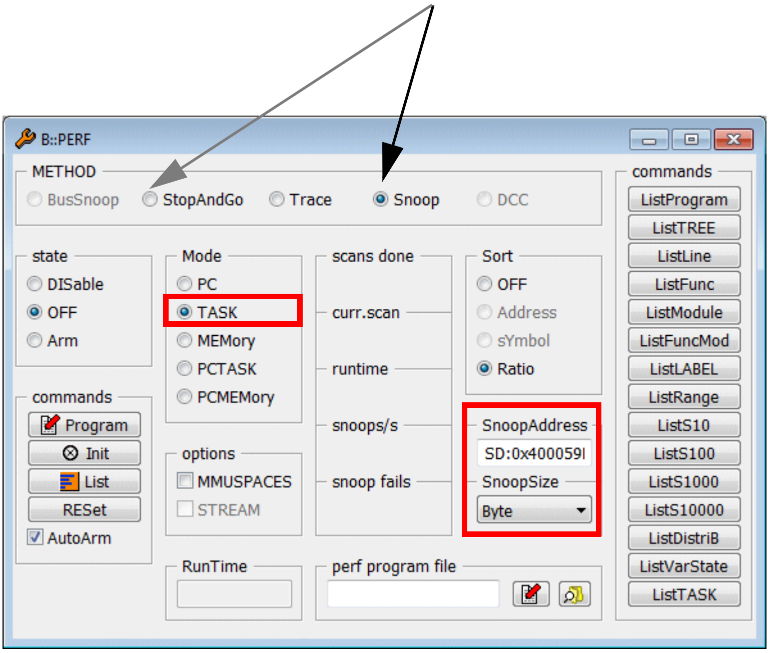
Steps to be taken:

1. Open the PERF configuration window.





2. **Select Mode TASK.**



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

TRACE32 fills the following fields when TASK mode is selected:

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

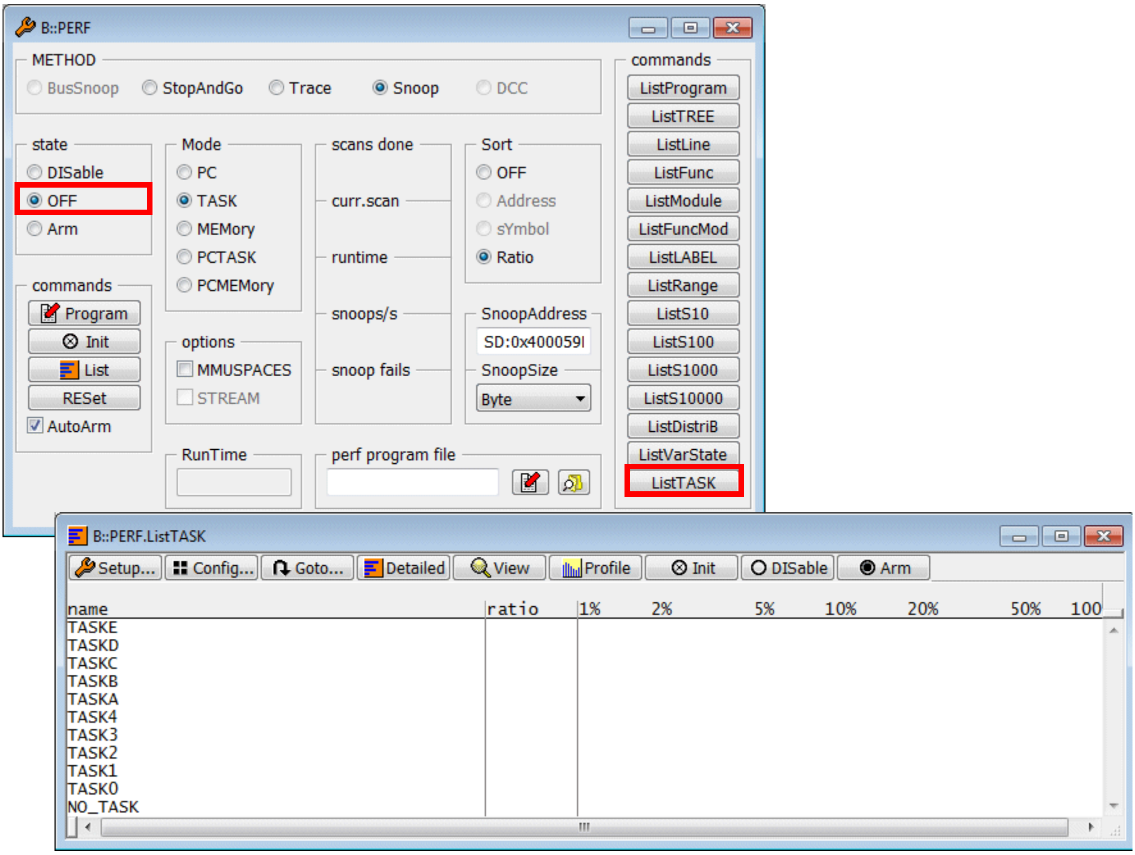
- the **SnoopSize** field with the size of the variable.

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

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

```
PERF.Mode TASK
```

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



METHOD: ○ BusSnoop ○ StopAndGo ○ Trace ● Snoop ○ DCC

state: ○ DISable ● OFF ○ Arm

Mode: ○ PC ● TASK ○ MEMory ○ PCTASK ○ PCMEMory

Sort: ○ OFF ○ Address ○ sYmbol ● Ratio

options: □ MMUSPACES □ STREAM

☑ AutoArm

SnoopAddress: SD:0x400059l

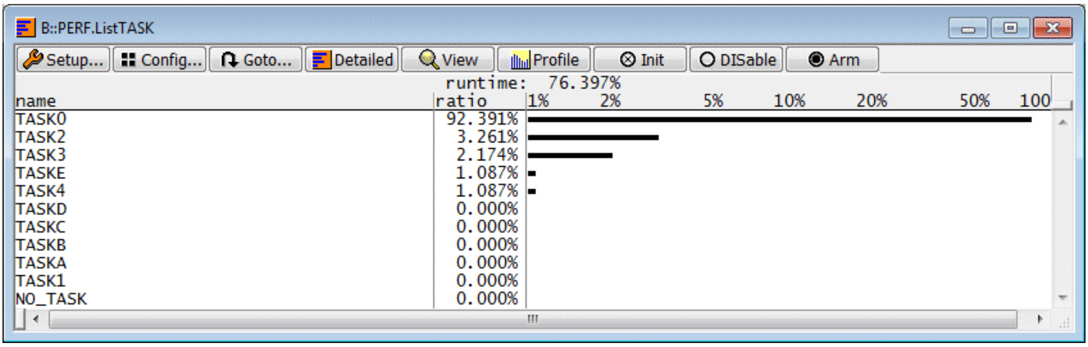
SnoopSize: Byte

```
name                                      ratio    1%    2%     5%    10%    20%     50%   100
TASKE
TASKD
TASKC
TASKB
TASKA
TASK4
TASK3
TASK2
TASK1
TASK0
NO_TASK
```

PERF.OFF Enable the sample-based profiling

PERF.ListTASK

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



```
                                     runtime:  76.397%
name                                 ratio   1%    2%     5%    10%    20%     50%   100
TASK0                                92.391%
TASK2                                 3.261%
TASK3                                 2.174%
TASKE                                 1.087%
TASK4                                 1.087%
TASKD                                 0.000%
TASKC                                 0.000%
TASKB                                 0.000%
TASKA                                 0.000%
TASK1                                 0.000%
NO_TASK                               0.000%
```