LAUTERBACH
DEVELOPMENT TOOLS

Training Source Level Debugging

Release 02.2023

MANUAL

Training Source Level Debugging

Training Source Level Debugging

Version 10-Feb-2023

Load the Application Program

Data.LOAD *<filename>* [**/***<option>*]	General load command
Data.LOAD.<sub_cmd> *<filename>* [**/***<option>*]	Compiler specific load command

It is recommended to use the compiler/format specific **Data.LOAD** command thereby all compiler/format specific options can be used.

Which actions are performed by TRACE32 when the Data.LOAD command is executed?

- All symbol and debug information already available in TRACE32 is removed.
- The code/data provided by **<file>** is loaded to the target memory.
- The symbol and debug information provided by **<file>** is loaded into TRACE32.
- The paths for the HLL source files provided by **<file>** are loaded into TRACE32.
- A TRACE32 symbol database is generated out of the loaded information.

Options that refer to Code/data

The options that refer to code/data are mainly used for the following tasks:

- to verify that code/data is loaded correctly.
- to suppress the loading of code/data if the correct code/data is already in the target.

DIFF	Data in the memory is compared against the file, the memory is not changed. • FOUND() returns TRUE, when a difference between the file and the memory is found. • FOUND() returns FALSE, when no difference between the file and the memory is found.
NoCODE	Symbol and debug information plus source path information gets loaded to the debugger, but do no code/data is downloaded to the target memory. Useful if the code/data is already in memory.

```
Data.LOAD.Elf demo.elf
Data.LOAD.Elf demo.elf /DIFF

IF FOUND()
  PRINT %ERROR "Loading of program failed"


Data.LOAD.Elf demo.elf /NoCODE
```

Options that Refer to the Symbol and Debug Information

The options that refer to the symbol and debug information are mainly used to relocate the symbol information.

```
; relocate all symbols by 2000
symbol.RELOCate.shift 2000

; Load the symbol and debug information from the file t_li_elf.axf and
; relocate all symbols of the section t_li_elf.axf to address 3000
sYmbol.List.SECTion
Data.LOAD.Elf thumble.axf /RELOC t_li_elf.axf AT 3000 /NoCODE
```

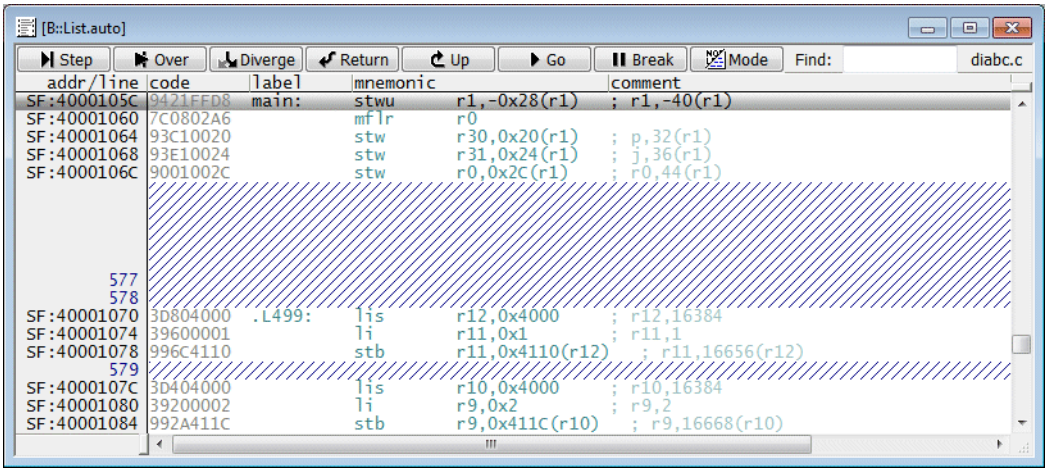
sYmbol.RELOCate.shift *<offset>*	Relocate code and data symbols by *<offset>*
Data.LOAD.Elf ***<file>*** **/RELOC** *<sector>* **AT** *<address>*	Relocate the specified sector to the defined address
Data.LOAD.Elf ***<file>*** **/RELOC** *<sector>* **AFTER** *<sector_other>*	Relocated the specified sector after an another sector
sYmbol.List.SECtion	List the section information of the TRACE32 symbol database

NoClear	By default, whenever a new **Data.LOAD** command is started, the already available symbol and debug information is removed. With this option the already available symbol and debug information is not removed. This option is necessary if more than one program is loaded.
More	This option speeds up the downloading of large projects consisting of several programs. This option suppresses the generation of the internal symbol database when using the **Data.LOAD** command.

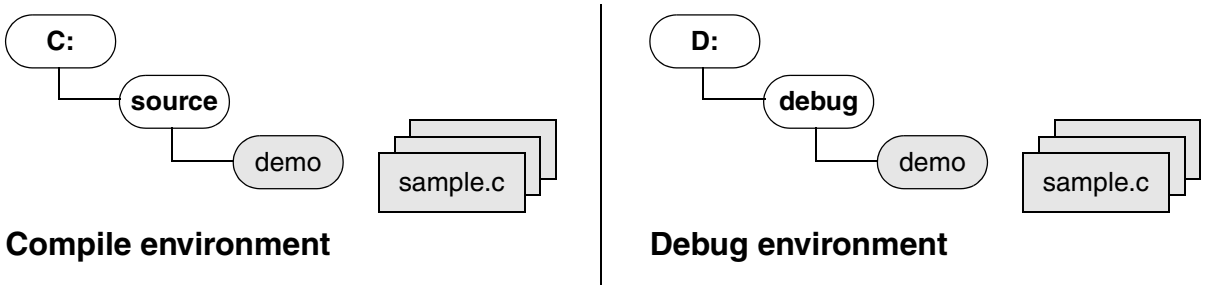
```
Data.LOAD file1 /More                   ; load file1 but suppress the
                                        ; generation of the internal
                                        ; symbol database

Data.LOAD file2 /NoClear /More          ; load file2 but don't remove the
                                        ; already available symbol and
                                        ; debug information before
                                        ; loading and suppress the
                                        ; generation of the internal
                                        ; symbol database

Data.LOAD file3 /NoClear /More

.

.

.

Data.LOAD filen /NoClear                ; load filen but don't remove the
                                        ; already available symbol and
                                        ; debug information before
                                        ; loading, this is the last file
                                        ; so generate the internal symbol
                                        ; database now
```

A video tutorial about the source path correction can be found here:

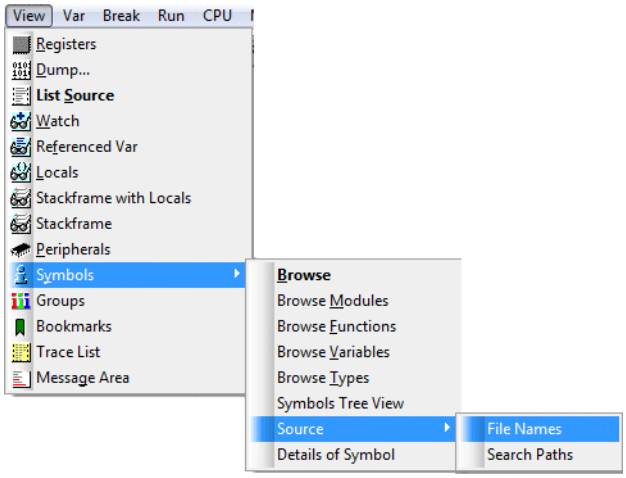
https://www.lauterbach.com/tut_sourcecode.html

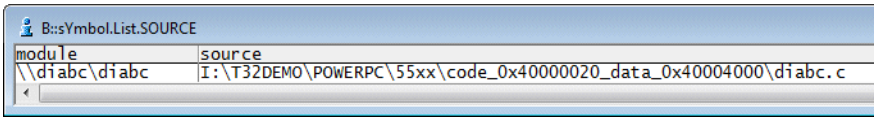


If the **Source Listing** displays hatched areas instead of the source code information, the source code paths provided by the loaded program have to be corrected. These corrections become necessary because the compile environment differs from the debug environment. The graphic below shows a very simple example.

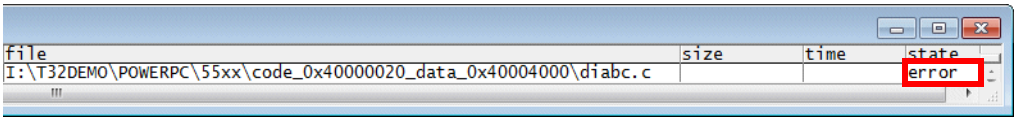


To inspect the paths for the source code files provided by the loaded program proceed as shown below:





The compile paths provided by the loaded program are listed in the **source** column



error
in the state column indicates that a required file was not found in the current debug environment

sYmbol.List.SOURCE	Display source file details.

TRACE32 provides the following ways to correct the compile paths so they fit the paths in the debug environment:

Example 1: Provide the source paths directly

+ Quick and easy
+ Recommended for small project
+ Source paths can be corrected without reloading the program

Example 2: Translate compile path to debug path

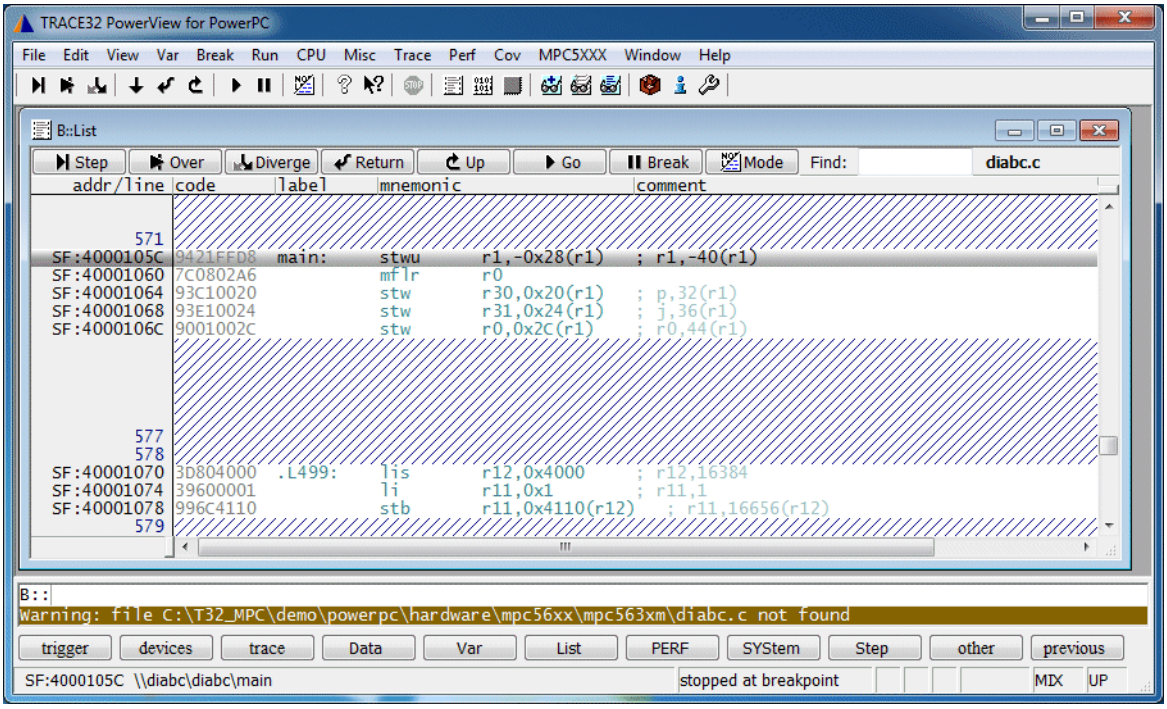
+ Recommended for large projects
+ Source paths can be corrected without reloading the program
+ Not flexible enough for a generic script

Example 3: Personalized debug paths

+ Recommended for large projects
+ Flexible for generic scripts
+ Requires a fixed location for the script that loads the program

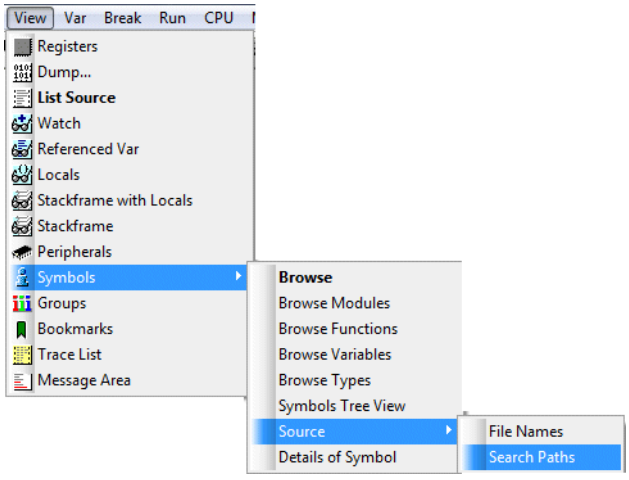
Example 4: Convert cygdrive paths to Window paths

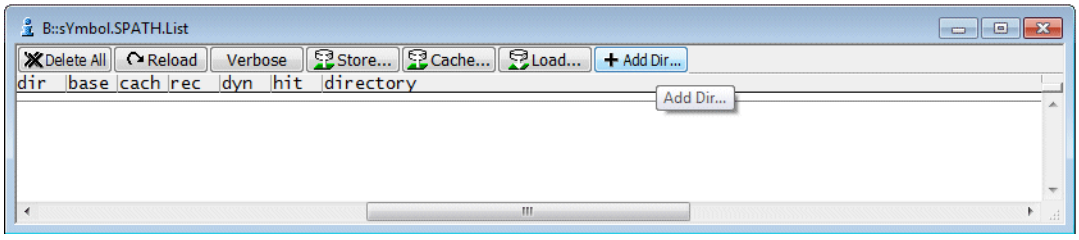
Example 5: Load Elf file with relative paths only



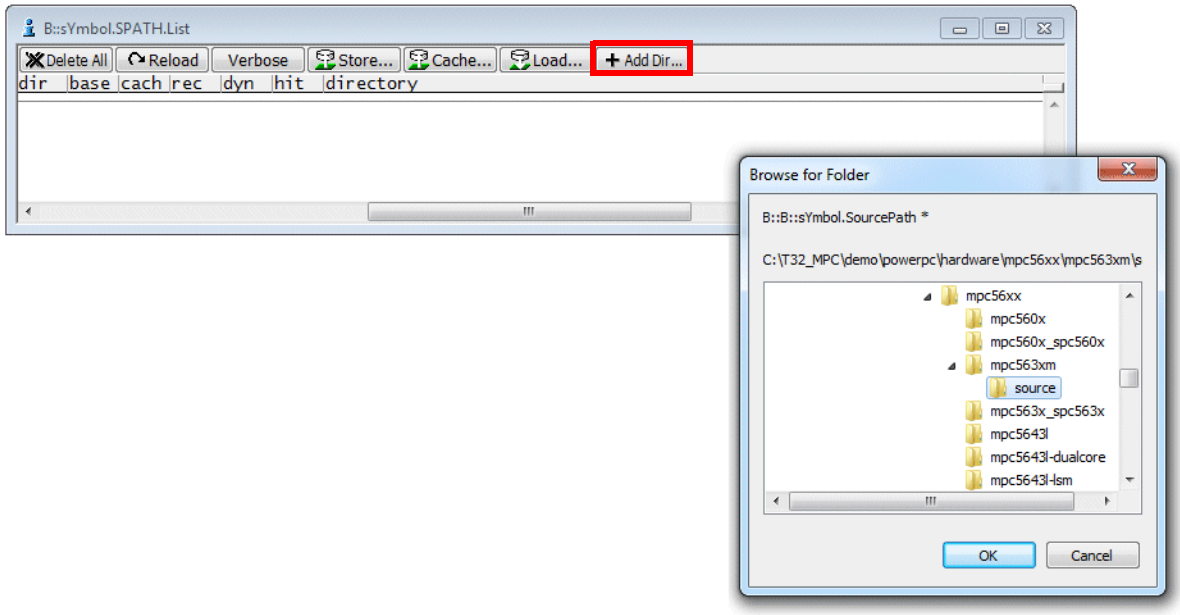
TRACE32 displays a warning when a required source file was not found and the source listing displays hatched areas instead of the source code information. One way to solve this issue is to directly provide the correct path for the source file.

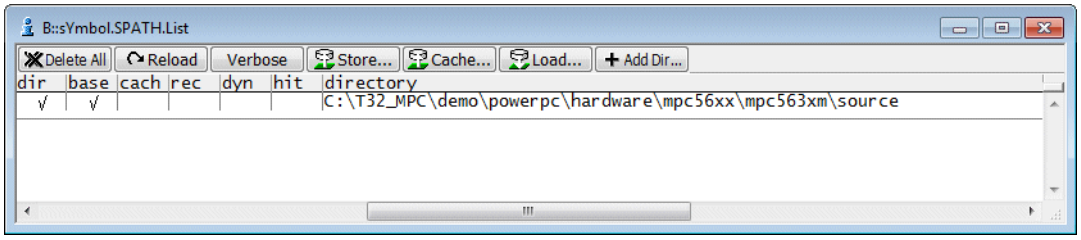
1. Open a **sYmbol.SPATH.List** window.





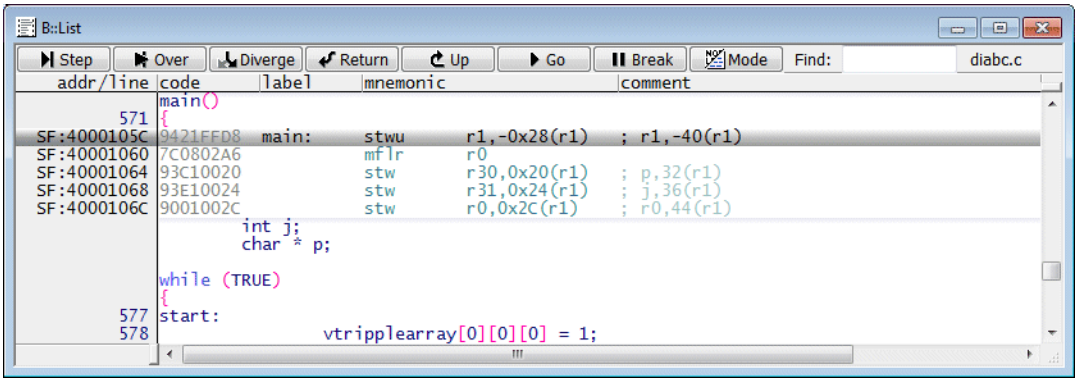
2. Use the **+AddDir …** button in the **sYmbol.SPATH.List** window to open a folder browser. Select the directory in which the missing source file is located.



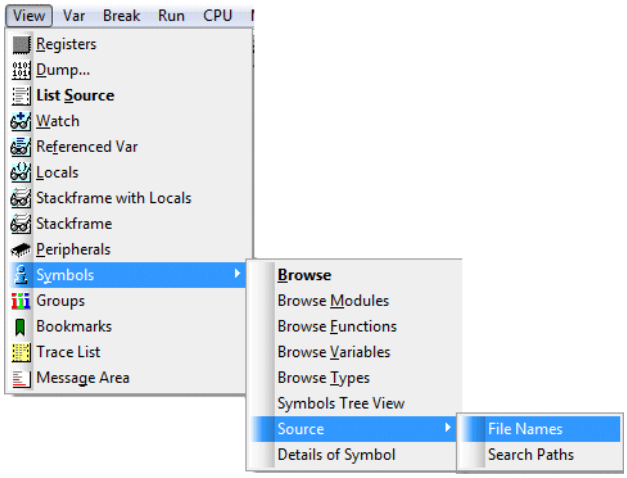


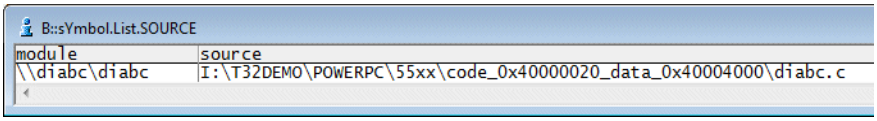
The **sYmbol.SPATH.List** window lists all provided directories.

As soon as the required source file was found, its source code is visible in the **Source Listing**.

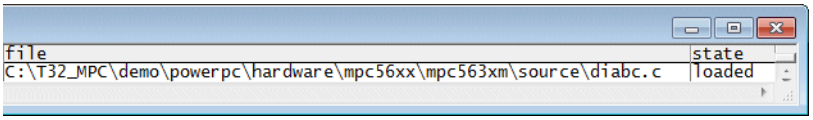


If you want to check if the correct source file was used, proceed as shown below:





The compile paths provided by the loaded program are listed in the **source** column

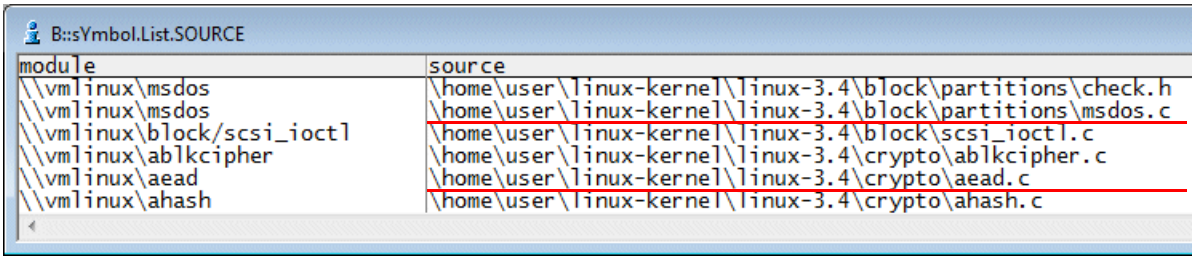


The path from which a source file was actually loaded is listed in the **file** column.

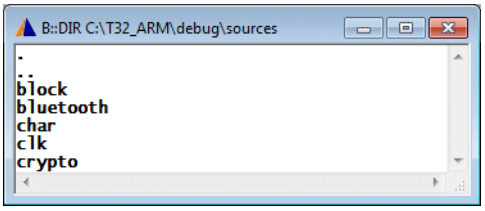
sYmbol.SourcePATH.List	List source file search information.
sYmbol.SourcePATH.SetDir *<directory>*	Define directory as direct search path.
sYmbol.List.SOURCE	Display source file details.

If you have a large project with a lot of subdirectories it is work-intensive to provide all source paths directly. An easier solution works as follows:

Path information from the compile environment



Source file directories in the current debug environment

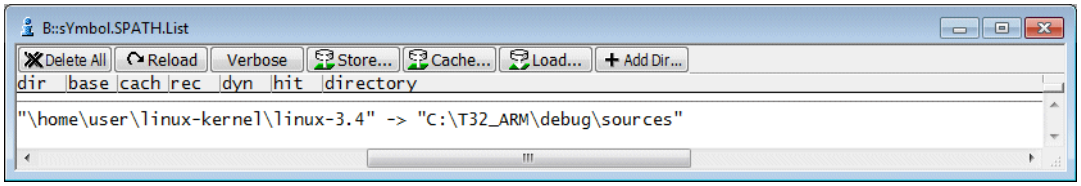


If we take a closer look e.g. to the files **msdos.c** and **aead.c**, we can see that the following command can solve the issue easily.

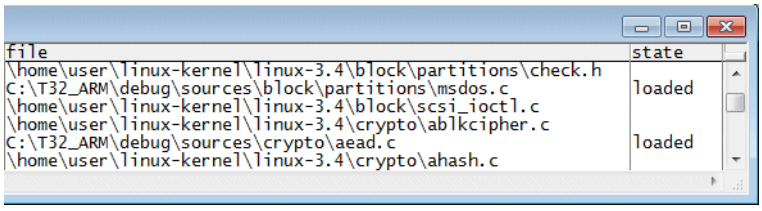
```
sYmbol.SourcePATH.Translate "\home\user\linux-kernel\linux-3.4" \
"C:\T32_ARM\debug\sources"
```

The invalid part of the source file paths (`"\home\user\linux-kernel\linux-3.4"`) is translated to the correct part (`"C:\T32_ARM\debug\sources"`).

The **sYmbol.SPATH.List** window shows this translation.



The source files can now be loaded from the correct location.



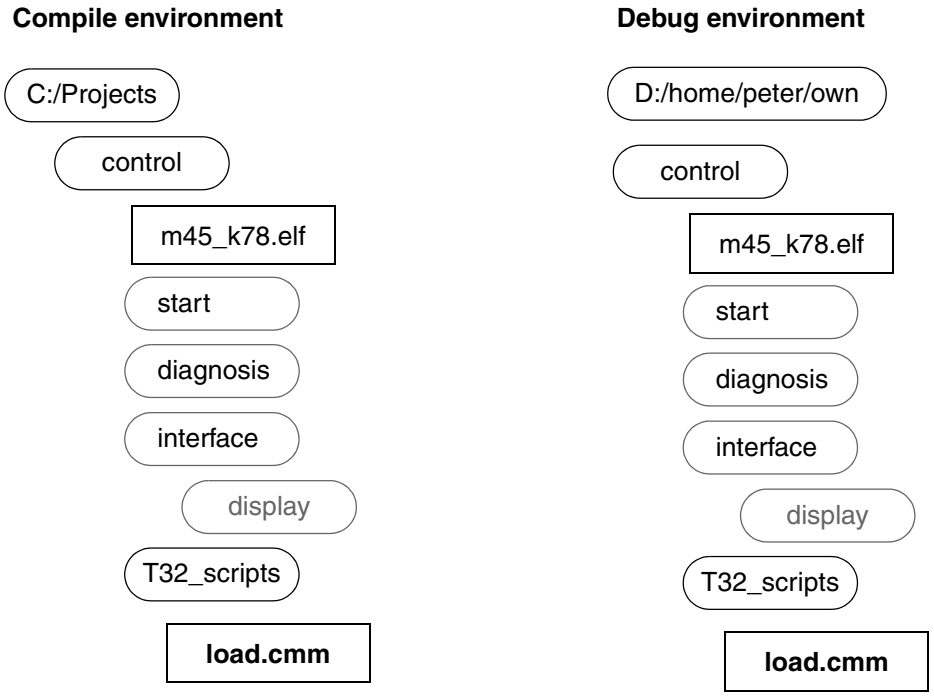
Translate *<invalid_part>* of source file paths to *<correct_part>*.

sYmbol.SourcePATH.Translate *<invalid_part>* *<correct_part>*

sYmbol.SourcePATH.List	List source file search information.
sYmbol.List.SOURCE	Display source file details.

Translating the compile paths to the debug paths is not flexible enough, if each user has its own debug environment. The following example shows a generic solution for a personalized debug paths.

For this generic solution it is required that the script that loads the program (here **load.cmm**) is part of the project, as shown in the example below.



The idea is now the following:

1. When the program is loaded, the start of the compile path including the project name (here: control) is stripped by the command:

   Data.LOAD.*<file_format>* *<file>* **/StripPART** *<project_name>*

2. Now the new personalized start of the debug path has to be provided.

   The presented solution takes advantage of the fact that TRACE32 includes shortcuts that represent directories and that these shortcut can be used as path prefixes. The shortcut needed for our solution is ~~~~ and it represents the directory where the currently running script is located.

   ~~~~/.. represents exactly the start of all source paths (including the project name) in the debug environment. This new start for all source paths can be specified by the following command.

   sYmbol.SourcePATH.SetBaseDir ~~~~/..
   ~~~~

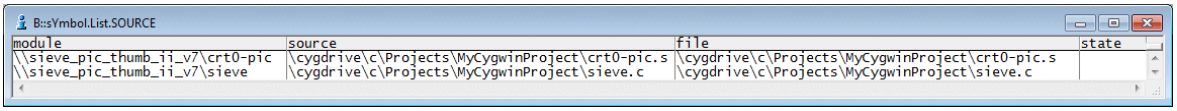
The script **load.cmm** has to include the following:

```
…

; cut the following from the source paths:
; C:/Projects/control
Data.LOAD.Elf ~~~~/../m45_k78.elf /StripPART "control"

; specify new base directory (here d:/home/peter/own/control)
; for relative paths
; sYmbol.SourcePATH.SetBaseDir ~~~~/..

…
```

Example 4: Convert cygdrive paths to window paths

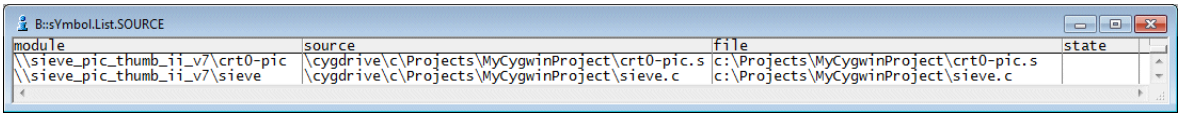
If the source files are compiled in a cygdrive enviroment, cygdrive paths are provided by the loaded program.

```
Data.LOAD.Elf sieve_pic_thumb_ii_v7.elf
```



The option **/CYGDRIVE** advises TRACE32 to convert the cygdrive paths to Windows paths.

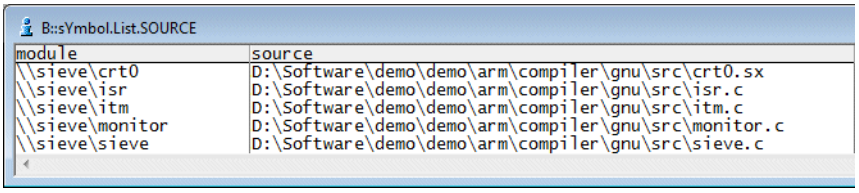
```
Data.LOAD.Elf sieve_pic_thumb_ii_v7.elf /CYGDRIVE
```



Data.LOAD.Elf *<file>* **/CYGDRIVE**	Load .elf file, convert cygdrive paths to Window paths.
sYmbol.List.SOURCE	Display source file details.

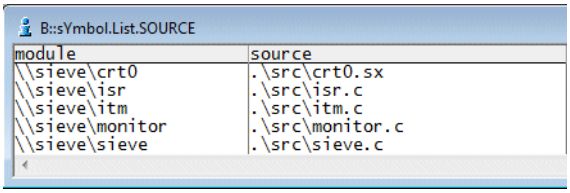
If source files are compiled with relative paths, the resulting .elf file contains both, all *<relative_path>* as well as the *<compile_directory>*. By default TRACE32 performs as follows:

```
// Load Elf file, construct source file paths by
// combining <compile_directory><relative_path>
Data.LOAD.Elf C:/T32_ARM/demo/arm/compiler/gnu/sieve.elf
```

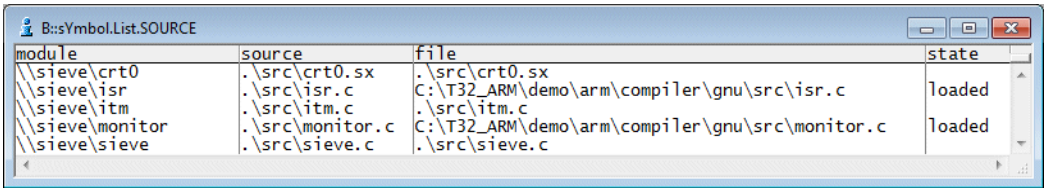


The option **/RelPATH** advises TRACE32 to not use the *<compile_directory>*.

```
// Load Elf file, provide only all <relative_path> for source files
Data.LOAD.Elf C:/T32_ARM/demo/arm/compiler/gnu/sieve.elf /RelPATH
```



TRACE32 is trying to load the source files now relatively to the location of the ELF file.



If this does not work, you can provide the start of the source paths directly:

```
sYmbol.SourcePATH.SetBaseDir C:\T32_ARM\demo\arm\compiler\gnu
```

Data.LOAD.Elf *<file>* **/RelPATH**	Load .elf file with relative paths only.
sYmbol.SourcePATH.SetBaseDir *<base_directory>*	Provide start of source paths directly.
sYmbol.List.SOURCE	Display source file details.

Loader Options for the Virtual Memory

TRACE32 provides a so-called virtual memory on the host. With the following options the code is loaded into this virtual memory.

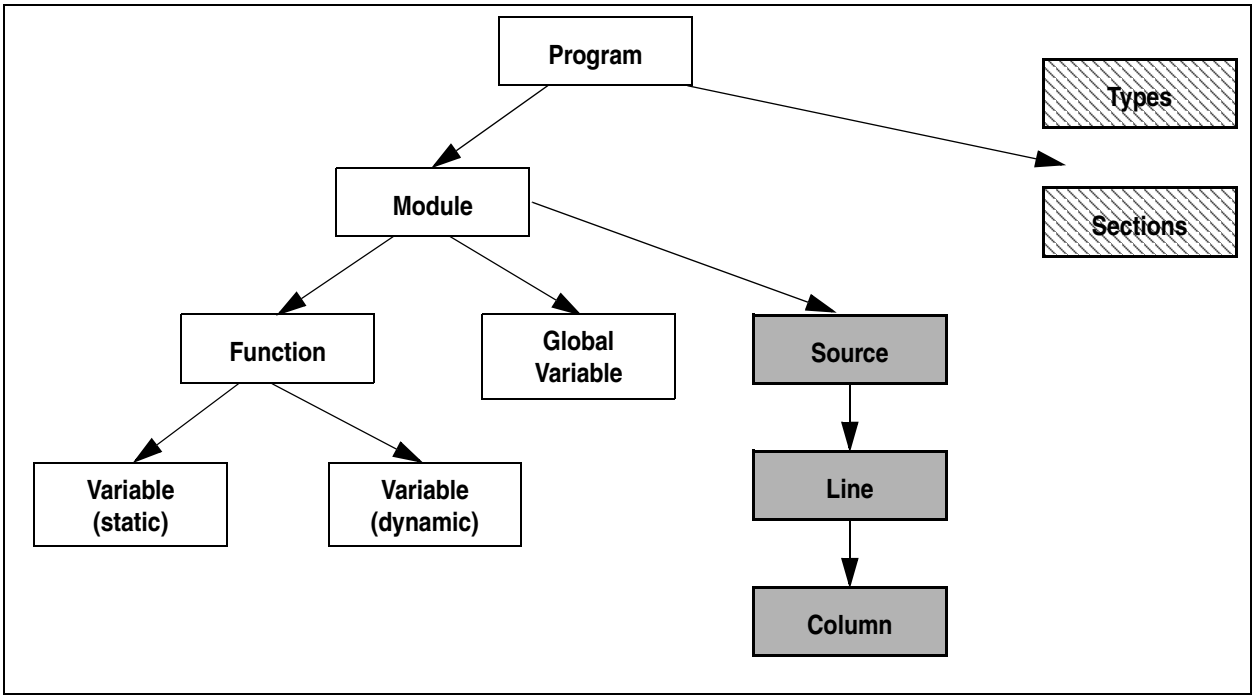
VM	Load the code/data into the virtual memory.
PlusVM	Load the code/data into the target and into the virtual memory.

```
Data.LOAD.COFF arm.abs /VM          ; load code/data from <file> into the
                                    ; virtual memory

Data.LOAD.COFF arm.abs /PlusVM      ; load code data from <file> into the
                                    ; target memory and into the virtual
                                    ; memory
```

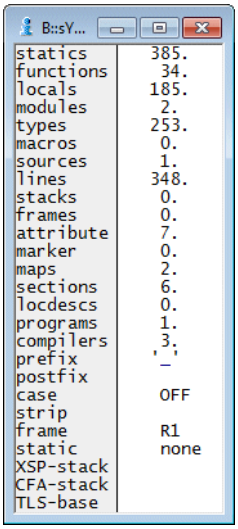
A detailed description of the use cases for the TRACE32 virtual memory are given in **"TRACE32 Virtual Memory"** in TRACE32 Glossary, page 54 (glossary.pdf).

Structure of the Internal Symbol Database

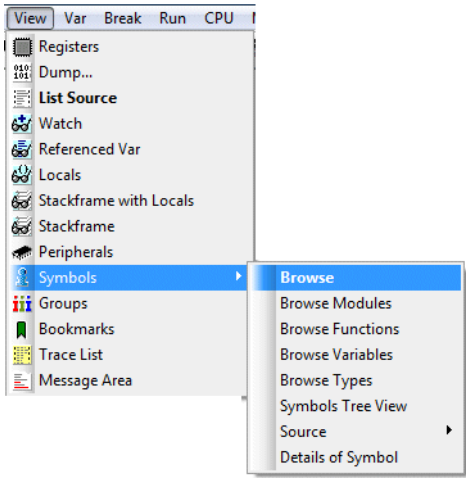
The symbol and debug information loaded with the **Data.LOAD** command is organized in an internal symbol database by TRACE32.

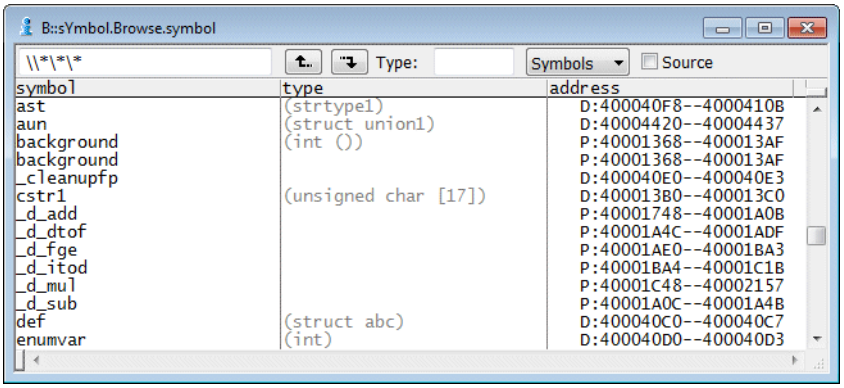


sYmbol.STATE Display general information about symbol database



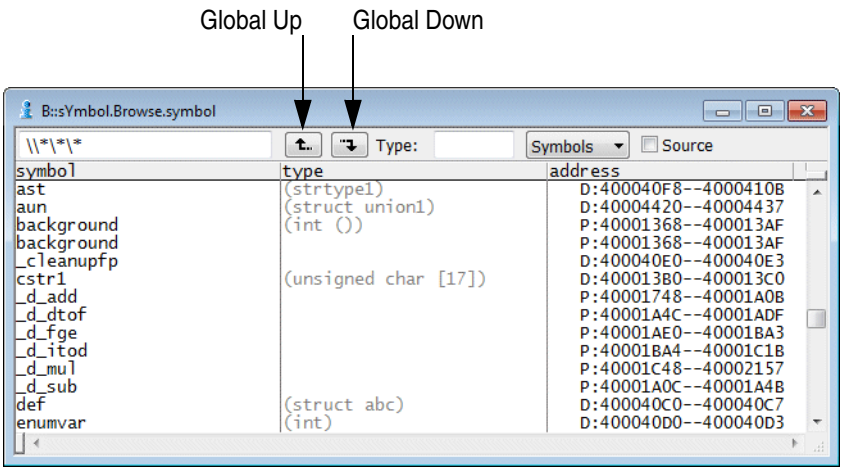
Symbol Browser





sYmbol.Browse [*<name_pattern>* [*<type_pattern>*]] Browse symbol information

```
sYmbol.Browse a*

sYmbol.Browse a* struct*

sYmbol.Browse * *struct*
```

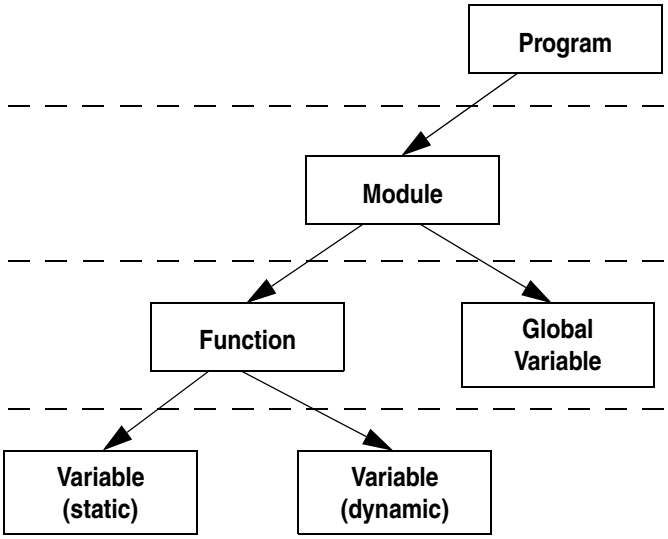


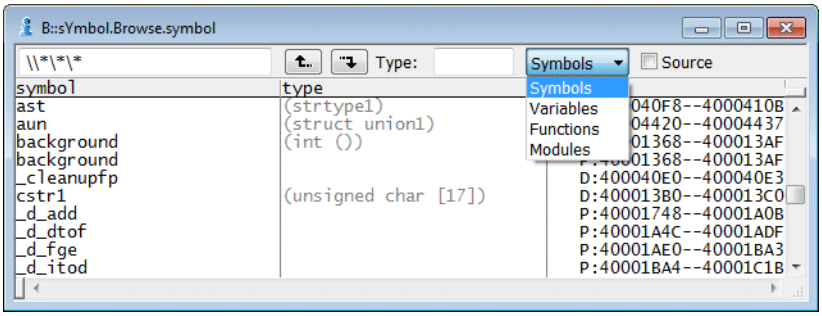
***** (all programs)

****** (all modules)

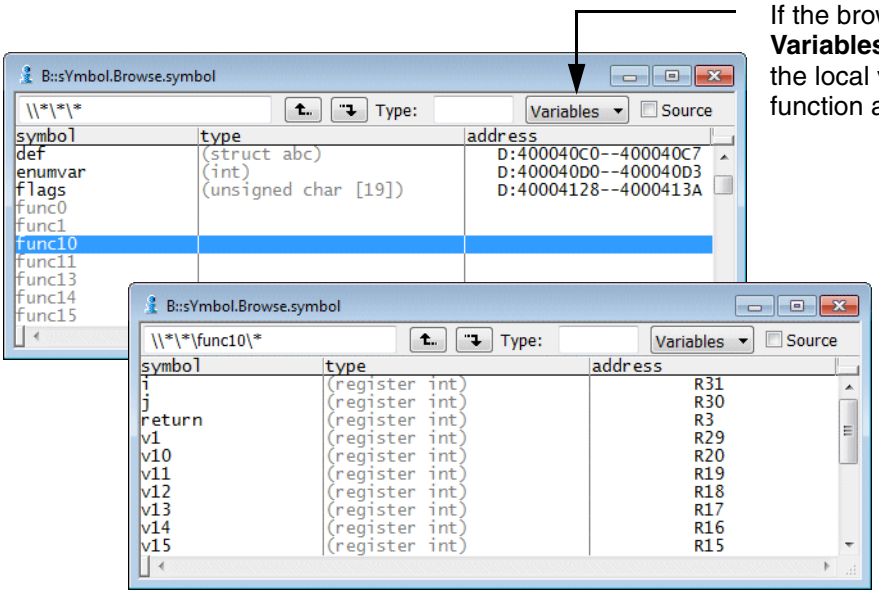
******* (all functions, all global variables)

******** (all local variables)

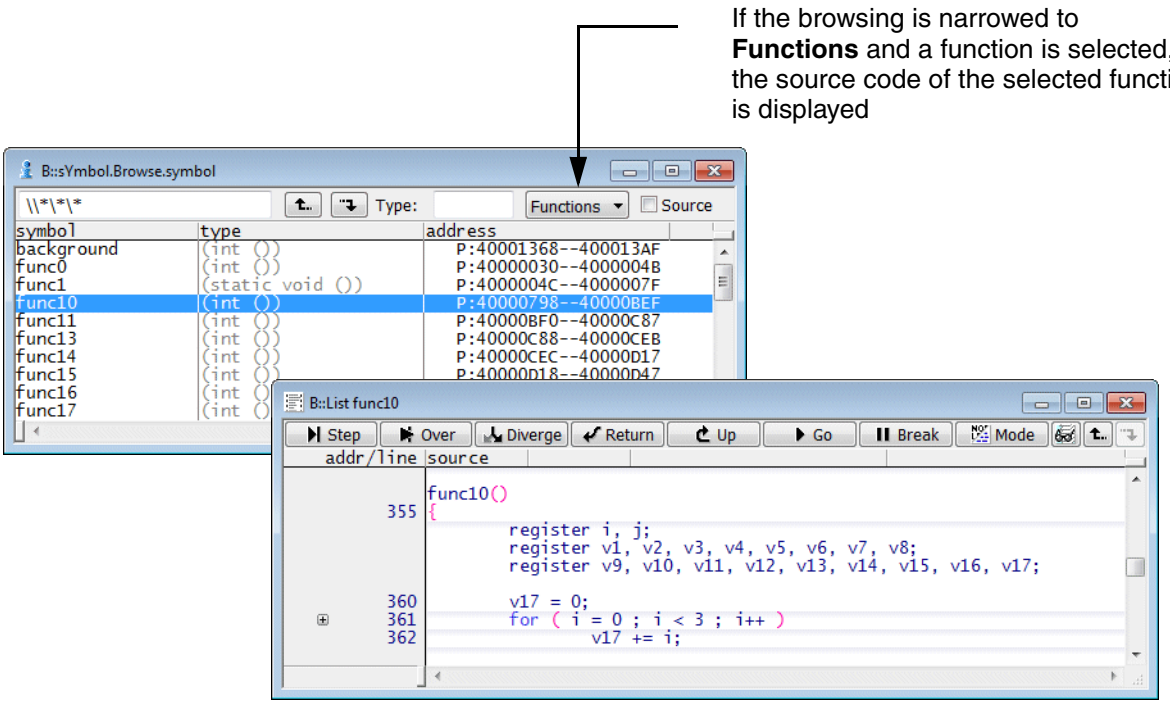


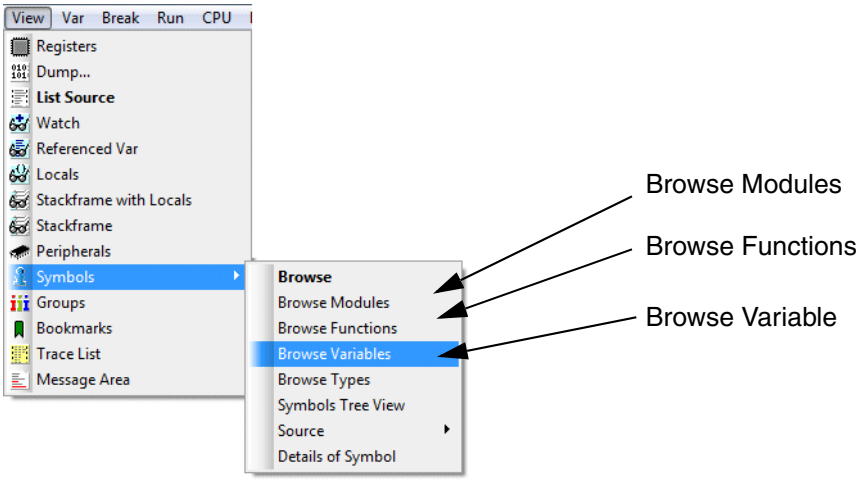


Select display type	
Symbols	Display all symbols
Variables	Display all variables
Functions	Display all functions
Modules	Display all modules



If the browsing is narrowed to **Variables** and a function is selected, the local variables of the selected function are displayed

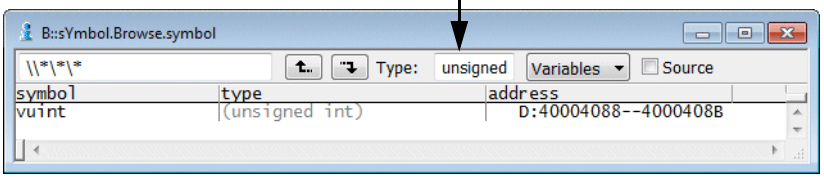




sYmbol.Browse	Browse symbol information
sYmbol.Browse.Function	Browse functions
sYmbol.Browse.Var	Browse variables
sYmbol.Browse.Modules	Browse modules

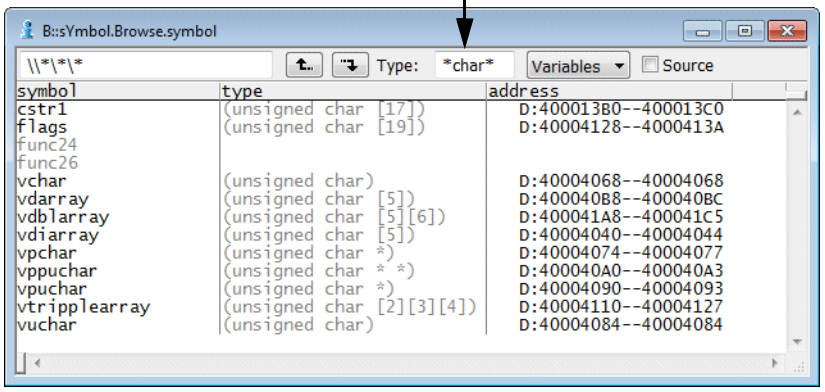
Browsing for a Specific Type

Display all variables of the type **unsigned int**



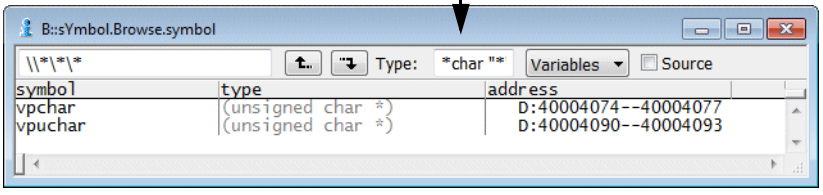

```
sYmbol.Browse.Var * unsigned int
```

Display all variables where the type name contains the keyword char **(*char*)**



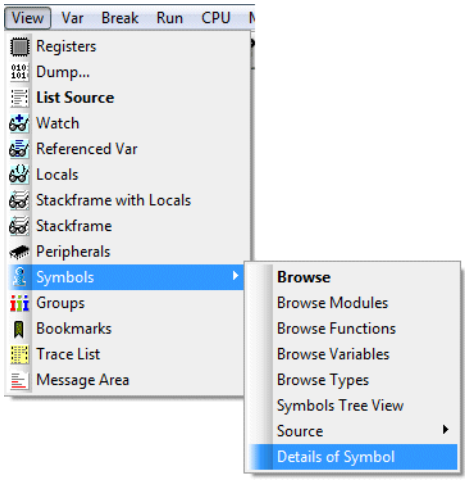

```
sYmbol.Browse.Var * *char*
```

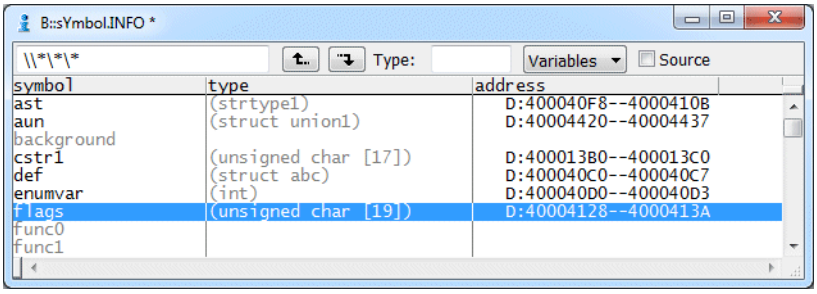
Display all variables of the type pointer to char **(*char "*")**

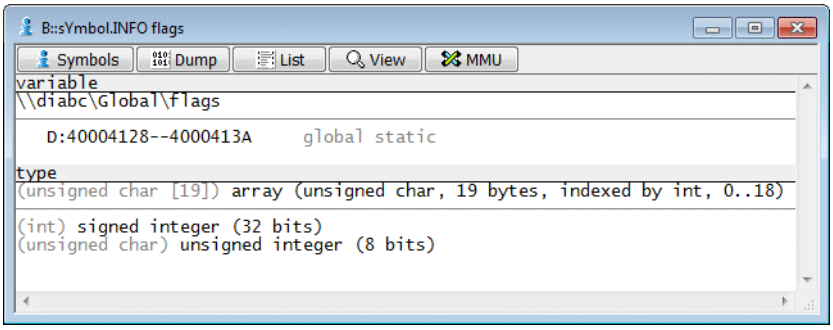


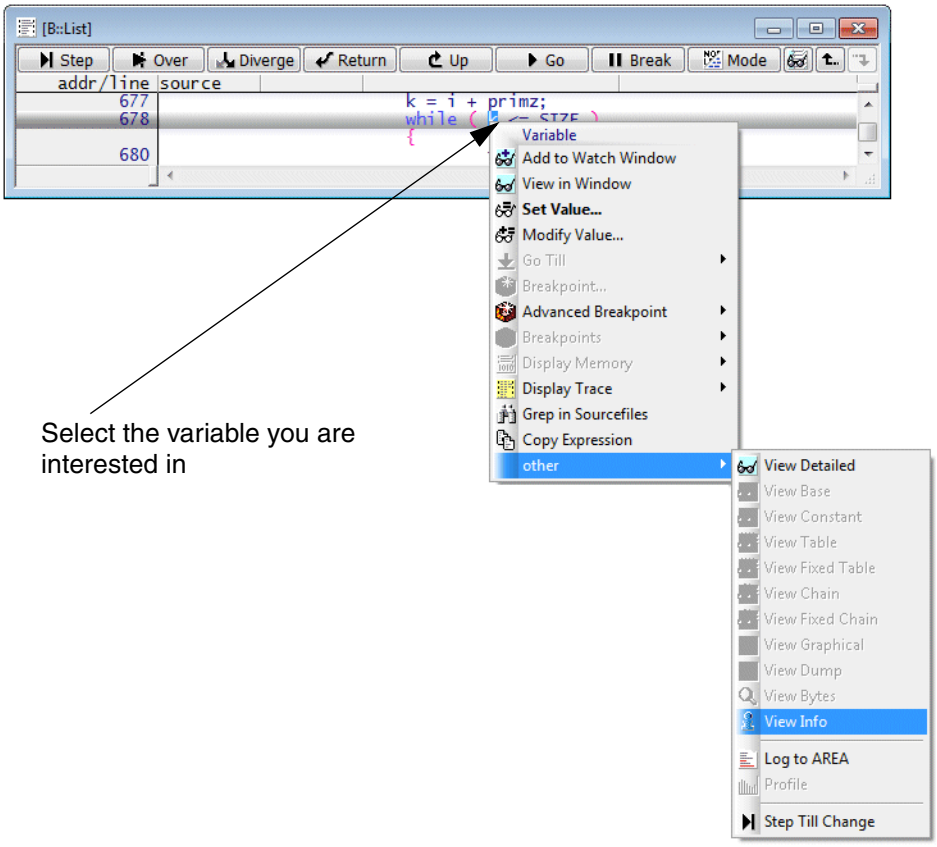

```
sYmbol.Browse.Var * *char "*"
```

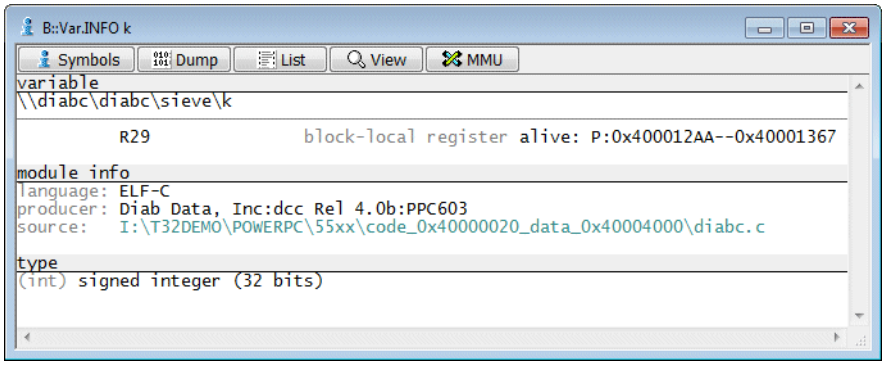
Details about a Selected Symbol







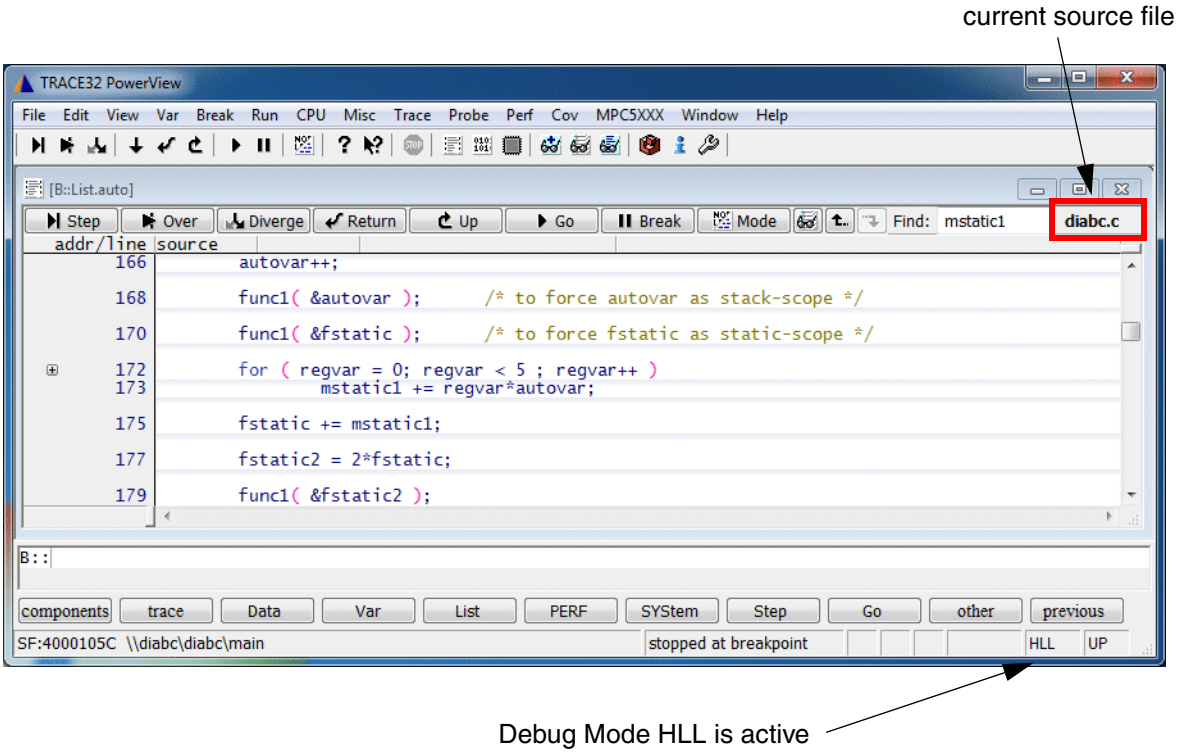




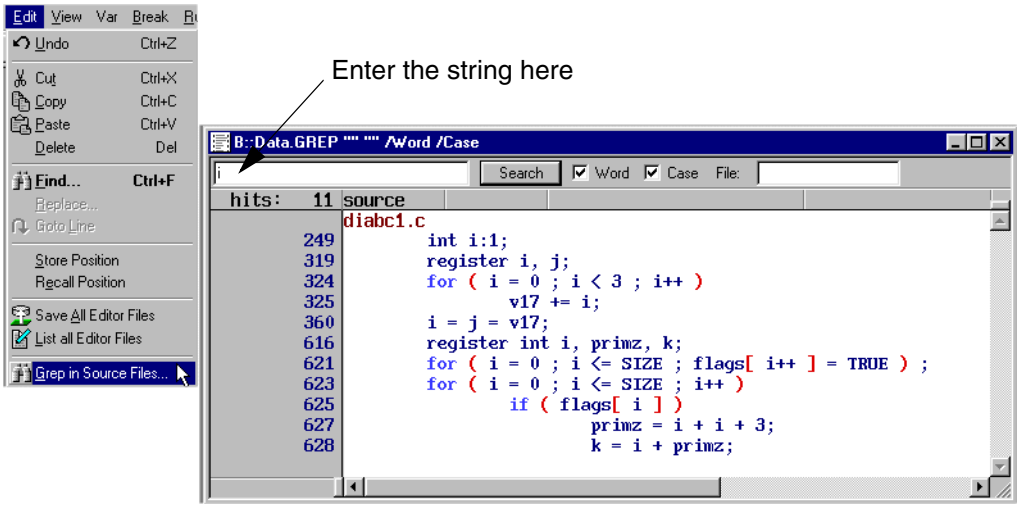
sYmbol.INFO	Display symbolic address, location, scope and layout of a symbol
Var.INFO	Display symbolic address, location, scope and layout of a variable or function

Search a String in the Current Source File

If debug mode HLL is active, the entered string is searched in the current source file.



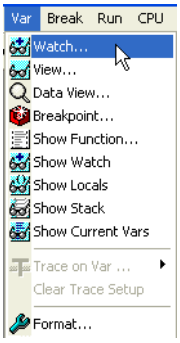
Search a String in all Source Files



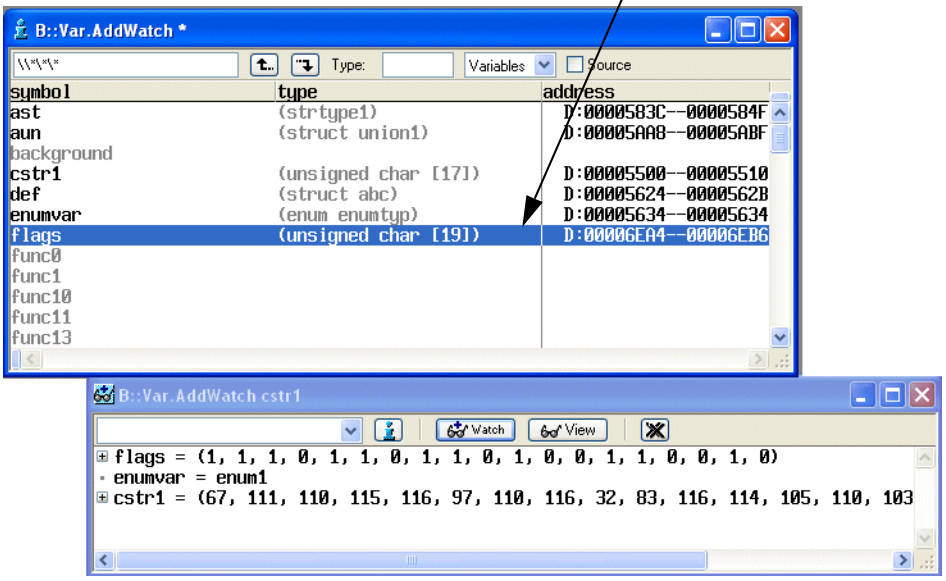
TRACE32 searches in all source files for the defined string.

Watch Window

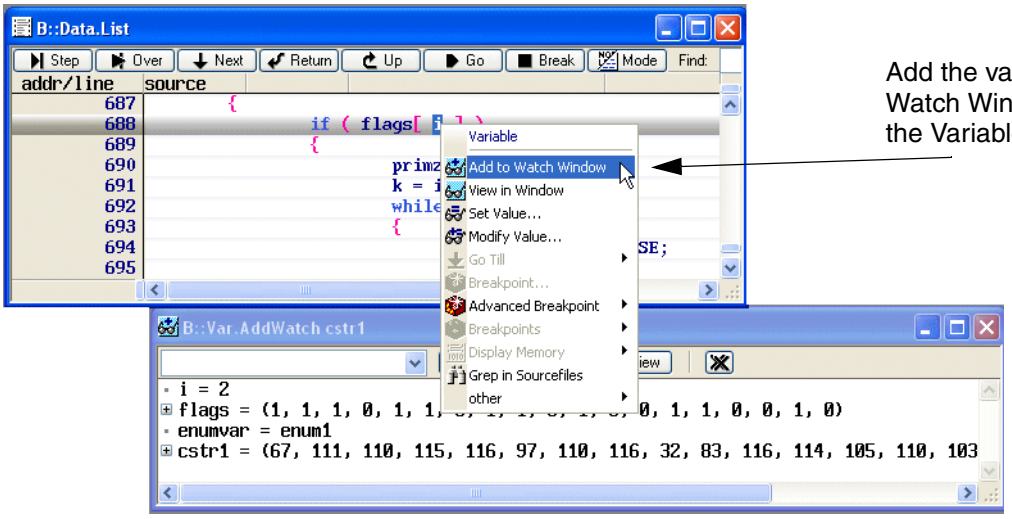
Adds the selected variable to the top of the ***Variable Watch*** window. If no Watch Window exists, a new Watch Window is created.

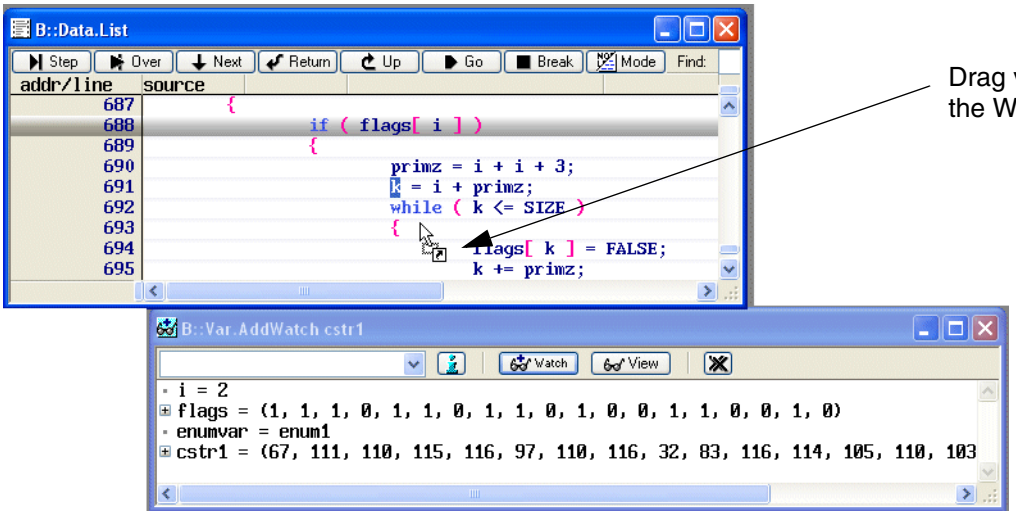


The selected variable is added to the top of the Variable Watch window

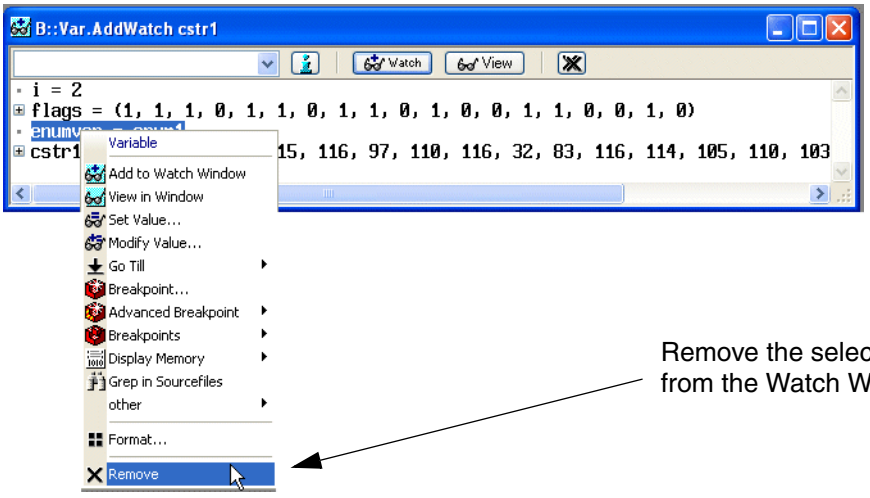


Add the variable to the Watch Window by using the Variable pull down





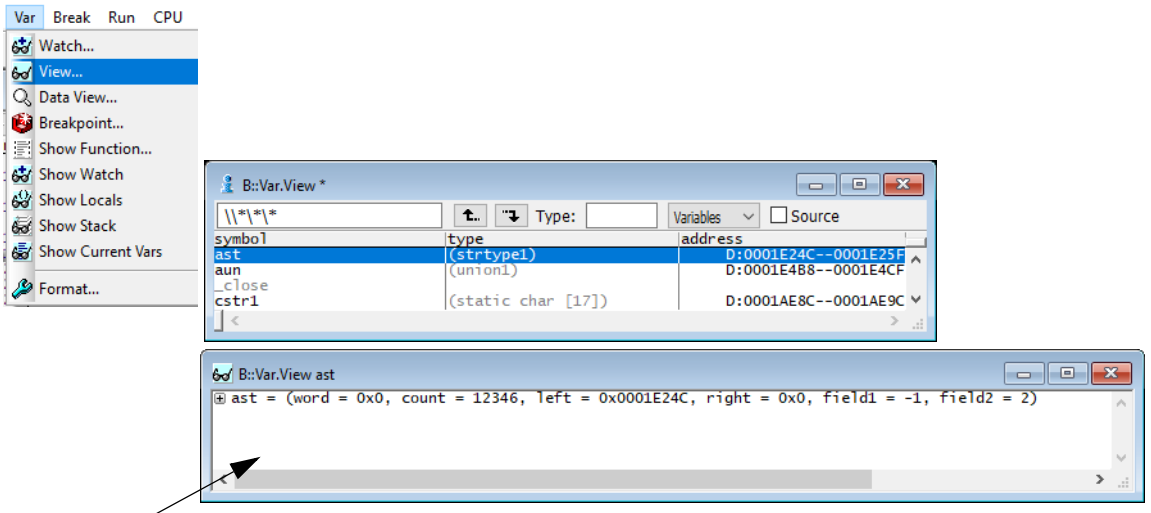
Drag variable to the the Watch Window



Remove the selected variable from the Watch Window

Var.Watch [%<*format*>] [<*variable*>]	Open a watch window and display variable
Var.AddWatch [%<*format*>] <*variable*>	Add variable to watch window

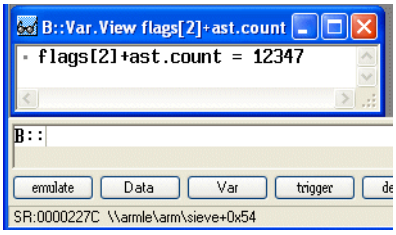
Opens a new ***Variable View*** window for the selected variable.



A new ***Variable View*** Window is opened
to display the selected variable

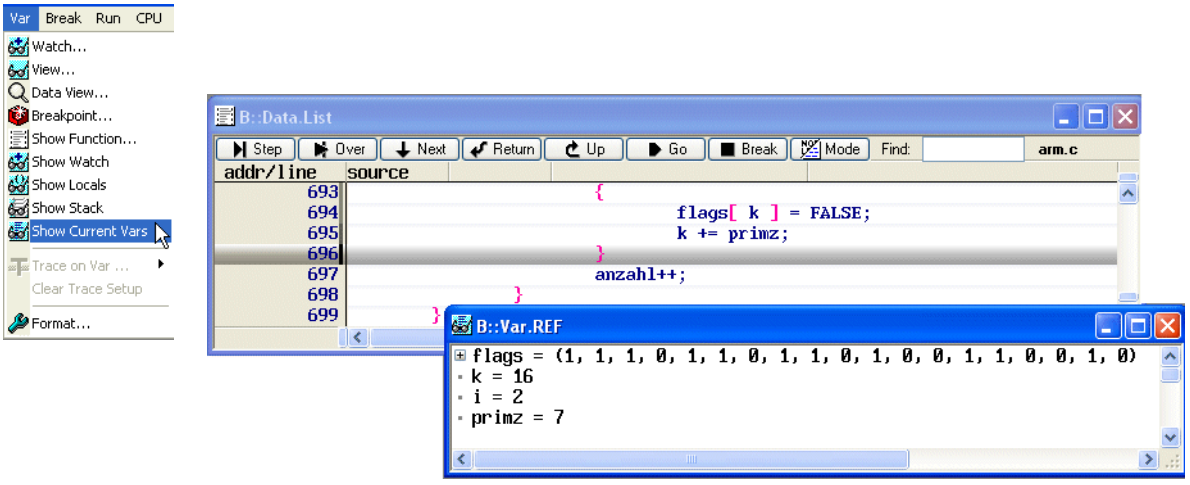
Var.View [%<*format*>] <*variable*> Display variable in a separate window

- If a formula is entered, it is interpreted and the result is displayed.



Referenced Variables

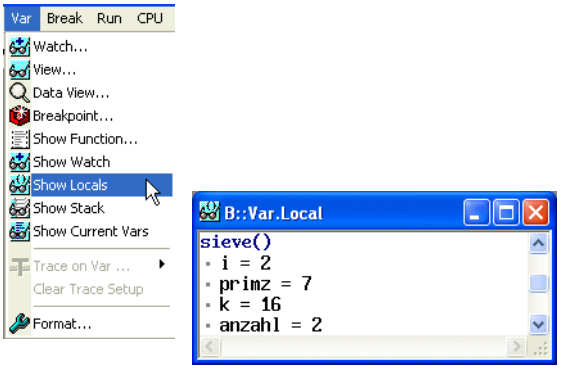
Opens a ***Var.REF*** window. The variables referenced by the current source line are automatically added to this window.



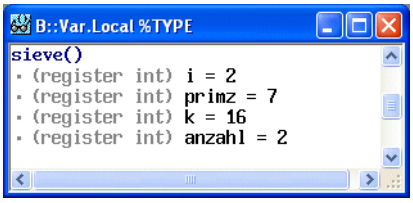
Var.Ref [%<*format*>]	Display the variables referenced by the current code line

Local Variables

Open a window to display the local variables of the current function.

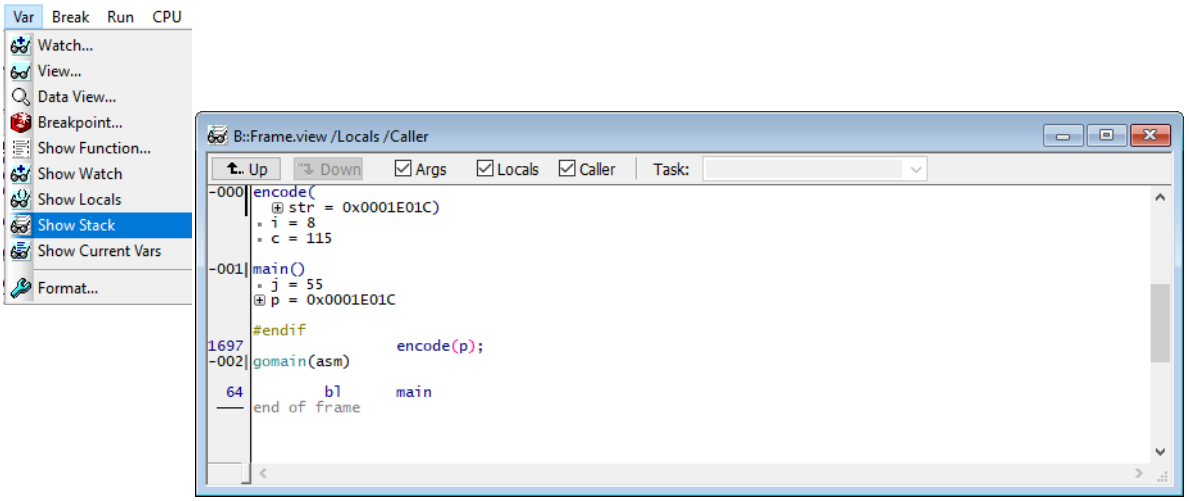


Var.Local [%<*format*>] Display local variables



Stack Frame

Display a “stack trace” to show the functions´ nesting.



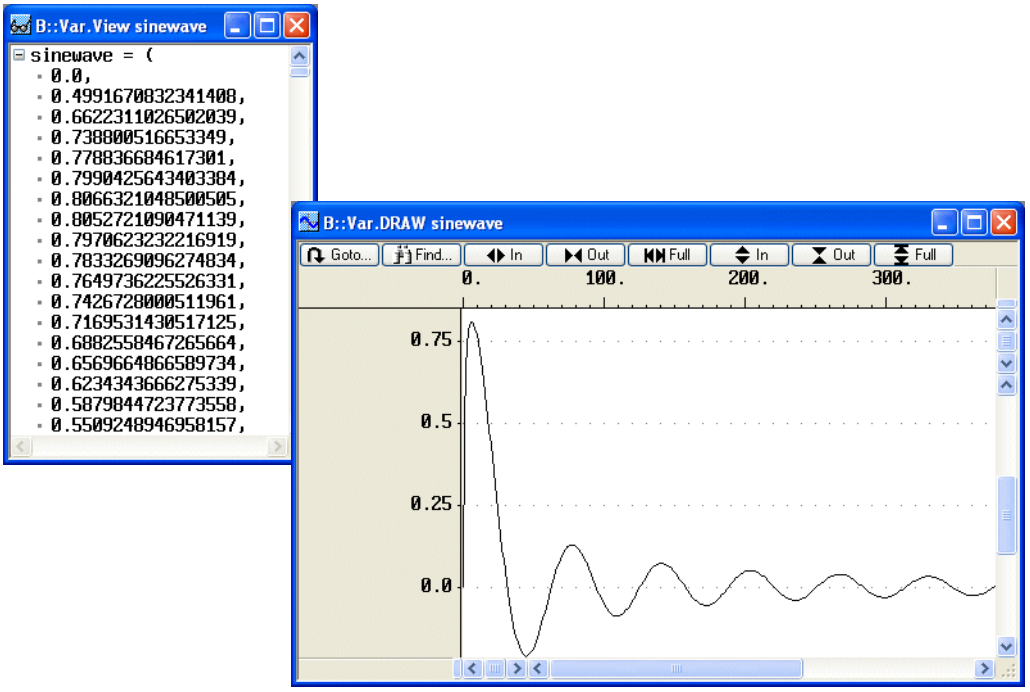
Args	Display the arguments.
Local	Display the local variables.
Caller	Display of the high level language block from which the function was called.

Frame.view [%<*format*>] [/option] Display a ´stack trace´

Graphical Display

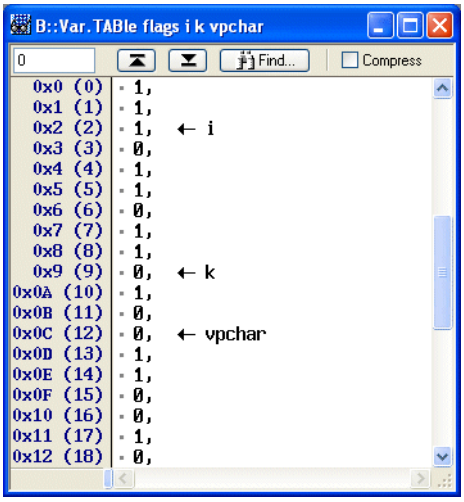
Var.DRAW [%<*format*>] <*array*> Display the contents of an array graphically

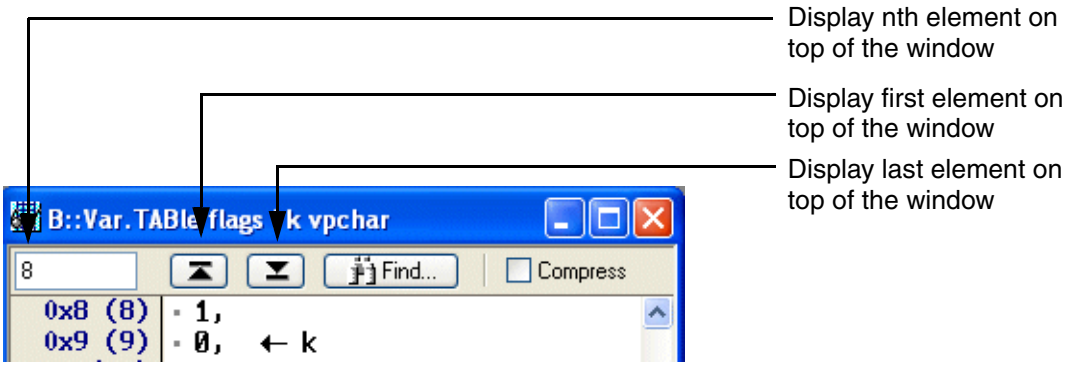
```
Var.DRAW sinewave
```

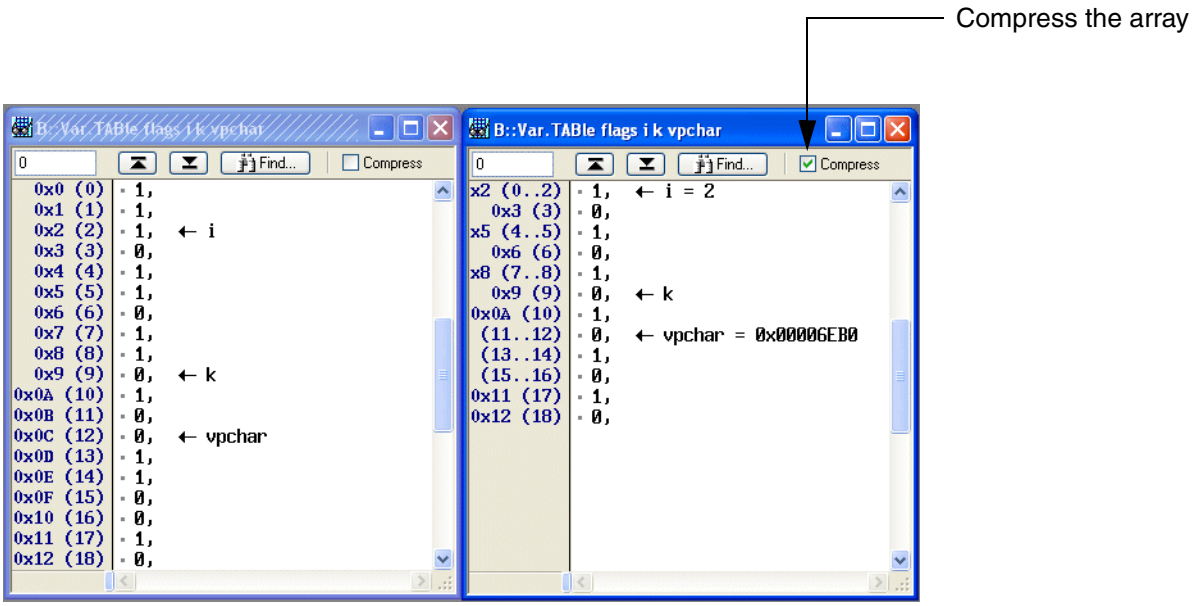


Var.TABle [%<*format*>] <*array*> <*index*> […]	Display an array together with indices and pointers

```
Var.TABle flags i k vpchar                    i and k are indices,
                                              vpchar is a pointer
```

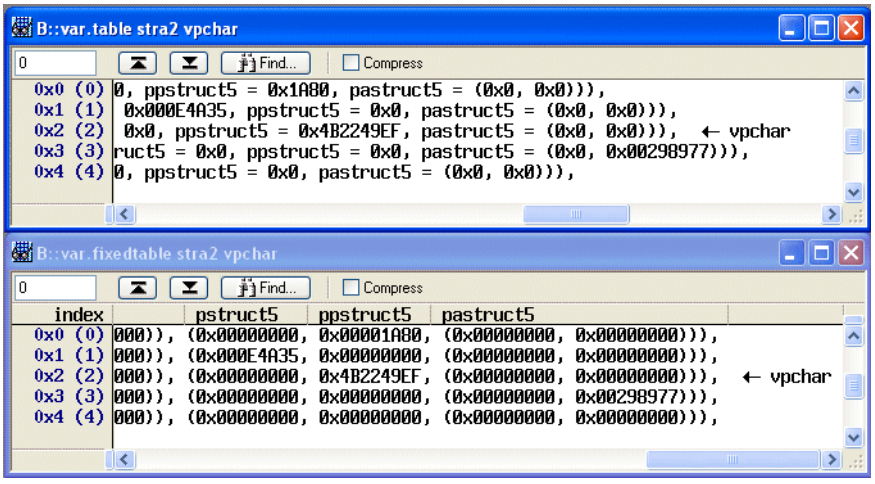






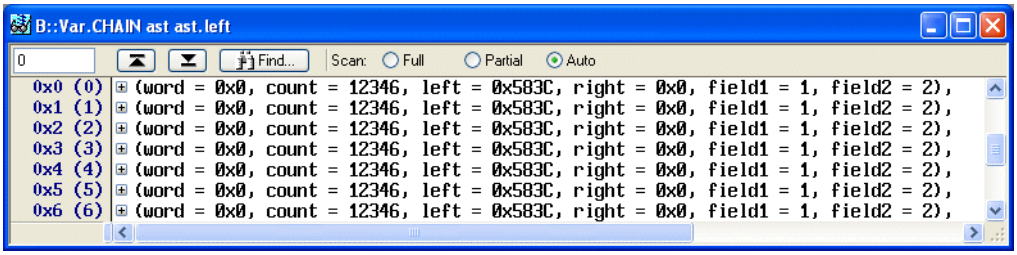
Var.FixedTABle [%<*format*>] <*array*> <*index*> […] — Display an array together with indices and pointers in a fixed format

```
Var.FixedTABle stra2 vpchar
```



Var.CHAIN [%<*format*>] <*first*> <*next*> […] Display a linked list

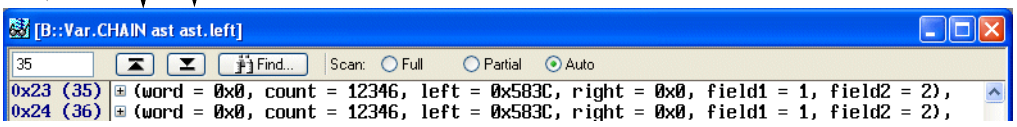
```
Var.CHAIN ast ast.left                  ast is the first element of the
                                        linked list,
                                        ast.left provides the pointer to
                                        the next element
```



Display the nth element on top of the window

Display the first element on top of the window

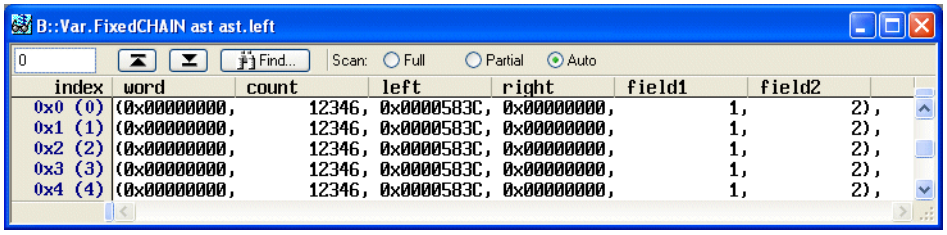
Display the last element on top of the window



Scan Modes	The linked list is permanently scanned to keep it up to date. This may reduce the performance of the TRACE32 user interface. 3 different scan modes are supported
Full	The linked list is scanned completely. This may reduce the performance of the TRACE32 user interface considerably.
Partial	The linked list is only scanned from the record at the top of the screen. The influence on the performance of the TRACE32 user interface is very small.
Auto	This mode provides a compromise between an up to date linked list and a fast TRACE32 user interface. For a specific time (20-50 ms) the list is updated and for the same time user inputs are served. The number beside the Auto button is the number of the last updated record.

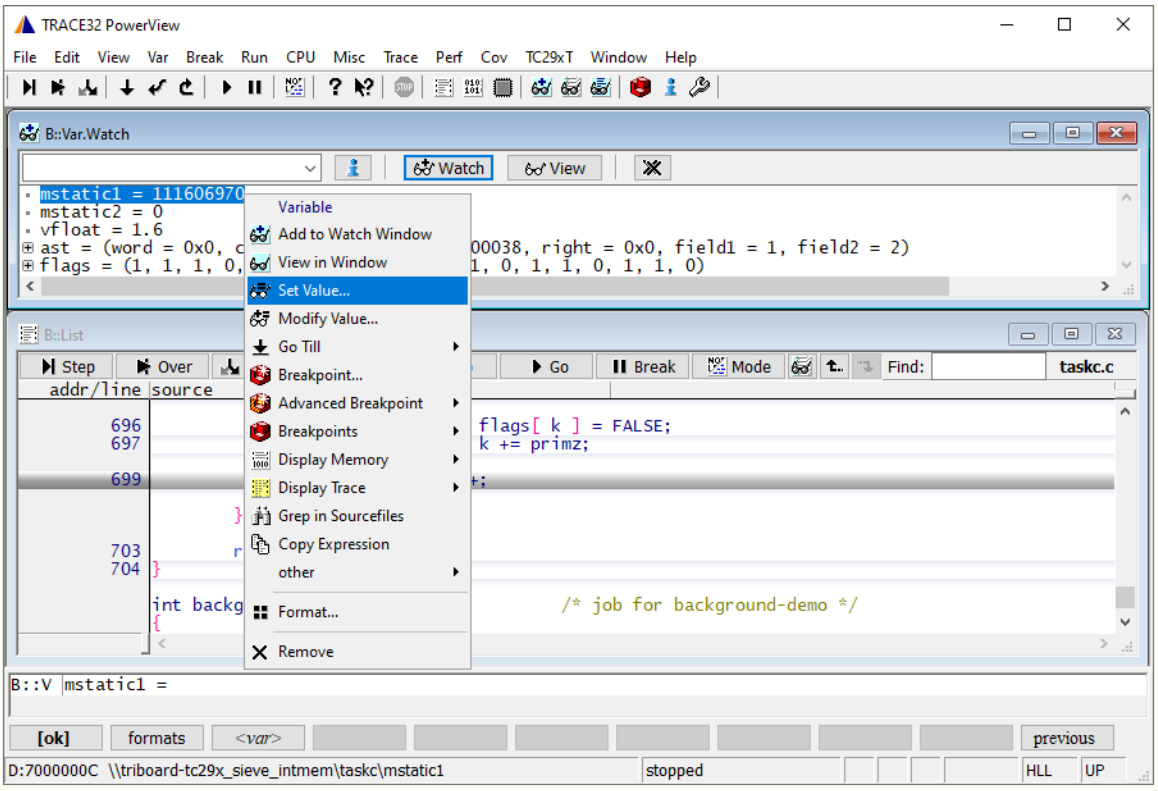
Var.FixedCHAIN [%<*format*>] <*first*> <*next*> […] Display a linked list in a fixed format

```
Var.FixedCHAIN ast ast.left
```



Change a Variable Value

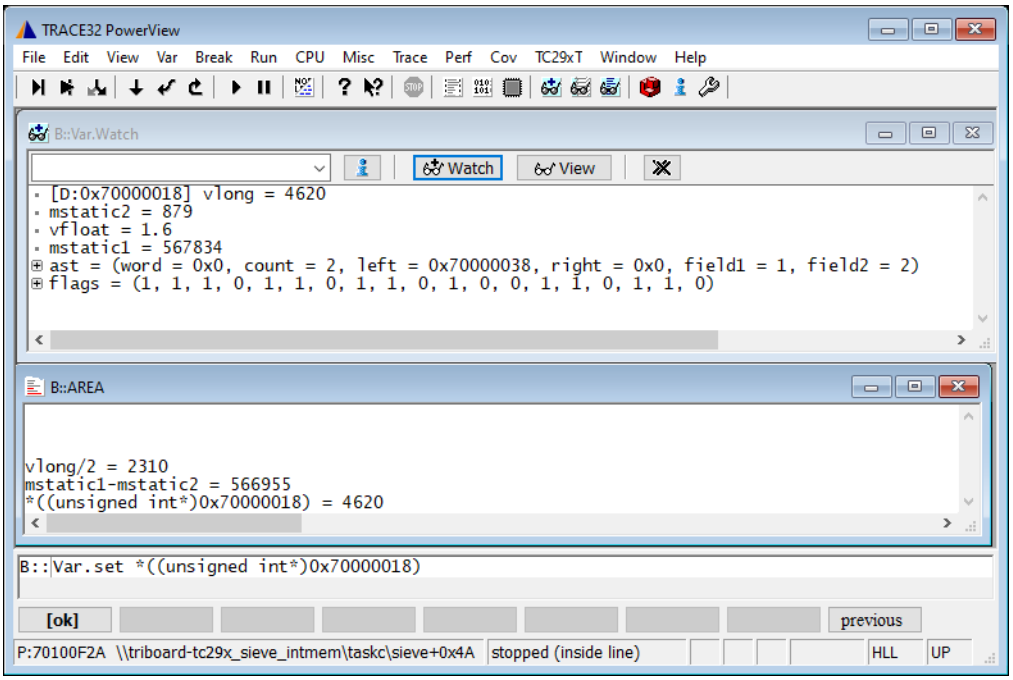
To change the content of a variable, you can use the **Set Value…** command from the variable pull-down or simply double-click on the variable. In either case, the command **Var.set** appears in the command line together with the name of the variable.



Here are a few examples for the use of the **Var.set** command:

```
Var.set mstatic1 = 111609970          ; assign a decimal value to the
                                      ; variable mstatic1

Var.set mstatic1 = k+3                ; add 3 to the content of the
                                      ; variable k and assign the result
                                      ; to the variable mstatic1

Var.set mstatic1 = k+i                ; add the contents of the variables
                                      ; k and i and assign the result
                                      ; to the variable mstatic1

Var.set flags[3] = 1                  ; assign decimal value 1 to the 4th
                                      ; element of the array flags

Var.set ++k                           ; increment the content of the
                                      ; variable k
```

The command **Var.set** can also be used to evaluate a high-level language expression. The expression and its result are then displayed in the message area.



At this point it should be pointed out again that the **Var** command group uses as parameter syntax the syntax of the used high-level language (usually C/C++). While all other TRACE32 commands must use the TRACE32 parameter syntax, which is based on C but has some special features and handles debug symbols more like a compiler's linker does. The following table illustrates this:

	High-level expression	**TRACE32 expression**
Value of C/C++ Variable myvar	myvar	Var.VALUE(myvar)
Address of C/C++ Variable myvar	&myvar	myvar
Size of C/C++ Variable myvar	sizeof(myvar)	Var.SIZEOF(myvar)
Value of 32-bit word at address 0x2000	*((unsigned int*)0x2000)	Data.Long(D:0x2000)
Decimal constant	42	42.
5th element of array myarray	myarray[5]	Var.VALUE(myarray[5])
Data of element val in struct mystruct	mystruct.val	Var.VALUE(mystruct.val)

Value of core register R7	\Register(R7)	Register(R7)
String containing host OS	\VERSION_ENVironment(OS)	VERSION.ENVironment(OS)
Value on APB bus address 0x4000	\Data_Long((APB:0x4000))	Data.Long(APB:0x4000)

Here are some examples:

```
Var.set mstatic2 = mstatic1

Data.Set mstatic2 %Long Var.VALUE(mstatic1)
```

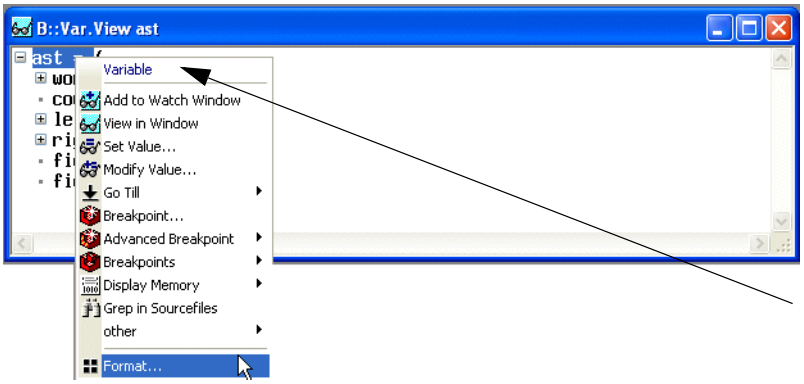
```
Var.set mstatic2=*((unsigned int*)0x7000004C)

Data.Set mstatic2 %Long Data.Long(D:0x7000004C)
```

```
Var.set mstatic2=\Register(D8)

Data.Set mstatic2 %Long Register(D8)
```

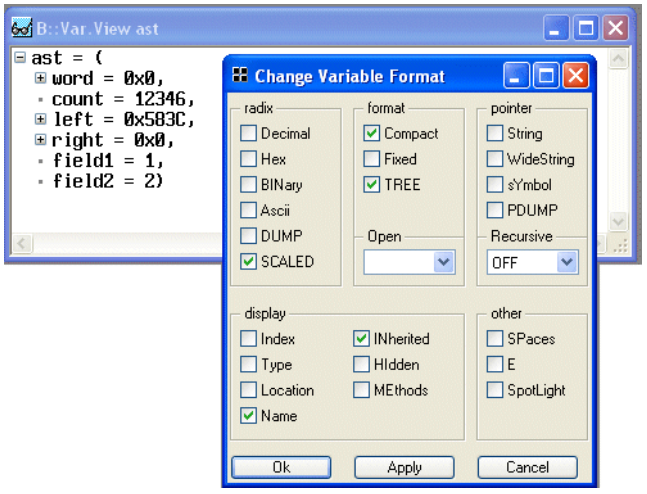
```
Var.Set mstatic2++

Data.Set mstatic2 %Long Var.VALUE(mstatic2)+1
```

```
; result as a decimal number
Var.PRINT \Data_Long((D:0x7000000C))

; result as a hex number
PRINT Data.Long(D:0x7000000C)
```

Format a Variable using the Format Dialog Box



Select the variable and press the right mouse button to open the ***Change Variable Format*** dialog box

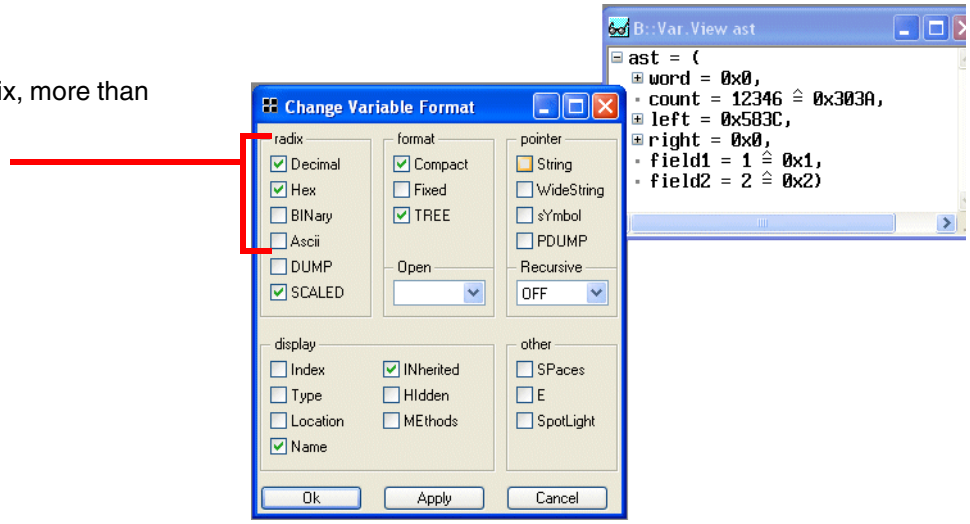


- **Numeric formats**

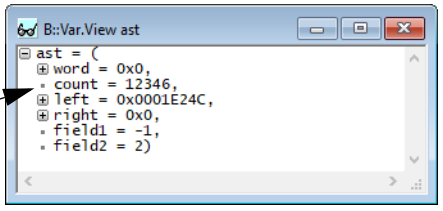
  By default integers are displayed in decimal format and pointers in hex format.

Radix	
Decimal	All numeric values are displayed in decimal format.
Hex	All numeric values are displayed in hex format.
BINary	All numeric values are displayed in binary format.
Ascii	All numeric values are displayed as ASCII characters.

Select the radix, more than one possible

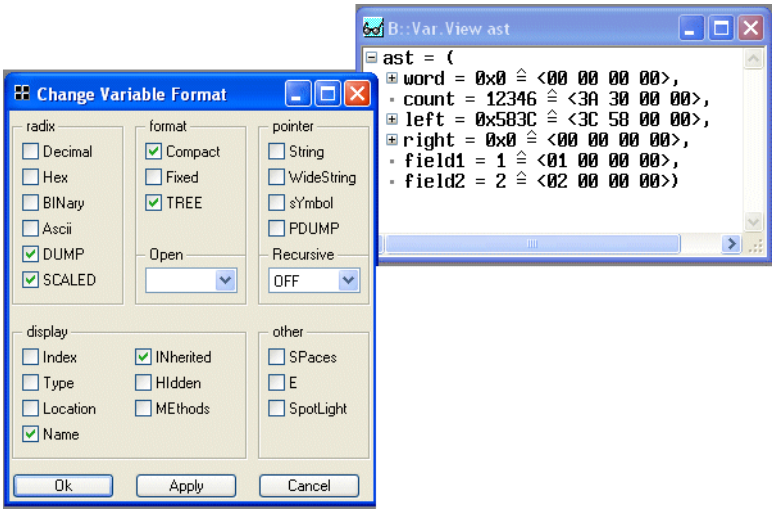


Click to the small dot on the left side of the variable to display a numeric value in different formats



- **Dump**

  Display the contents of the variable additionally as a hex dump.

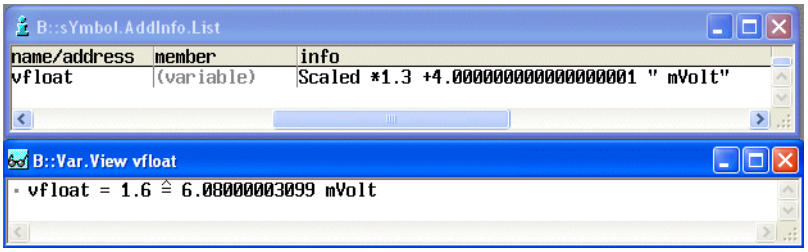


- **Scaled**

  Display the variable in the defined scaling

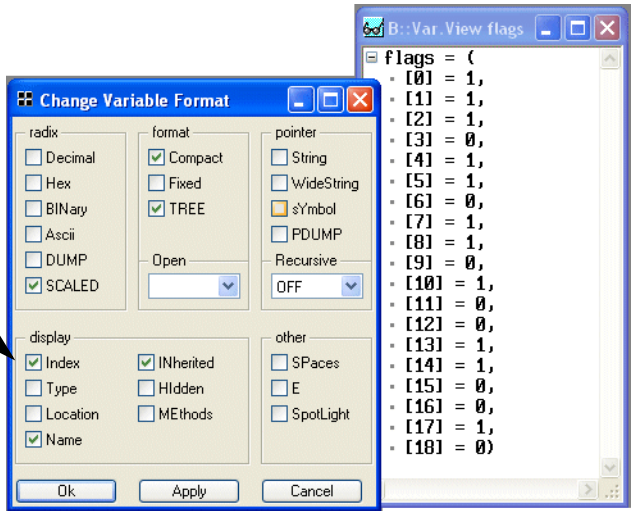
sYmbol.AddInfo.Var *<var>* **Scaled** *<multiplier> <offset> <format>*	Define a scaling for a variable
sYmbol.AddInfo.List	List all defined scalings
sYmbol.AddInfo.RESet	Reset list

```
sYmbol.AddInfo.Var vfloat Scaled 1.3 4 " mVolt"

sYmbol.AddInfo.List
```



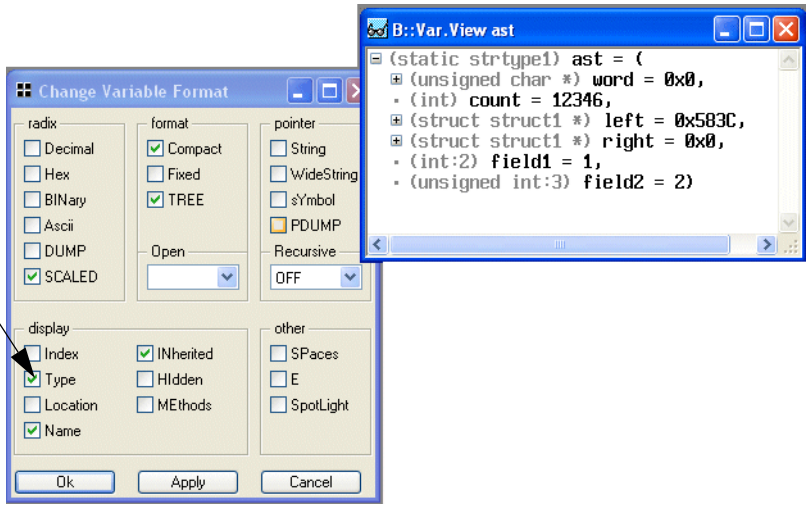
- **Index**

Display array with indices



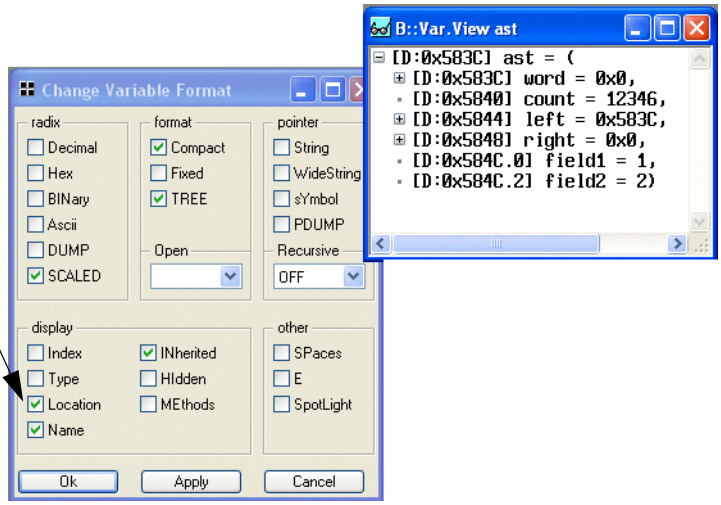
- **Type**

Display variable with type information



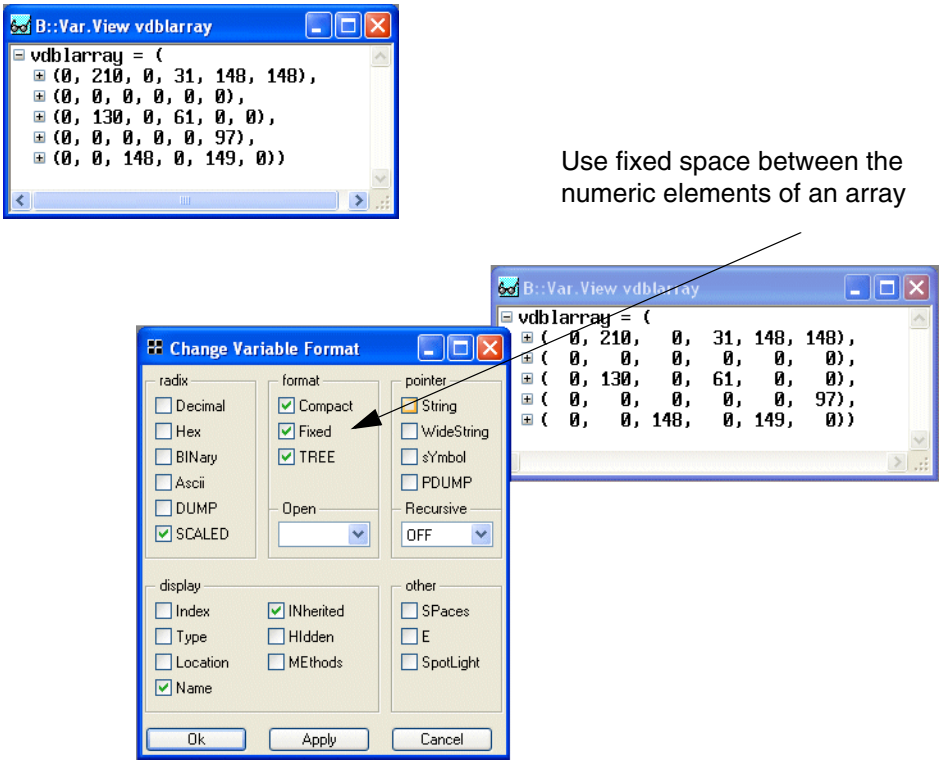
- **Location**

Display variable with location information

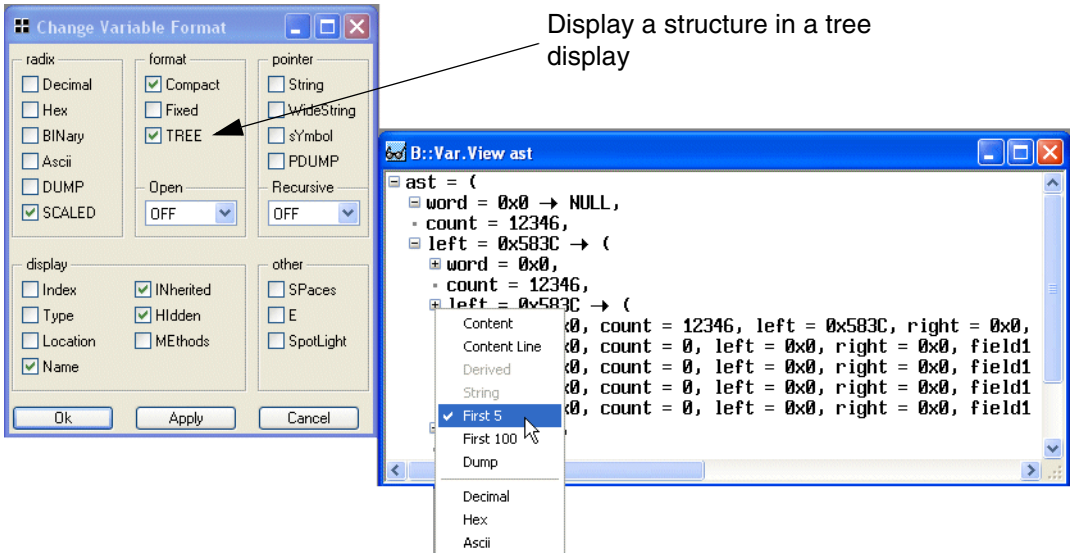


- **Fixed**

  Display all numeric values in a fixed format.



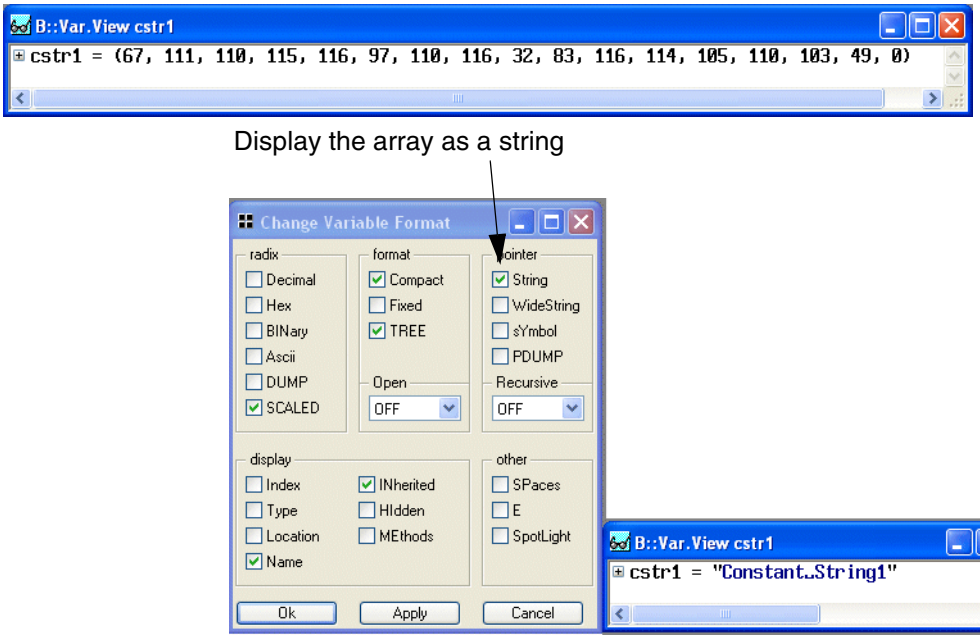
- **Tree**



- **String/WideString**

  This format can be used for arrays or pointer to characters.

String	Each character is a byte.
WideString	Each character is a word e.g. for some DSPs or unicode.

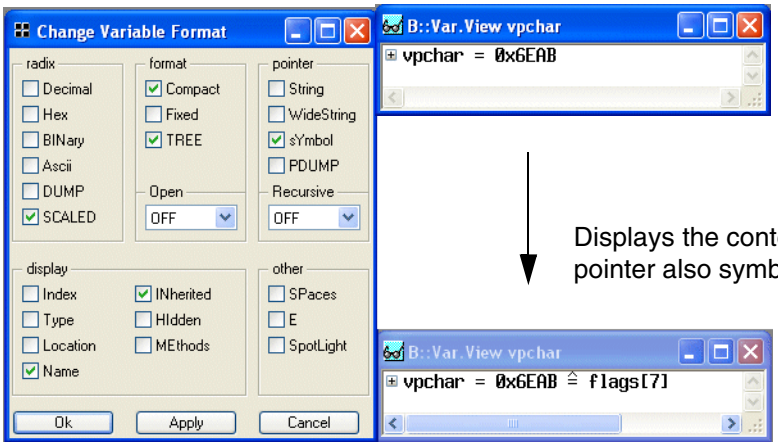


sYmbol.AddInfo.Var *<var>* **ZSTRING**	Define variable contents as a zero-terminated string
sYmbol.AddInfo.List	List all definitions
sYmbol.AddInfo.RESet	Reset list

```
sYmbol.AddInfo.Var cstr1 ZSTRING    The contents of cstr1 is a zero-
                                    terminated string

sYmbol.AddInfo.List                 Display definition list
```

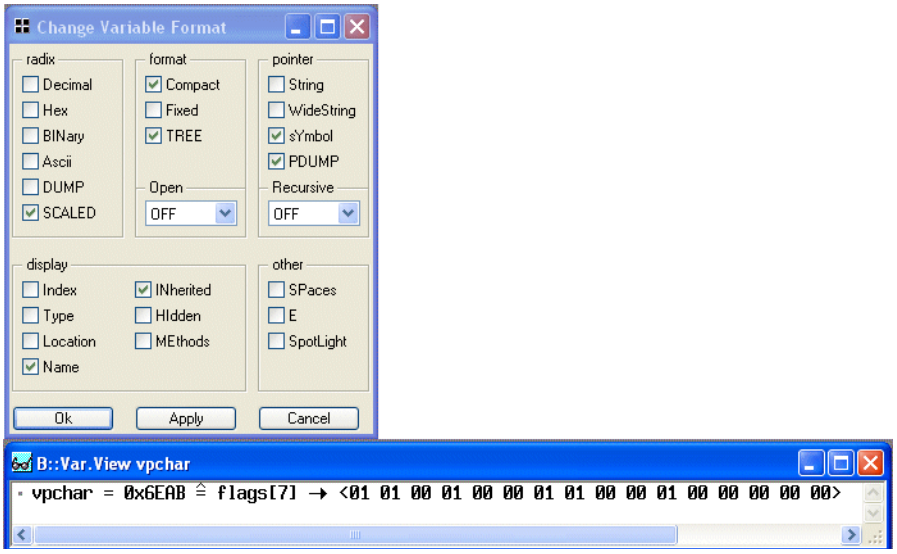

B::sYmbol.AddInfo.List
name/address
member
info
cstr1
(variable)
ZSTRING
B::Var.View cstr1
cstr1 = "Constant_String1"

- **sYmbol**

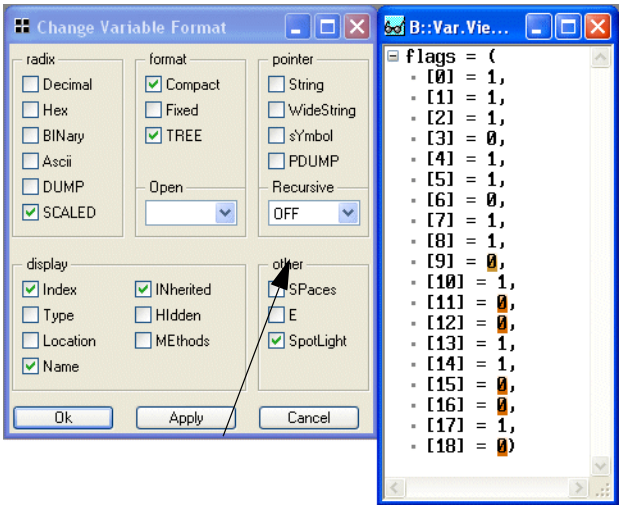


- **PDUMP**

Display a 16 byte hex dump starting at the address where the pointer is pointing to.



- **SpotLight**



Highlight all changed variable elements:
The variable elements changed by the last step are marked in dark red. The variable elements changed by the step before the last step are marked a little bit lighter. This works up to a level of 4.

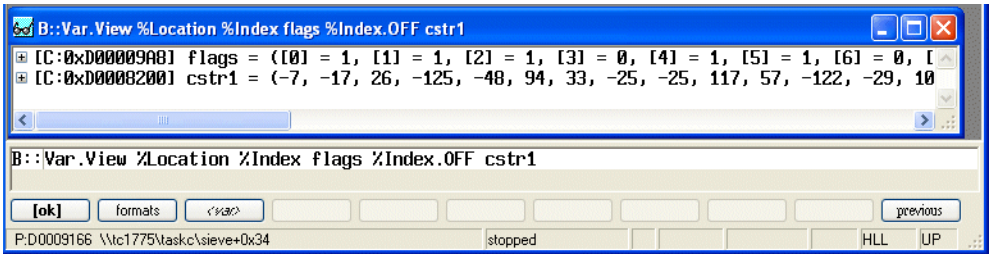
If a variable is formatted using the ***Change Variable Format*** dialog box, the format information will not be stored when the windows configuration is saved in a PRACTICE file.

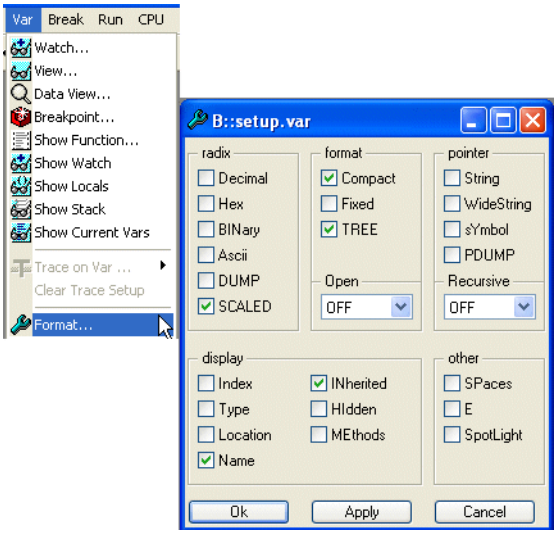
The format information will be stored only, if the variable was formatted using the command line.

Var.View [%<*format*>] <*variable*>

- Format definitions are valid for all variables used in the command after the format definition.

- Format definitions can be switched off selectively.

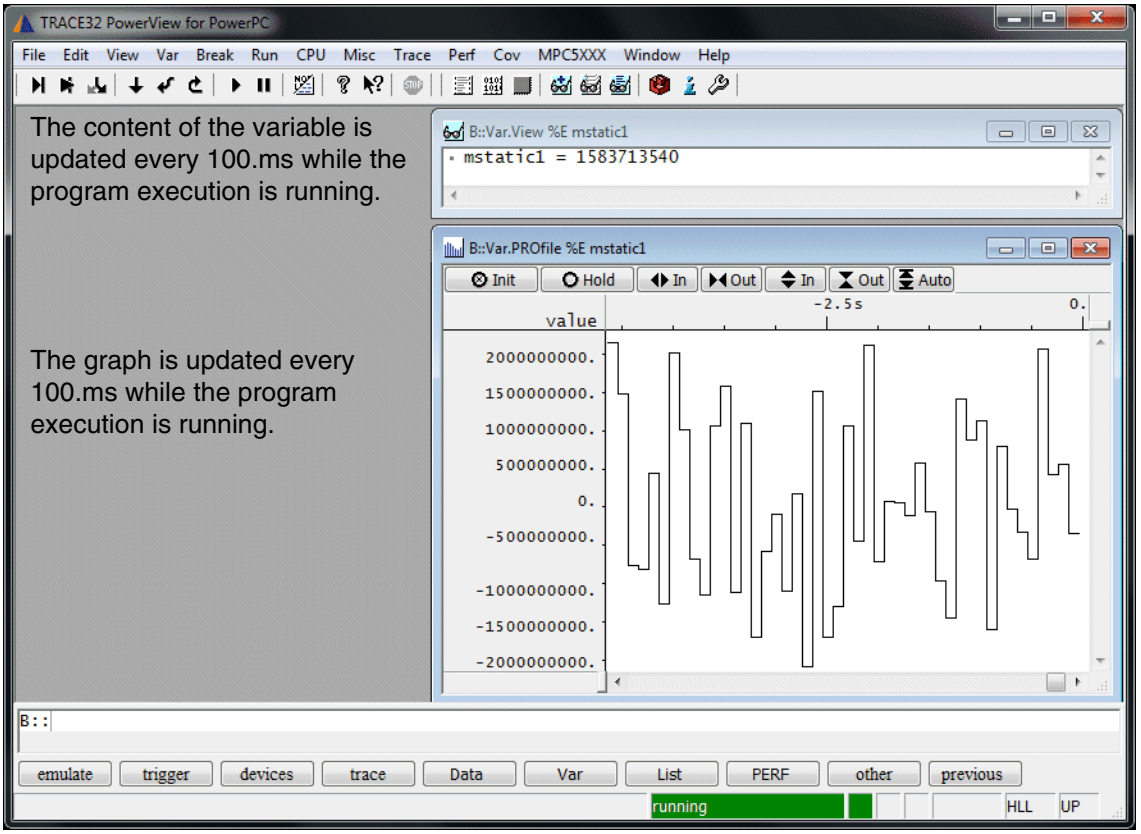




SETUP.Var [%<*format*>…] Change default display format for variables

Basics

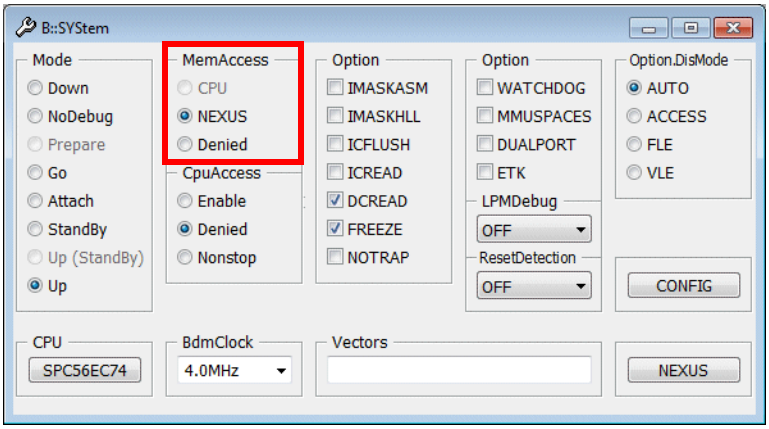
TRACE32 provides the possibility to monitor variable changes while the program execution is running. Monitoring the variable changes is only possible for variables with a fixed address.



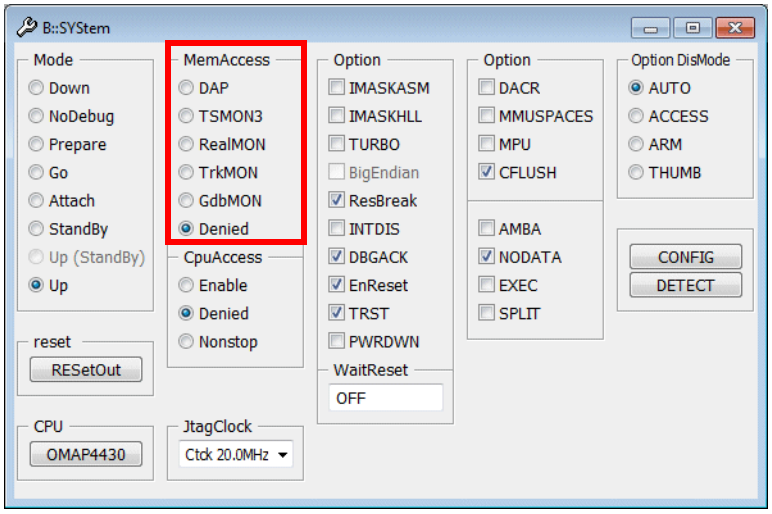
Run-time memory access: If the processor architecture in use allows the debugger to read the target's physical memory while the program execution is running, variables can be monitored without any impact on the program execution. For details, refer to **"Run-time Memory Access"** in TRACE32 Glossary, page 42 (glossary.pdf).

StopAndGo mode: If the processor architecture allows the debugger to read the target's memory only when the program is stopped or if other restrictions don't allow the debugger to read the variable while the program execution is running, the debugger can be configured to stop the program execution every 100 ms in order to read the variable content. For details, refer to **"StopAndGo Mode"** in TRACE32 Glossary, page 52 (glossary.pdf).

No preparation is required if run-time memory access (**SYStem.MemAccess**) is enabled by default.

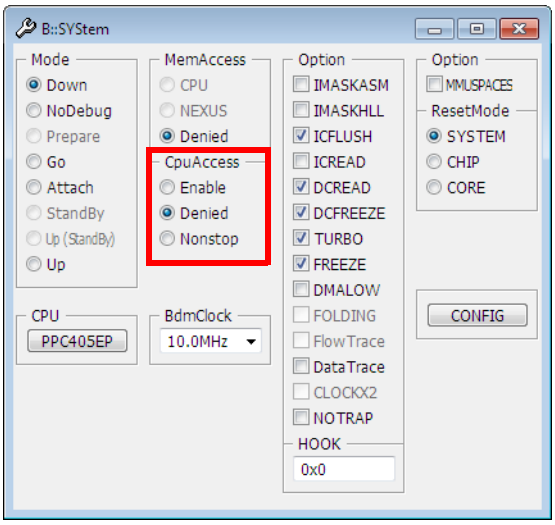


If run-time memory access is **denied** (**SYStem.MemAccess** Denied) by default, please refer to your **Processor Architecture Manual** before you enable it by selecting one of the radio buttons.



SYStem.MemAccess DAP Enable run-time memory access via Debug Access Port (ARM/Cortex architecture)

If run-time memory access is not supported by the processor architecture in use or if other restriction don‘t allow the debugger to read the variable while the program execution is running, you can configure the debugger for StopAndGo mode.



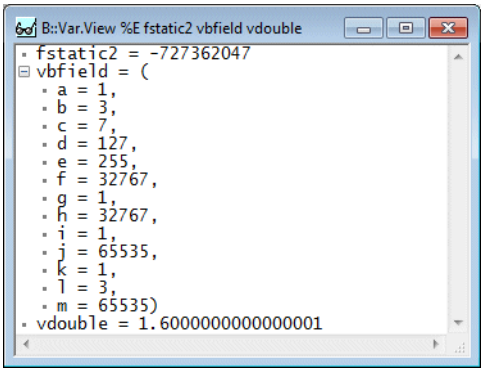
SYStem.CpuAccess Enable Allow StopAndGo mode to read variables.

SYStem.CpuAccess Enable is not recommended for complex multicore chips that use caches and MMU.

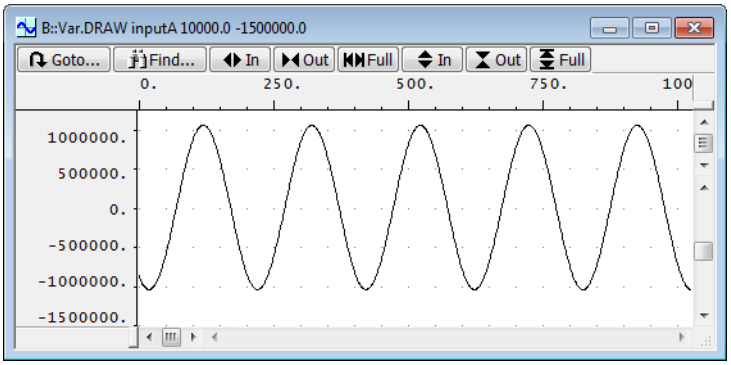
Format Option %E

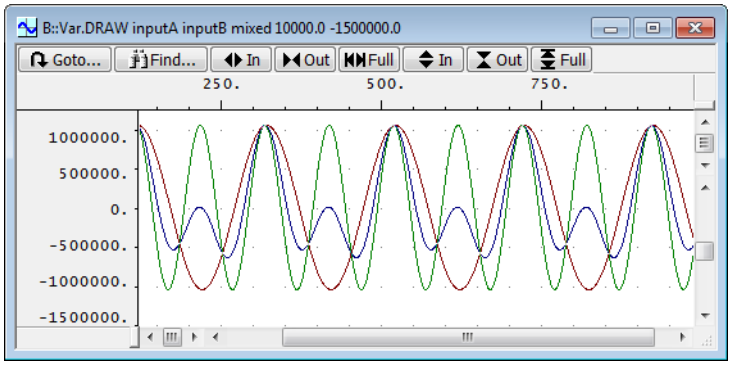
The option %E can be used for most commands that display variables. It advises the debugger to update the display for all variables with a fixed address 10 times per second.

```
Var.View %E fstatic2 vbfield vdouble
```

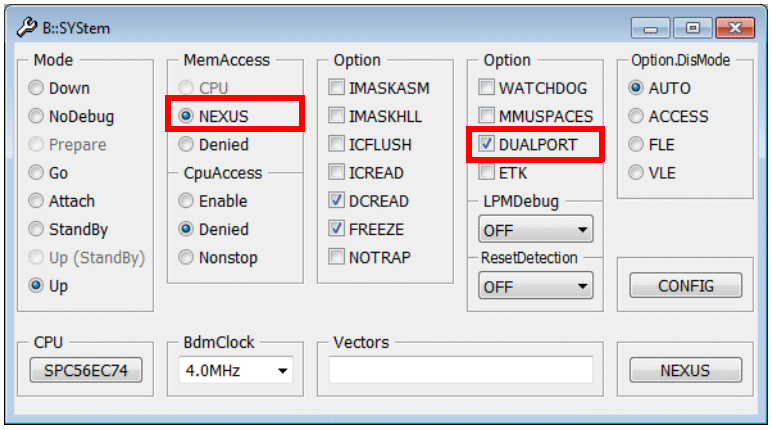



```
Var.DRAW inputA                          ; Displays the contents of an HLL
                                         ; array graphically

Var.DRAW inputA inputB mixed
```





Processor architectures used in the automotive industry provide the option **DUALPORT** in the **System** window. If DUALPORT is checked run-time memory access is automatically enabled for all windows that display memory (e.g. source listing, memory dumps, variable displays, displays of SFR). The format option %E can be omitted in this case.

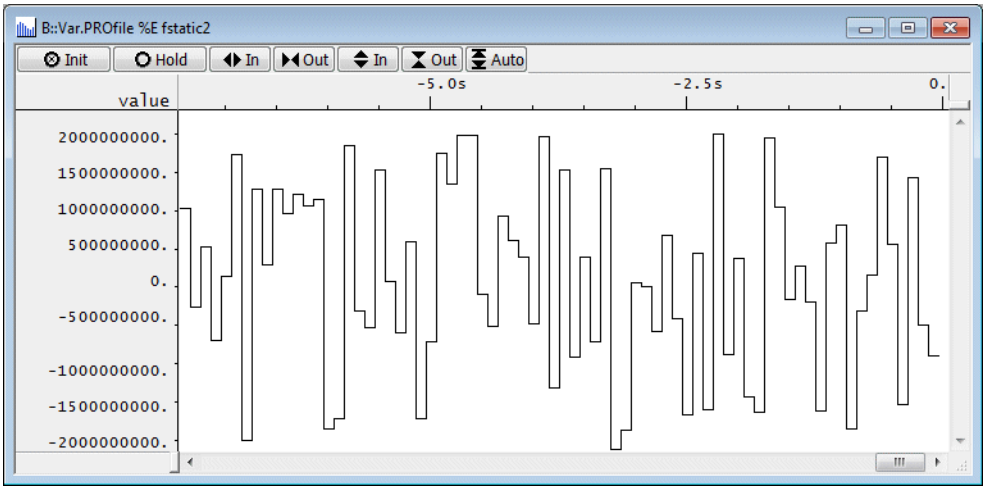


SYStem.Option.DUALPORT ON

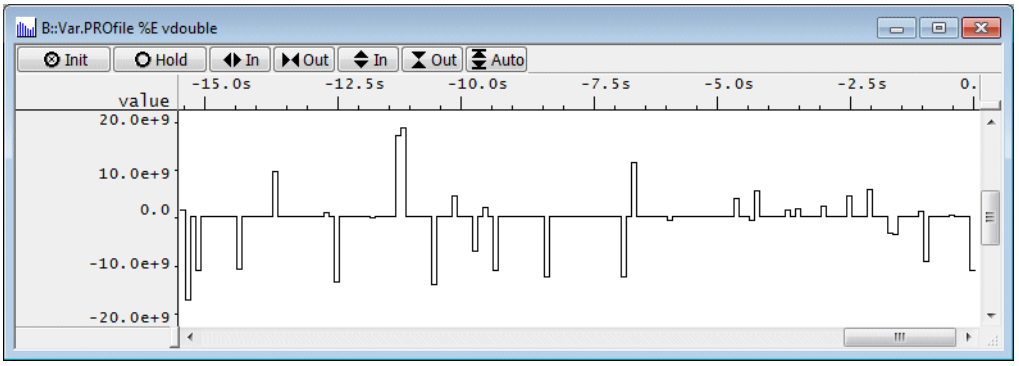
Var.PROfile Command

The command **Var.Profile** allows to monitor numeric variables and display their changes graphically.

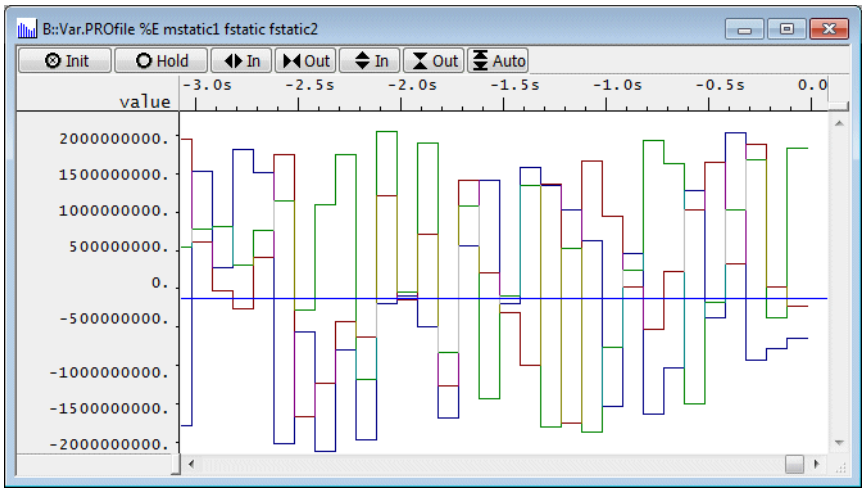
```
Var.PROfile %E fstatic2
```




```
Var.PROfile %E vdouble
```



Up to three variable can be superimposed if required. The following color assignment is used: first variable value red, second variable value green, third variable value blue.



Var.PROfile %E *<variable1>* [*<variable2>*] [*<variable3>*]

Variable Logging

Variable changes can be logged in the following way:

- **SNOOPer trace**
- **Var.LOG command**

SNOOPer Trace

A video tutorial about the SNOOPer trace can be found here:

https://www.lauterbach.com/tut_snooper.html

Basics

Some processor architectures allow the debugger to read the target's physical memory while the program execution is running. For details refer to **"Run-time Memory Access"** in TRACE32 Glossary, page 42 (glossary.pdf).

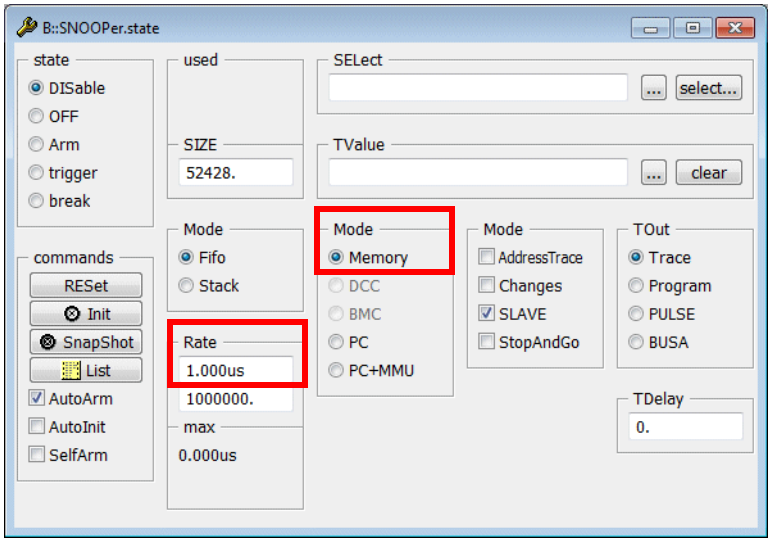
TRACE32 implements the so-called SNOOPer trace based on this feature. Memory content is read periodically or as fast as possible and stored with timestamp information into a trace memory. The trace memory for the SNOOPer is allocated on the host.

First example:

The following steps are required to set up the SNOOPer trace:

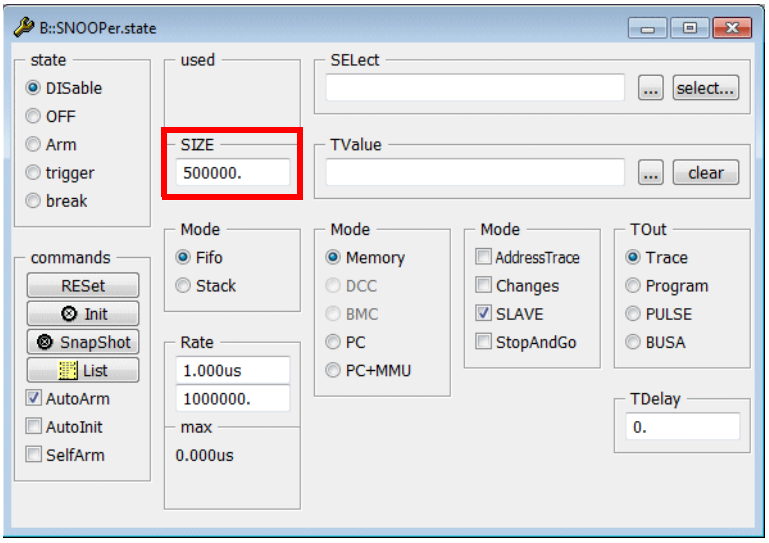
3. Open the SNOOPer configuration window.

SNOOPer.state



SNOOPer.Mode Memory	Reading memory in the specified Rate is the default setting for the SNOOPer trace.

4. **Specify the SNOOPer size as *<number of trace records>*.**



TRACE32 allocates memory on the host for the requested size.

The SNOOPer size is only limited by the size of RAM on the host. It is recommended to stay far below this limit so that sufficient free memory is available for TRACE32 and other applications.

SNOOPer.SIZE *<number of records>*

```
SNOOPer.SIZE 500000.
```

5. Specify the variable you are interested in.

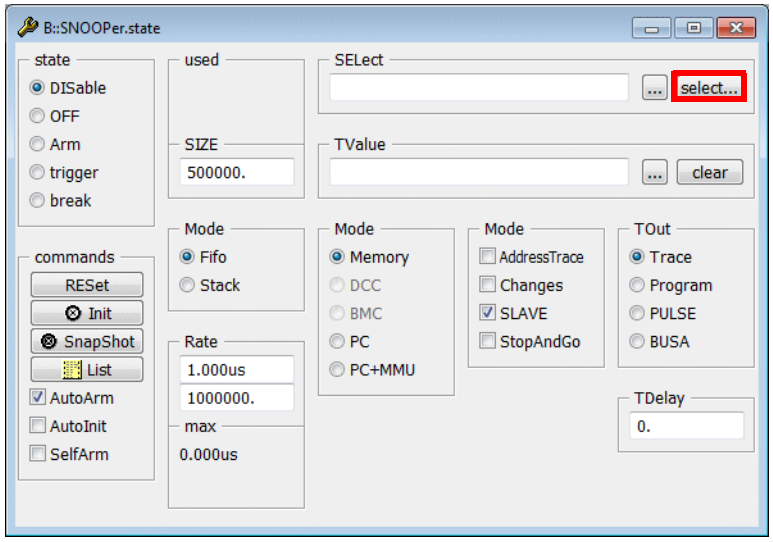
It is best to read variables via the SNOOPer whose sizes are smaller or equal the data bus width of the core in use.

To specify the variable:

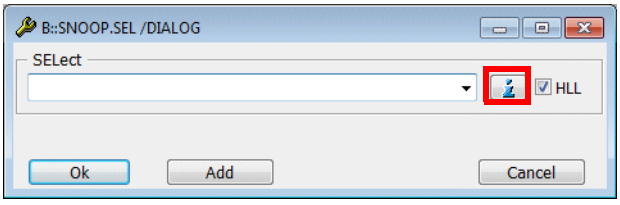
3.1. Use the **select…** button in the SNOOPer configuration window to open the **SNOOPer.SELect** dialog.

3.2. Use the **List Symbols** button in the SNOOPer.SELect dialog to get a list of all variables.

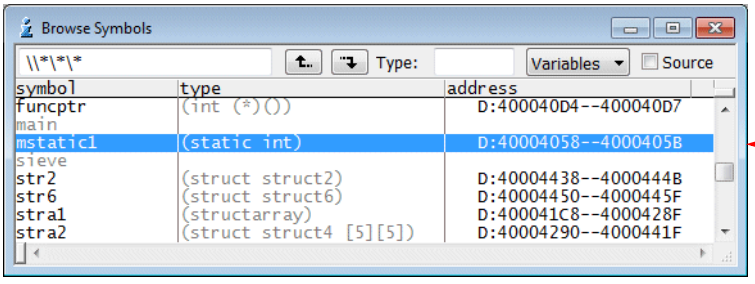
3.3. Select the variable you are interested in.



3.1

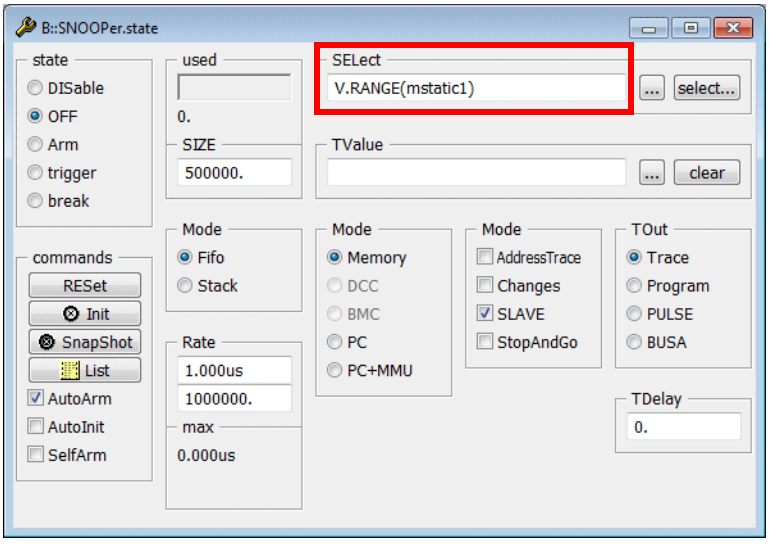


3.2



3.3

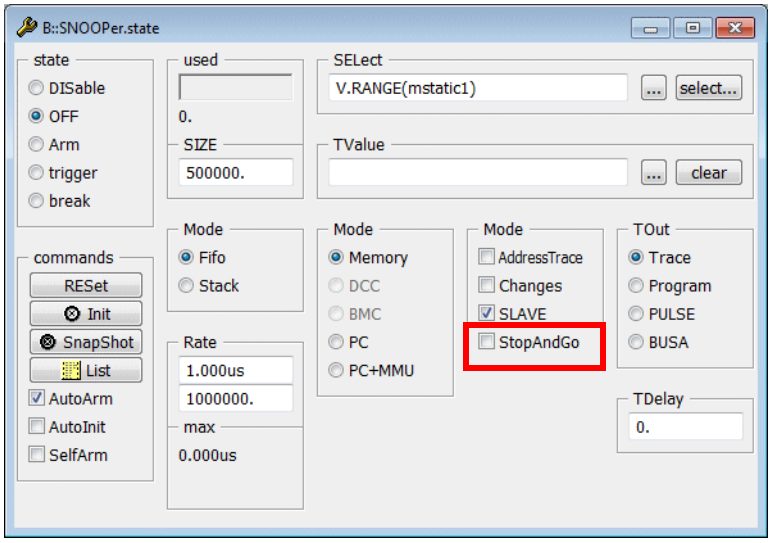
The selected variable is listed in the **SELect** field of the SNOOPer configuration window.



SNOOPer.SELect Var.RANGE(*<variable>***)**

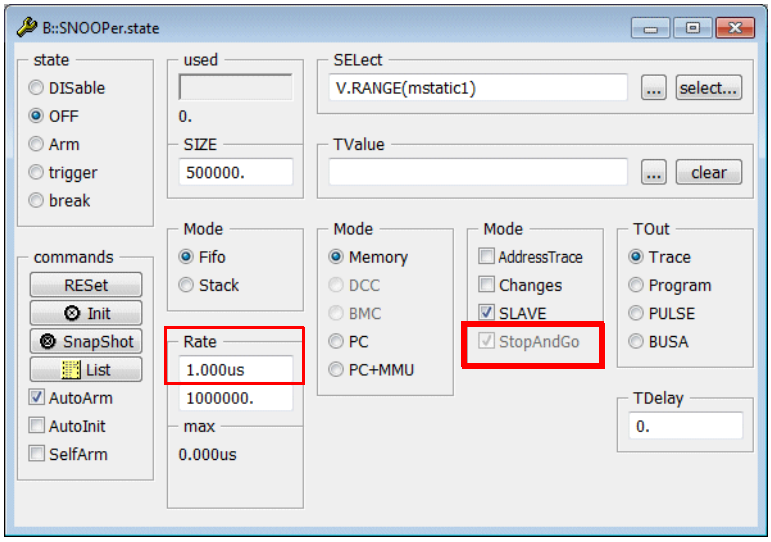
Var.RANGE(*<variable>***)**	This TRACE32 function returns the address range used by a variable

SNOOPer.Mode StopAndGo



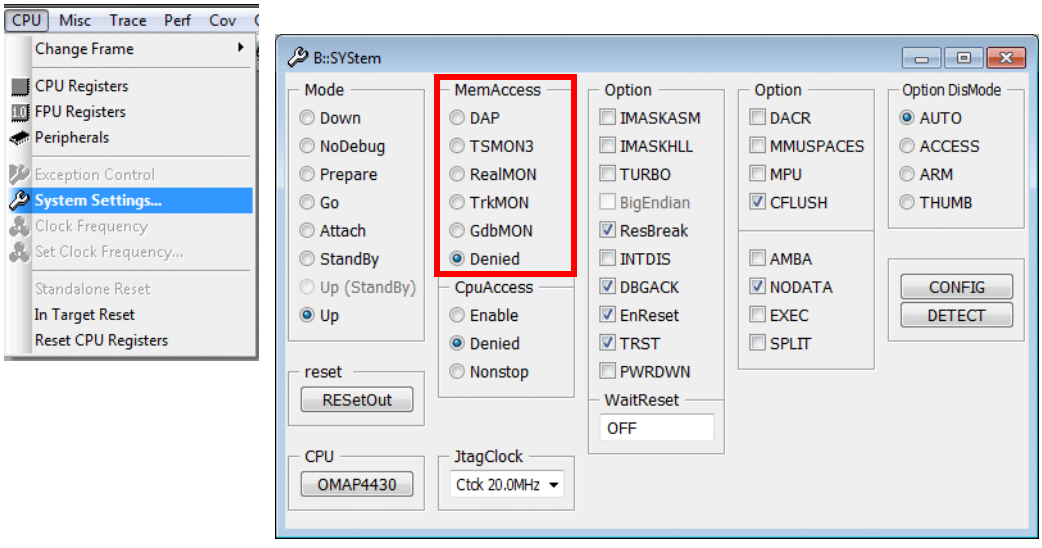
TRACE32 checks/unchecks the **StopAndGo** checkbox automatically.

OFF	The processor architecture in use allows the debugger to read physical memory while the program execution is running and this debugger feature is enabled.



ON	The processor architecture in use does not allow the debugger to read physical memory while the program execution is running or this debugger feature is disabled. If the SNOOPer is working in StopAndGo mode, the program execution is stopped in the specified Rate in order to read the variable content. Such a stop can take more than 1 ms in the worst case scenario.

Open the **SYStem** settings window to check if reading the physical memory while the program execution is running can be enabled for your debugger.



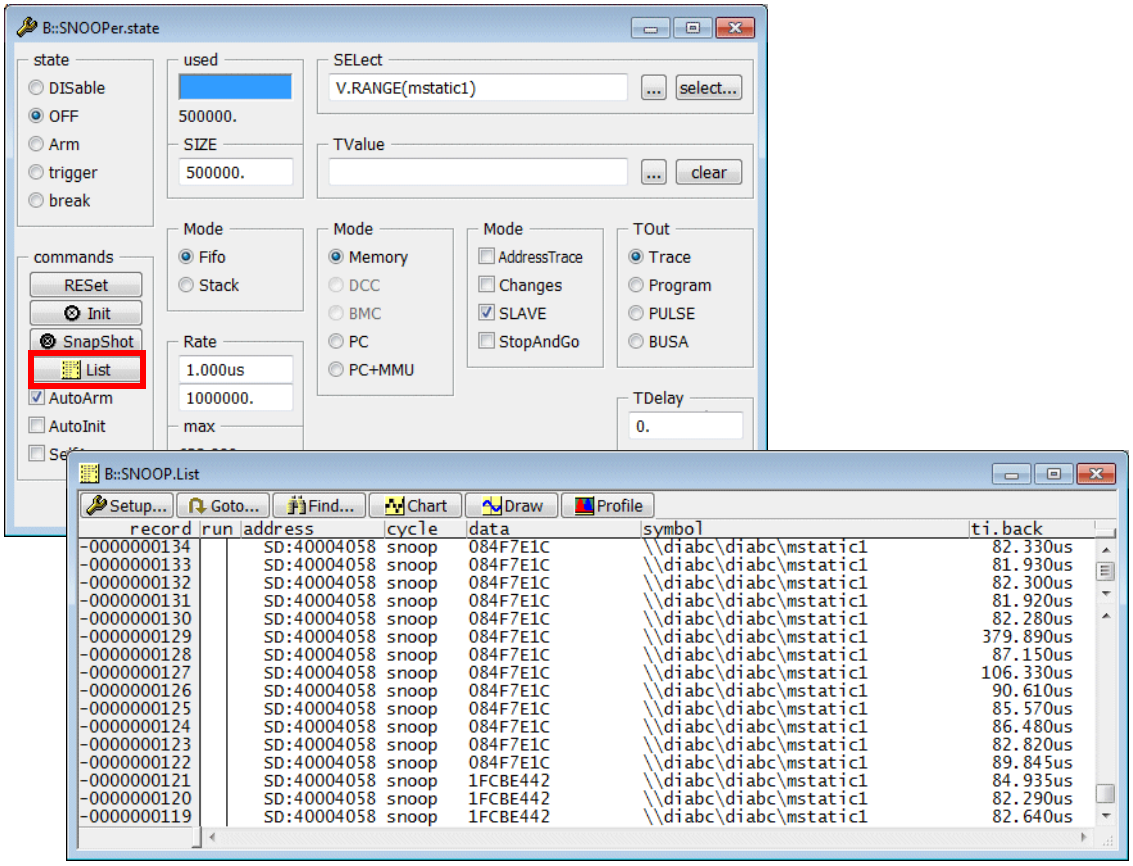
If there are beside **Denied** other selectable radio buttons in the **MemAccess** field refer to your **Processor Architecture Manuals** before you select one.

6. **Start the program execution.**

7. **Stop the program execution.**

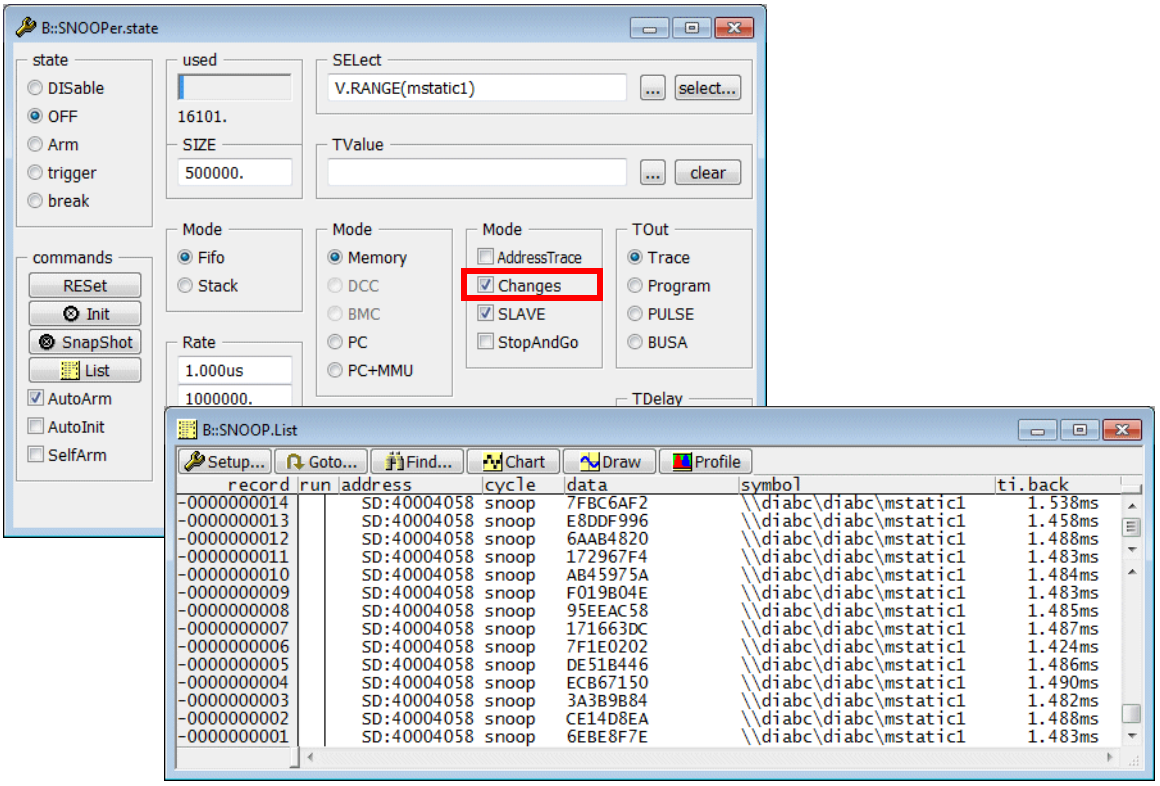
   Please be aware that the contents of the SNOOPer trace can not be read while recording.

8. **Display the result by pushing the List button.**



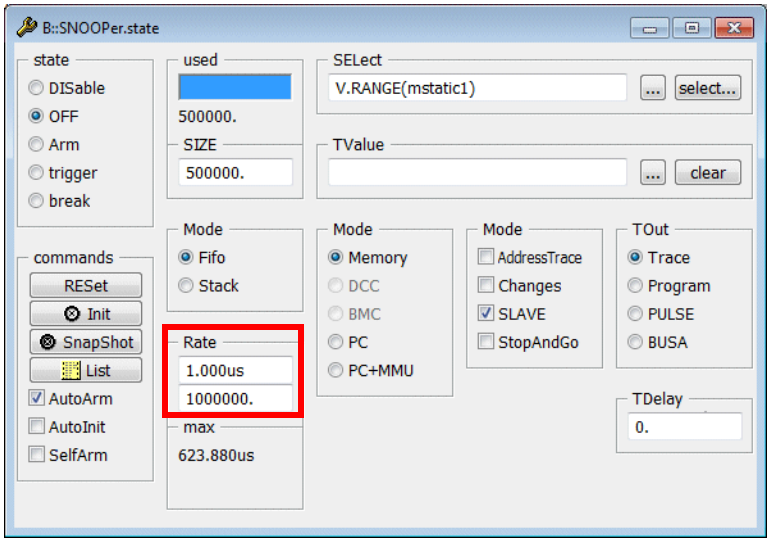
SNOOPer.List

Check **Mode Changes**, if the read variable content should only be stored to the SNOOPer trace when it has changed.



SNOOPer.Mode Changes ON

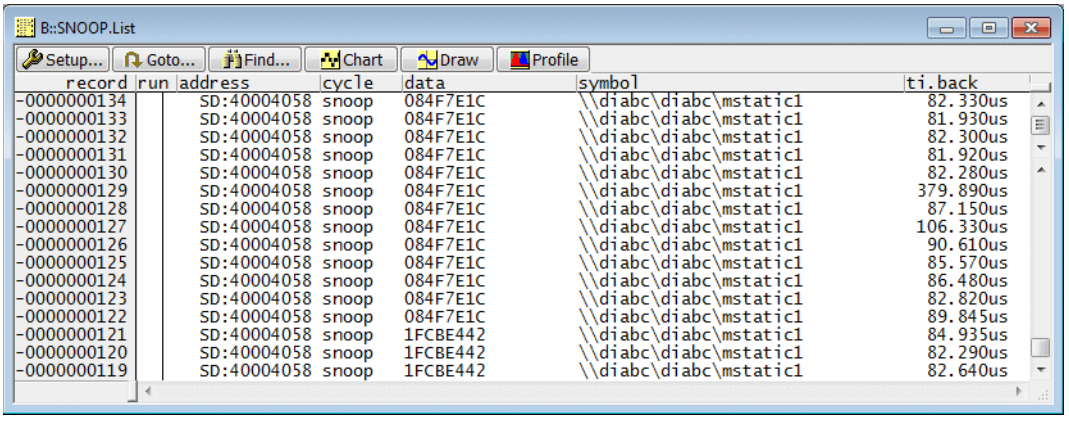
The time interval (**SNOOPer.Rate**) at which TRACE32 reads the physical memory at program runtime is set to 1.us by default.



The rate at which the debugger can actually read the physical memory is bigger.

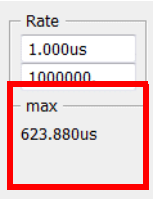
The actual rate might be increased by a higher JTAG clocks (**SYStem.JtagClock** *<frequency>*). Please refer to your processor/chip manual to find out what the max. JTAG clock can be.

In the example recording below the average time interval is about 85.us. So it is recommended to use the **SNOOPer** only for variables that are changed at a higher rate by the application program.

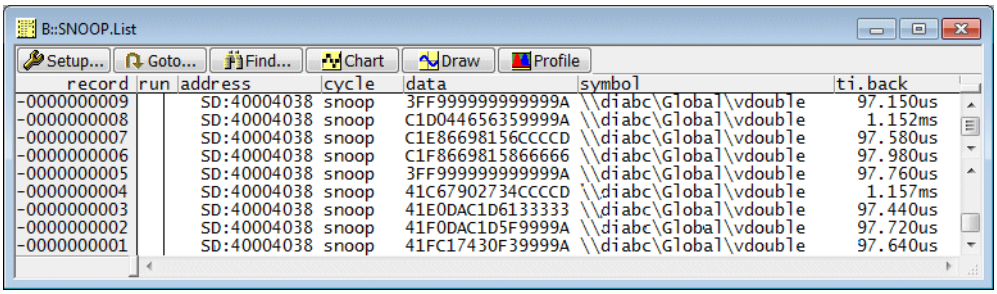


record	run	address	cycle	data	symbol	ti.back
-0000000134		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	82.330us
-0000000133		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	81.930us
-0000000132		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	82.300us
-0000000131		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	81.920us
-0000000130		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	82.280us
-0000000129		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	379.890us
-0000000128		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	87.150us
-0000000127		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	106.330us
-0000000126		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	90.610us
-0000000125		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	85.570us
-0000000124		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	86.480us
-0000000123		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	82.820us
-0000000122		SD:40004058	snoop	084F7E1C	\\diabc\diabc\mstatic1	89.845us
-0000000121		SD:40004058	snoop	1FCBE442	\\diabc\diabc\mstatic1	84.935us
-0000000120		SD:40004058	snoop	1FCBE442	\\diabc\diabc\mstatic1	82.290us
-0000000119		SD:40004058	snoop	1FCBE442	\\diabc\diabc\mstatic1	82.640us

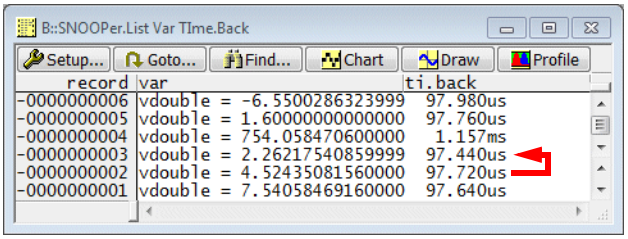
Both, the host and the debugger are no real-time systems, so individual time intervals can be longer then the average interval. The longest snooping interval for the current recording is displayed in the **max** field of the **SNOOPer.state** window.



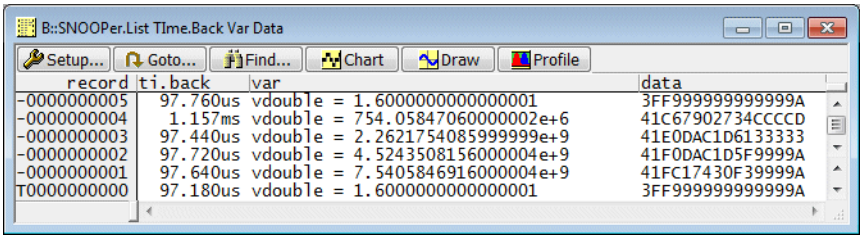
In addition to the default **SNOOPer.List** display various other display options are provided.



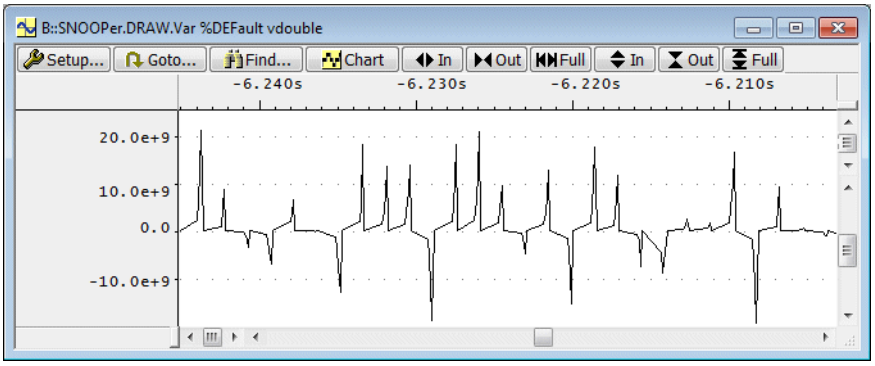

```
SNOOPer.List Var TIme.Back              ; list the recorded variable in
                                        ; its HLL representation together
                                        ; with the time relative to the
                                        ; previous record
```




```
SNOOPer.List TIme.Back Var Data         ; rearrange the column layout so
                                        ; it fits your requirements
```



```
SNOOPer.DRAW.Var %DEFault vdouble      ; display the changes of the
                                       ; variable over the time as a graph
```

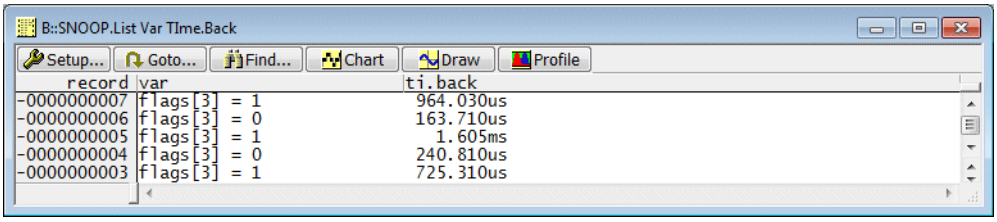


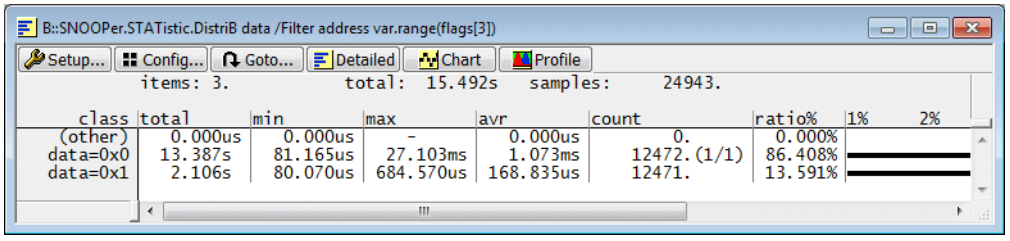
If you are analyzing a variable that maintains a state, the following display options might be useful:

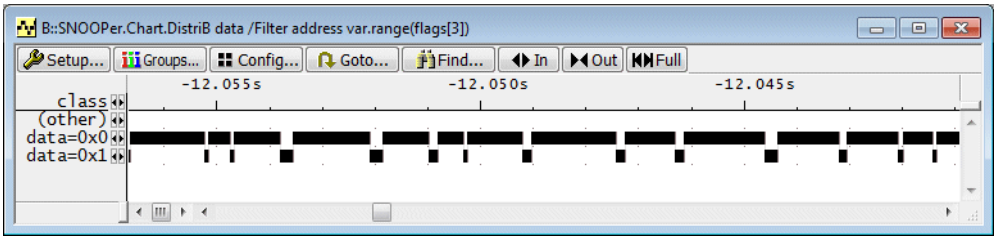
```
SNOOPer.List Var TIme.Back
```

```
; display the statistical distribution of a variable value over the time
; Data advise the command to analyze the recorded data information
; Address informs the command for which address the data
; should be analyzed
SNOOPer.STATistic.DistriB Data /Filter Address Var.RANGE(flags[3])
```

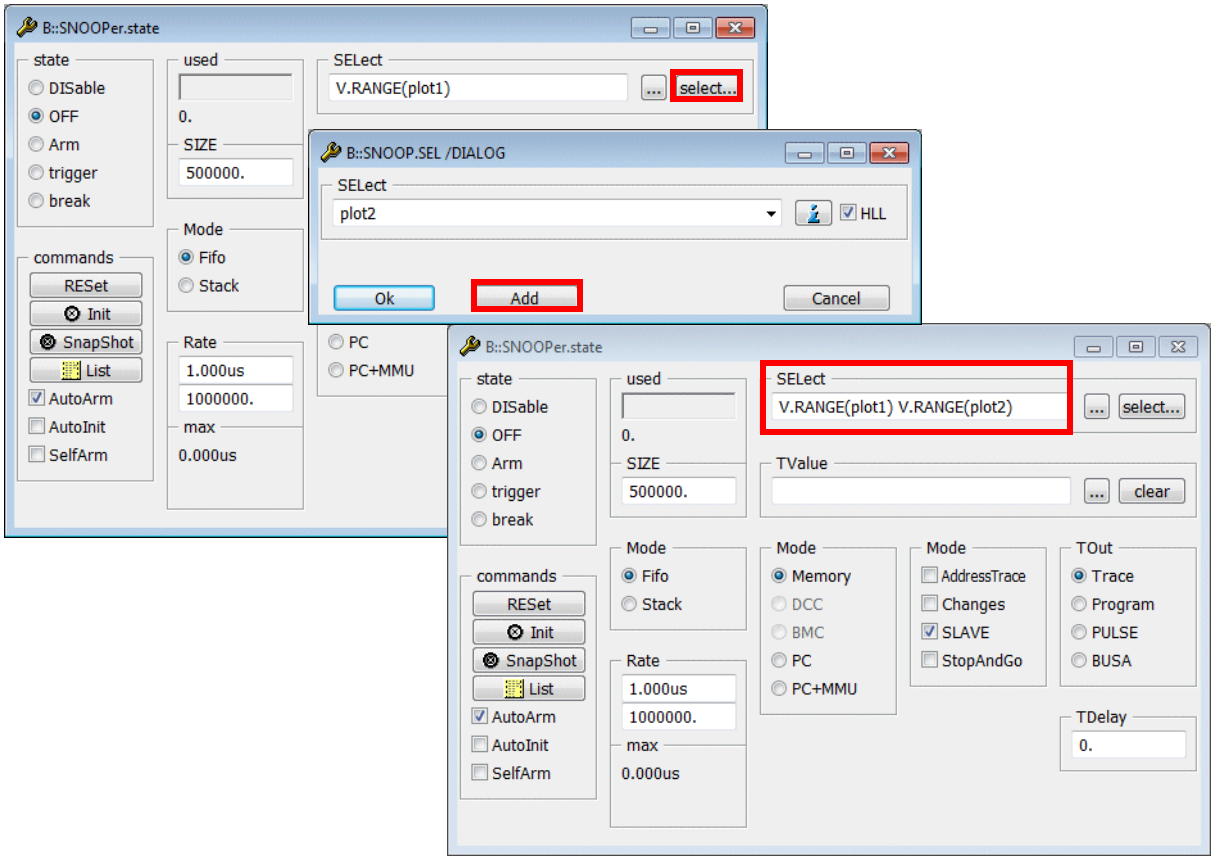
```
; display a time chart of the variable values
SNOOPer.Chart.DistriB Data /Filter Address Var.RANGE(flags[3])
```





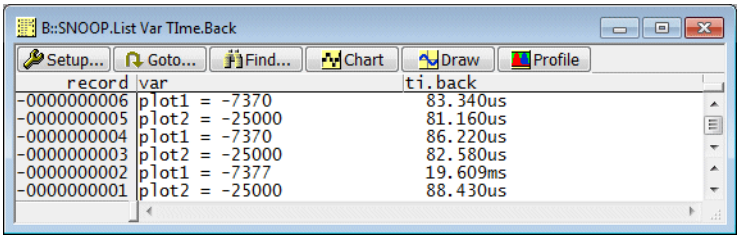


TRACE32 PowerView allows the logging of up to 16 variables.



If you use the **Add** button in the **SNOOPer.SELect** dialog, additional variables that should be read by the SNOOPer can be selected.

SNOOPer.SELect Var.RANGE(*<variable1>*) Var.RANGE(*<variable2>*) …

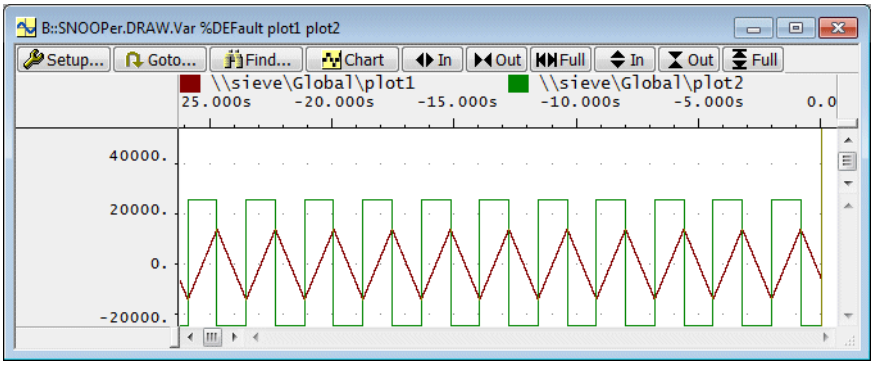


Please be aware that the time interval at which a single variable can be read by the debugger at program run-time is growing with every selected variable.

For the graphical display of variables changes over the time, you can:

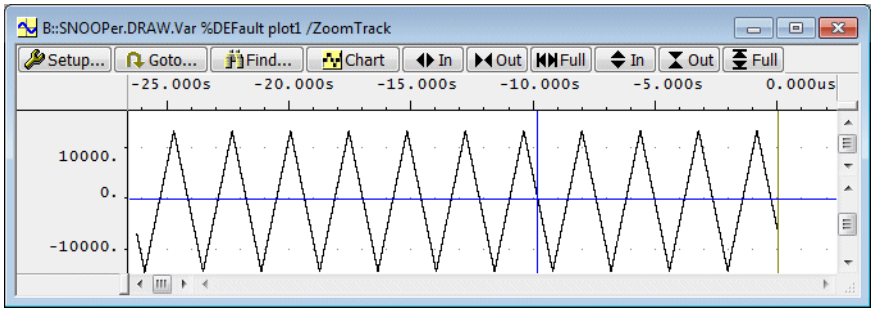
- superimpose up to three variables
- establish a time- and zoom-synchronization between the different displays

```
SNOOPer.DRAW.Var %DEFault plot1 plot2      ; superimpose variables
```

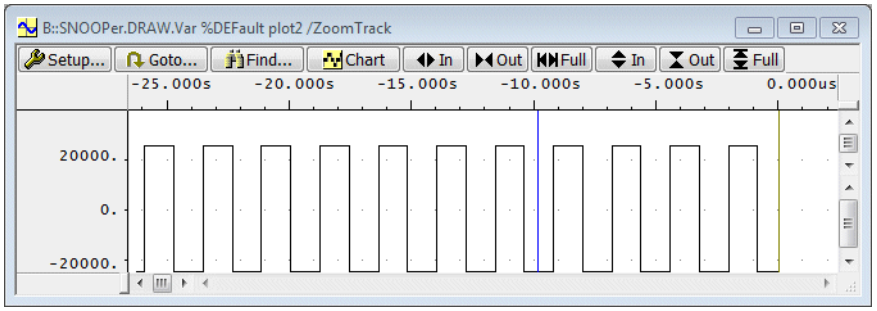


```
SNOOPer.DRAW.Var %DEFault plot1 /ZoomTrack      ; the option ZoomTrack
                                                ; establishes time- and
                                                ; zoom-synchronisation
                                                ; between display windows

SNOOPer.DRAW.Var %DEFault plot2 /ZoomTrack
```

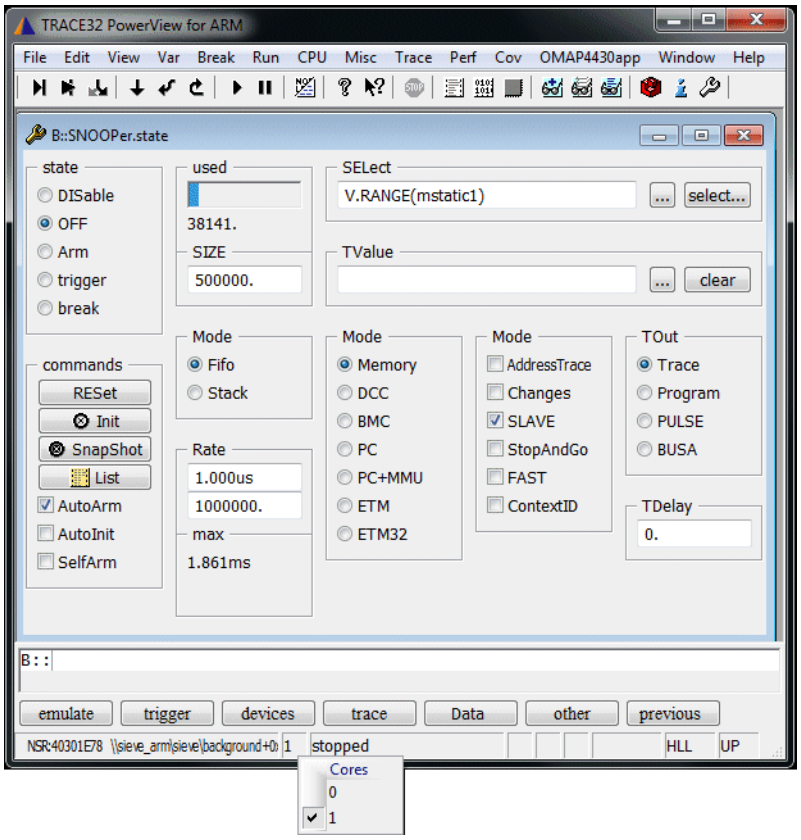
Active window

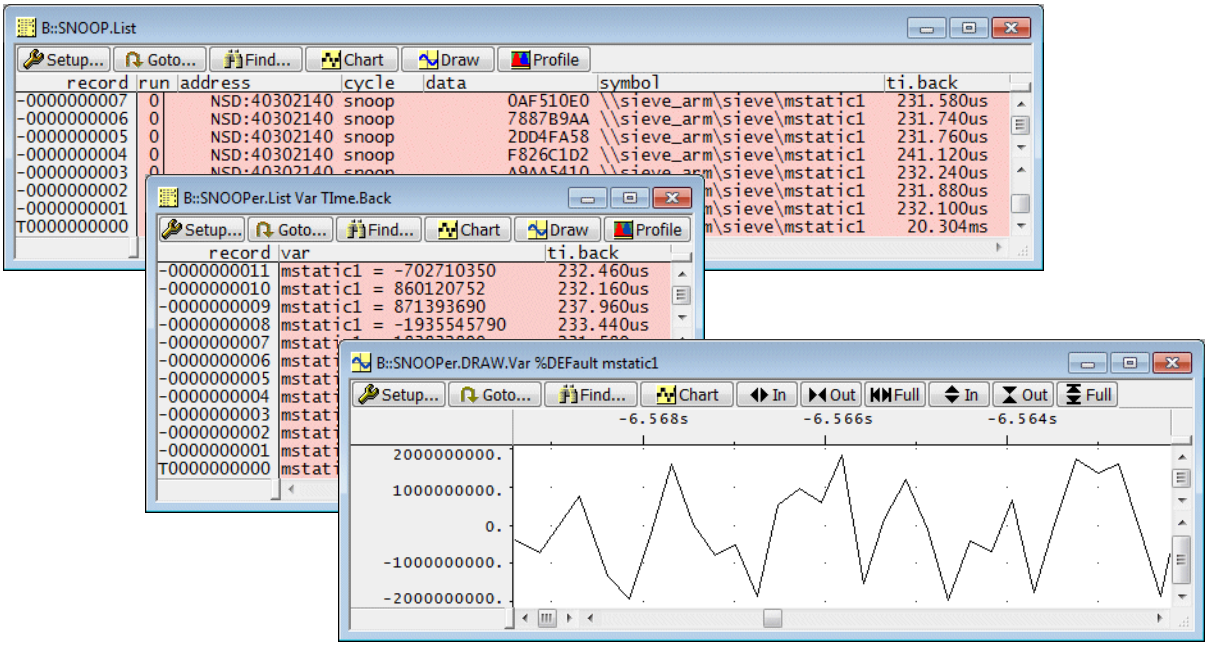


Windows with the option **/ZoomTrack** are time- and zoom-synchronized to the cursor in the active window



The SNOOPer can also be used while debugging an SMP system. The debugger can read the shared memory as an independent bus master .





Document the Logging Results

```
PRinTer.FILE snoop_plot1.lst          ; specify documentation file name

PRinTer.FileType CSV                  ; specify Comma-Separated Value as
                                      ; output format

WinPrint.SNOOPer.List                 ; save result of the command
                                      ; SNOOPer.List to file
```

Summary

- Only recommended if your processor architecture allows the debugger to read physical memory while the program execution is running.
- Recommended for variables whose sizes are smaller or equal to the core data bus width.
- Only recommended for variables that change with a lower frequency then the achievable SNOOPer frequency.

- Up to 16 variables can be read while the program execution is running.
- Read values are timestamped and stored in the SNOOPer trace memory. The SNOOPer trace size is only limited by the RAM on the host computer.
- SNOOPer trace can not be read while recording.
- Various display options are provided.

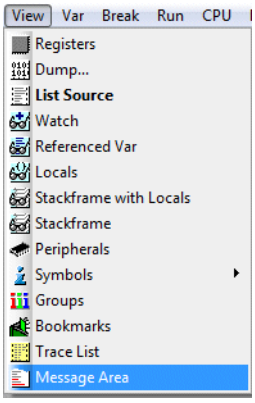
Script Example

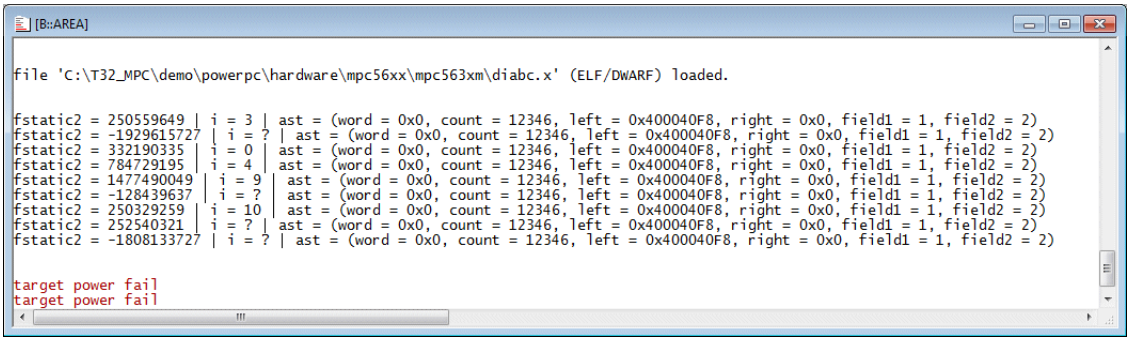
```
…

SNOOPer.RESet                          ; reset the SNOOPer functionality
                                       ; to its default settings

SNOOPer.state                          ; display a SNOOPer configuration
                                       ; window

SNOOPer.SIZE 500000.                   ; specify the size of the SNOOPer
                                       ; trace

SNOOPer.Rate 500.us                    ; specify the SNOOPer sampling rate

SNOOPer.AutoInit ON                    ; advise TRACE32 to delete the
                                       ; contents of the SNOOPer trace
                                       ; whenever the program execution is
                                       ; started with Go or Step

SNOOPer.SELect Var.RANGE(var1)         ; specify the variable that should
                                       ; be logged by the SNOOPer trace

Go                                     ; start the program execution

WAIT 5.s                               ; wait 5 seconds

Break                                  ; stop the program execution

SNOOPer.List Var TIme.Back             ; display the result as a list

SNOOPer.DRAW.VAR %DEFault var1         ; display the result as a time
                                       ; graph

…
```

Var.LOG Command

The command **Var.LOG** advises TRACE32 PowerView to log the contents of the specified variables to the TRACE32 PowerView Message AREA whenever the program execution is stopped. Any variable can be logged.

```
Var.LOG fstatic2 i ast
```





AREA.view	Display TRACE32 PowerView Message Area.
Var.LOG [%<*format*>] <*variable1*> …	Log specified variables to TRACE32 PowerView Message AREA.
Var.LOG	End logging.

Since the TRACE32 PowerView Message AREA also includes all system and error messages it is recommended to use a dedicated AREA for the variable logging.

AREA.Create *<name>*	Set up an new AREA window. Please be aware that *<name>* is case sensitive.
AREA.view *<name>*	Display AREA window.
AREA.CLEAR *<name>*	Clear the AREA window.
Var.LOG [**%***<format>*] *<variable1>* **.../AREA** *<name>*	Log the specified variables to the area.

```
AREA.Create VarLogging

AREA.view VarLogging

Var.LOG fstatic2 i ast /AREA VarLogging

…

Var.LOG
```

The following command allow to redirect the area outputs to a file.

AREA.OPEN *<name> <file>*	Save outputs to area *<name>* to *<file>.*
AREA.CLOSE *<name>*	Stop output and close *<file>.*

```
AREA.Create VarLogging

AREA.OPEN VarLogging log1.txt

AREA.view VarLogging

Var.LOG fstatic2 i ast /AREA VarLogging

…

Var.LOG

AREA.CLOSE VarLogging

TYPE log1.txt
```

Var.set [%<*format*>] <*var*> Execute a function in the target

```
Var.set func5(4,8,17)
```