LAUTERBACH
DEVELOPMENT TOOLS

General Commands Reference Guide V

Release 02.2024

MANUAL

General Commands Reference Guide V

General Commands Reference Guide V

Version 04-Mar-2024

History

23-Jan-2023 VPU commands support different architectures other than PowerPC.

Var

HLL variables and expressions

See also

- Var.AddSticker
- Var.AddWatch
- Var.AddWatchPATtern
- Var.Assign
- Var.Break
- Var.Call
- Var.CHAIN
- Var.DelWatch
- Var.DRAW
- Var.DRAWXY
- Var.DUMP
- Var.Eval
- Var.EXPORT
- Var.FixedCHAIN
- Var.FixedTABle
- Var.Go
- Var.IF
- Var.INFO
- Var.Local
- Var.LOG
- Var.NEW
- Var.NEWGLOBAL
- Var.NEWLOCAL
- Var.OBJECT
- Var.PATtern
- Var.PRINT
- Var.PROfile
- Var.Ref
- Var.set
- Var.Step
- Var.TABle
- Var.TREE
- Var.TYPE
- Var.View
- Var.Watch
- Var.WHILE
- Var.WRITE
- SETUP.Var
- sYmbol.CASE
- Var.ADDRESS()
- Var.END()
- Var.RANGE()
- Var.SIZEOF()
- Var.STRing()
- Var.TYPEOF()
- Var.VALUE()

- ▲ 'Var Functions' in 'General Function Reference'
- ▲ 'Release Information' in 'Legacy Release History'

Overview Var

Lower and upper case letters are distinguished in symbol names. The command **sYmbol.CASE** switches off this differentiation. The length of symbol names is limited to 255 characters. The maximum number of symbols depends on the size of the system memory.

Symbol Prefix and Postfix

Most of the compilers add a special character (for example "." or "_") in front of or behind the users symbol names. The user does not need to enter this character. The symbol management automatically adds the character, if necessary.

Example for the processing of prefix/postfix characters.

Symbol Table	Entry	HLL-Windows	Assembler windows
_vfloat	_vfloat or vfloat	vfloat	_vfloat

Symbol Paths

There are two modes during entry a symbol name: entering a complete symbol path or solely a symbol name. If only a symbol name is used, the access will occur to the symbol valid for the used program part (if symbol names are used more than once, local symbols are preferred to symbols of higher blocks).

By specifying a complete symbol path access to any symbol is possible. Each part of the symbol path is separated by a '\'. A complete path has to begin with a '\'. The following path versions are allowed:

```
\modul\global …
\modul\modul-local …
\\program\modul …
```

If the specified symbol represents a function, the access to local variables of this function and of nested functions will be possible:

```
…\function\local
…\function\function …
```

If using PASCAL, as many functions as chosen will be nested.
Line numbers can be specified in the following way:

```
\linenumber
\linenumber\columnnumber
\module\linenumber
\\program\module\linenumber
..\function\relative_linenumber
```

The address of the high level language block containing the specified line number is returned by this operation.

Search Paths

If no complete path is entered, the symbol will be searched in the following sequence

1. Local symbols (interior block … exterior block)
2. Static symbols of block
3. Static symbols of module
4. Global symbols of current program
5. All other static symbols

Mangled Names and C++ Classes

The class of a method can be left out, if this method exists only in one class and the names are ANSI mangled. The class is always required, if the constructor, destructor or an overloaded operator must be accessed. The quotation marks can help to allow special characters if the C++ name is used in the regular TRACE32 syntax. They are not required in the **Var** command group. However they can be used when specifying a local symbol. The command **sYmbol.MATCH** can control the behavior if an overloaded method is not specified with the prototype.

```
List `class1::method1`
List method1                                //access to same method (ANSI
                                            // mangled)

List `class1::class1`                       //creator of class class1
List `class1::~class1`                      //destructor of class class1

List `class1::operator++`                   //overloaded operator

List `class1::operator+(int)`               //overloaded operator with
                                            // prototype

Var.set `class1::operator+(int)\i`          //local variable of function
```

Function Return Values

The return value of a function is entered in the symbol list as a local variable of the function. It has always the name 'return'.

Special Expressions

The expression interpreter accept some extensions to the language. All type checks and range checks are handled as free as possible. Accessing data beyond the array limits is allowed.

A dereference of a plain number will assume that it is a pointer to character:

```
Var.set *0x2000 = 1                 //set byte at location 2000 (decimal)
```

All labels (typeless symbols) can be used in expressions. They are taken as variables of the type void. They can be cast directly to the wanted type.

```
Var.set __HEAP                //displays nothing (if __HEAP is a label)
Var.set *__HEAP               //assumes __HEAP as a pointer to character
Var.set (long)__HEAP          //takes __HEAP as a 'long' variable
```

Function calls can be made to plain addresses or typeless symbols. The return value is assumed to be 'void'.

```
Var.set (0x2000)(1,2,3)        //calls the function at 2000 (hex)
Var.set __HEAP(1,2,3)          //calls the function at the label __HEAP
```

Extracts of arrays can be made with 'range' expressions. The operations allowed with such extracts is limited. This allows display of zero sized arrays and display of pointers to arrays.

```
Var.set flags[2..4]                    //display elements 2 to 4
Var.set vdblarray[2..4][i-1..i+1]      //display part of two-dimensional
                                       //array
Var.set vpchar[0..19]                  //display array at pointer 'vpchar'
Var.set (&vchar)[0..19]                //takes the location of one element
                                       //to build up an array
Var.set vpchar[0..23][0..79]           //display a two dimensional array
                                       //at the pointer
```

Extracts of arrays can be assigned or compared to members of the array.

```
Var.set flags[0..19]=0         //clears the array to 0
Var.set flags[5..9]==0         //results a non-zero number if all
                               //elements are 0
```

Assigning strings can cause two different reactions. If the string is assigned to a NULL pointer, the target function 'malloc' is called to gather memory for the string and the resulting address is assigned to the pointer variable. If the string is assigned to a non zero pointer or an array, then the contents of the string are copied over the old contents of the array.

```
Var.set vpchar = 0
Var.set vpchar = "abc"         //will call the 'malloc' function

Var.set vpchar = 0x100
Var.set vpchar = "abc"         //copy the string "abc" to location 0x100
```

Comparing a pointer or array against a string compares the contents of the string.

```
Var.Go.Till pname=="TEST"      //execute program till string equal
```

Strings used in arguments to functions are allocated on the stack.

```
Var.set strlen("abc")          //the string will reside on the stack
```

A type alone can be an expression. This is especially useful for the **Var.TYPE** command to display the layout of a structure or C++ class.

```
Var.TYPE %Multiline Tree         //displays the layout of class 'Tree'
```

Elements of unions can be accessed by indexing the union like an array. The first element of the union is accessed with index 0.

```
struct
{
     enum evtype type;
     union
     {
          struct sysevent sys;
          struct ioevent  io;
          struct winevent win;
          struct lanevent lan;
     }
     content;
}
signal;

Var.View signal.content[signal.type]
```

Structures or limited arrays may be assigned or compared with multiple members.

```
Var.set ast=(1,2,3)              //assigns the first three members
                                 // values
Var.IF point==(0,0)              //condition is true when first elements
                                 // are zero
Var.set  flg[0..2]=(1,2,3)       //assigns the first three elements
                                 // values
```

Pointers to nested C++ classes may be converted into a pointer to the most derived class of the object. If this is not possible the operation returns the regular pointer.

```
Var.set *this            //displays the "regular" object
Var.set *[this]          //displays the most derived class of the object
```

The syntax for MODULA2/PASCAL expressions has been extended for type casts and hexadecimal numbers.

```
Var.View flags[0] := 12H         //standard MODULA hexadecimal syntax
Var.View flags[0] := 0x12        //also accepted (like 'C')

Var.View CARDINAL(1.24)          //typecast like 'C': (CARDINAL) 1.23
Var.View ^CARDINAL(0x10          //typecast like 'C': (CARDINAL *) 0x10
```

Calling Functions

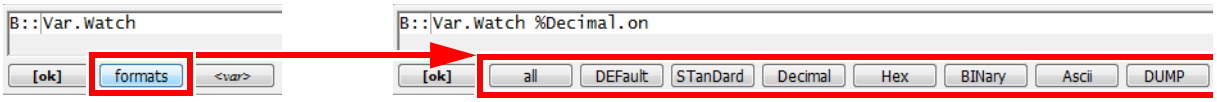
In expressions it is possible to call functions of the target. This feature should be used very carefully, as not proper working code in the target may be executed with the function call. Calling functions is only possible with the commands **Var.set** and **Var.Call**. The **Var.Call** command can be used to test a function with different parameters. If a function call fails, or was stopped by a breakpoint the original values of the CPU registers can be recalled with the **Frame.SWAP** command. The proper function call cannot be guaranteed for all processors and compiler options.

TRACE32 provides the following *<format>* parameters:

[%*<format>*] …

all
Ascii [**.on** | **.OFF**]
BINary [**.on** | **.OFF**]
Compact [**.on** | **.OFF**]
Decimal [**.on** | **.OFF**]
DEFault
DUMP [**.on** | **.OFF**]
runtimE [**.on** | **.OFF**]
Fixed [**.on** | **.OFF**]
Hex [**.on** | **.OFF**]
HIdden [**.on** | **.OFF**]
Index [**.on** | **.OFF**]
INherited [**.on** | **.OFF**]
INheritedName [**.on** | **.OFF**]
Location [**.on** | **.OFF**]
MEthods [**.on** | **.OFF**]
Multiline [.*<nesting_level>*][**.on** | **.OFF**]
Name [**.on** | **.OFF**]
Open [**.on** | **.OFF** | **.1** | **.2** | **.3** | **.4** | **.5** | **.6** | **.7** | **.8** | **.9** | **.ALL**]
PDUMP [**.on** | **.OFF**]
Recursive [**.on** | **.OFF** | **.2** | **.3** | **.4**]
SCALED [**.on** | **.OFF**]
SHOW [**.on** | **.OFF**]
SPaces [**.on** | **.OFF**]
SpotLight [**.on** | **.OFF**]
STanDard
String [**.on** | **.OFF**]
sYmbol [**.on** | **.OFF**]
TREE [**.on** | **.OFF** | **.OPEN**]
Type [**.on** | **.OFF**]
WideString [**.on** | **.OFF**]

The format parameters modify the input or output format of variables:

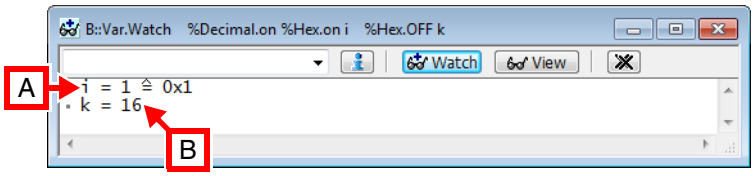


- A format parameter **affects only the variables that are listed behind it**.
- Multiple **format parameters can be combined** (e.g. **%Decimal** and **%Hex**), causing the variable to be output in multiple formats.
- Format parameters can be turned off selectively using the **.OFF** postfix.
- The **SETUP.Var** command defines the default settings. See also **DEFault** below.

For an illustration of the first three rules, see example below.

Example:

```
Var.Watch %Decimal.on %Hex.on i    %Hex.OFF k
```



A Decimal and hex for variable **i**

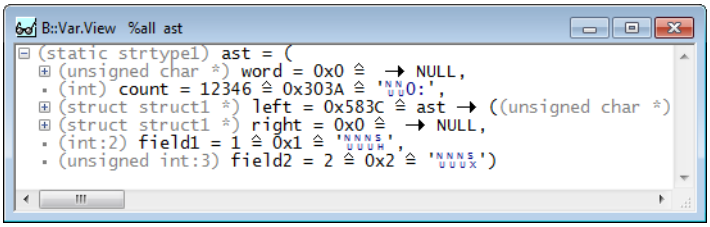
B Decimal only for variable **k**

all

all is a set of the following format options:

- **Type**
- **Decimal**
- **Hex**
- **Ascii**
- **Recursive**
- **String**
- **Index**
- **sYmbol**
- **Compact**
- **Multiline**

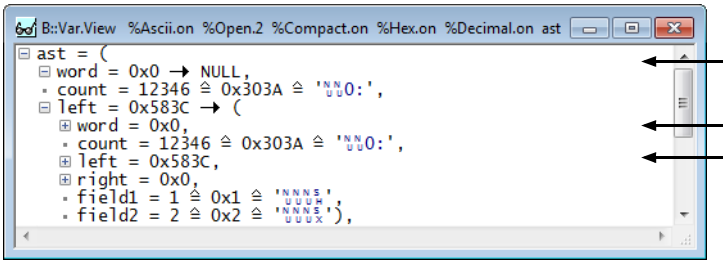
You can format the display of variables with all of these format options by using just **all**.



See also: **DEFault**, **STanDard**

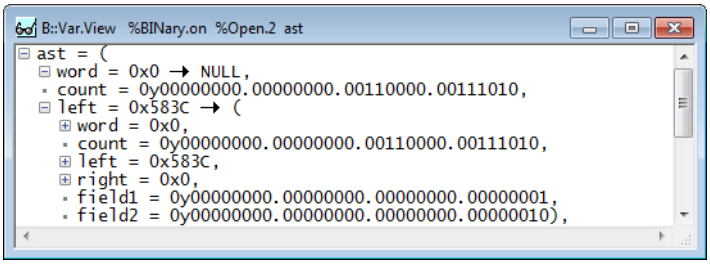
Ascii

Display of values as ASCII characters. This effects simple variables only. The **String** format can be used to display zero-terminated strings. If multiple type base formats are defined, the formats are displayed simultaneously.



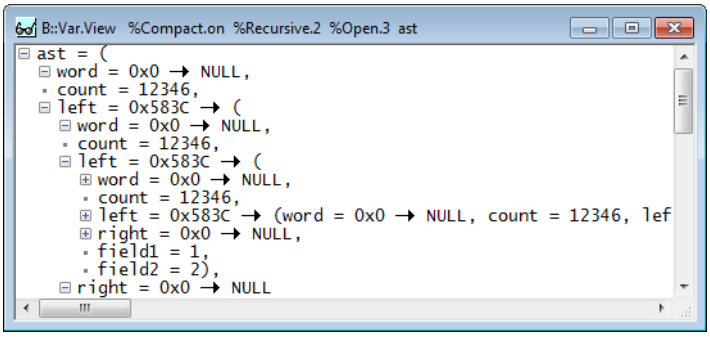
BINary

Binary display 0y...



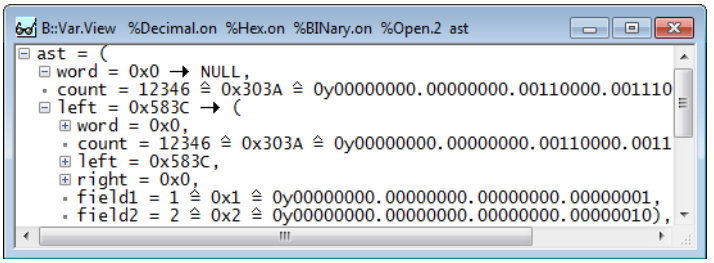
Compact

Produces a very compact output format in combination with **Multiline**.



Decimal

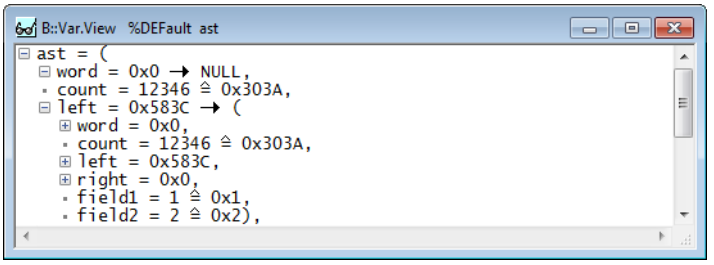
Display of values in decimal format.



DEFault

Applies all the format options that *you* have set to **ON** in the **SETUP.Var** window.

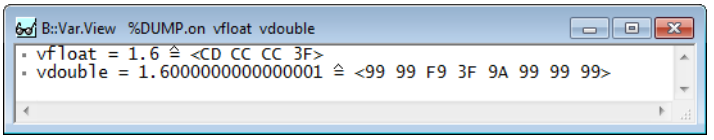
You can format the display of variables with all of these format options by using just **DEFault**.



See also: **all**, **STanDard**.

DUMP

Additional display of a short hex dump of each variable.

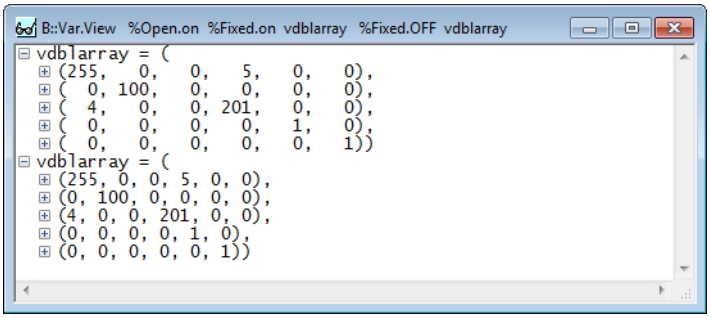


E

Access to static variables through the emulation memory. By this option global or static variables may be displayed during the real-time emulation. As this dual-port access cannot access target memory, this option allows 'save' memory accesses, as illegal pointer values cannot cause wrong accesses to the target.

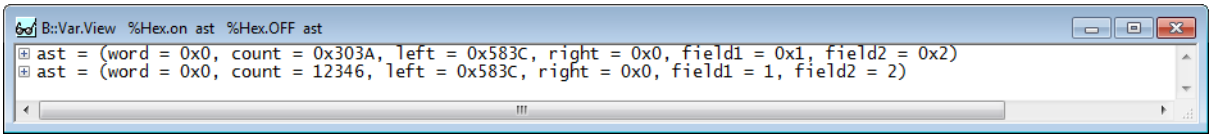
Fixed

Fixed width fields for all numeric values. Useful for two-dimensional arrays.



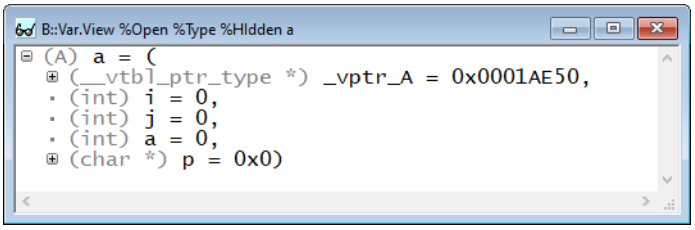
Hex

Display of values in hex format.



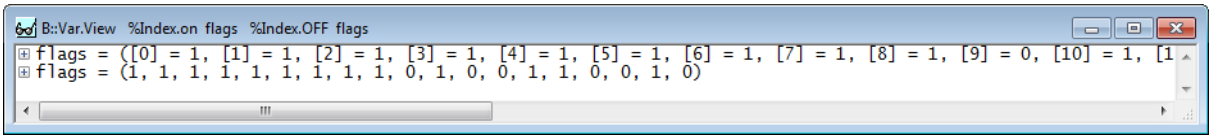
HIdden

Displays hidden members of C++ classes. 'Hidden' members are implementation specific members of nested classes. They are generated by the C++ 'cfront' preprocessor.



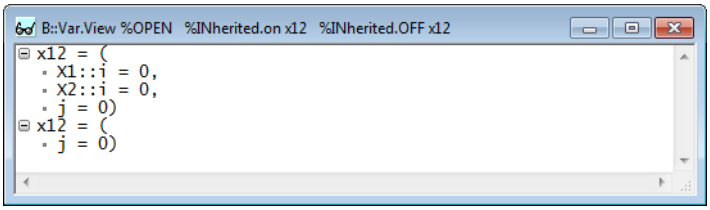
Index

Displays the index of an array element. The format is either decimal or hexadecimal. If information about the type of the index is available, the index is displayed according to this information.



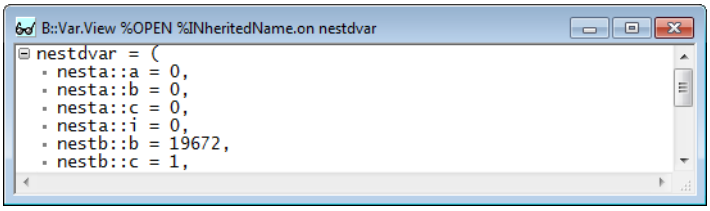
INherited

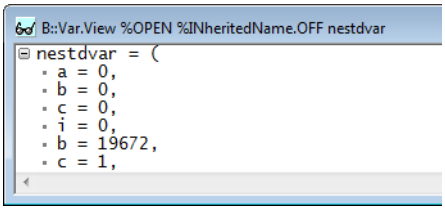
Displays members inherited from other classes (only C++).



INheritedName

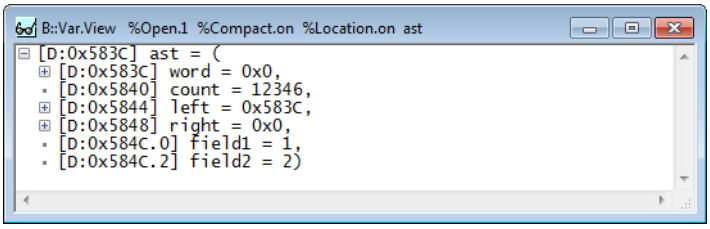
Shows or hides class names of members from inherited classes. This is useful if a class name is very long.





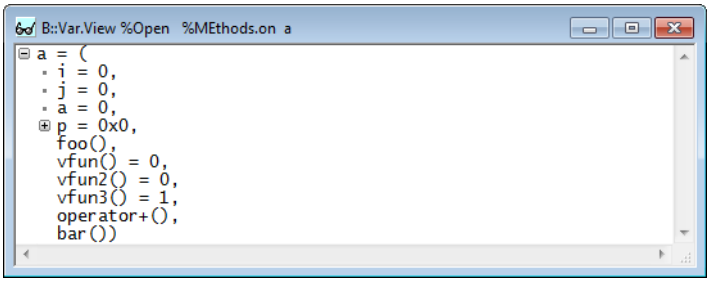
Location

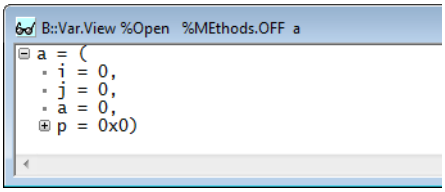
Displays the location of each variable or record element. The location can be an address or a register name.



MEthods

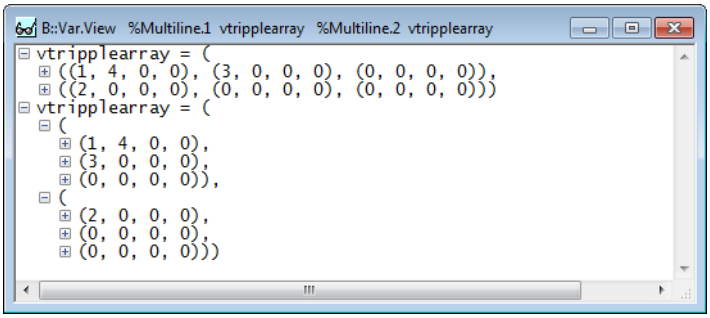
Displays the names and arguments of member functions (methods).





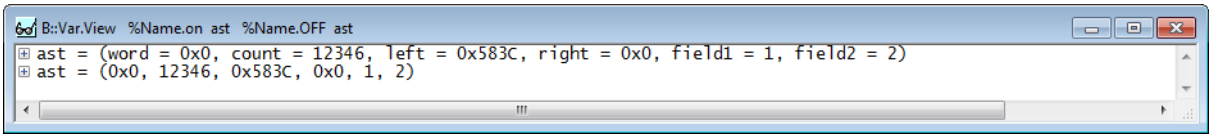
Multiline

Displays the structure elements in multiple line format. If the elements are in a multidimensional array, the numeric parameter *<nesting_level>* defines the number of levels displayed.



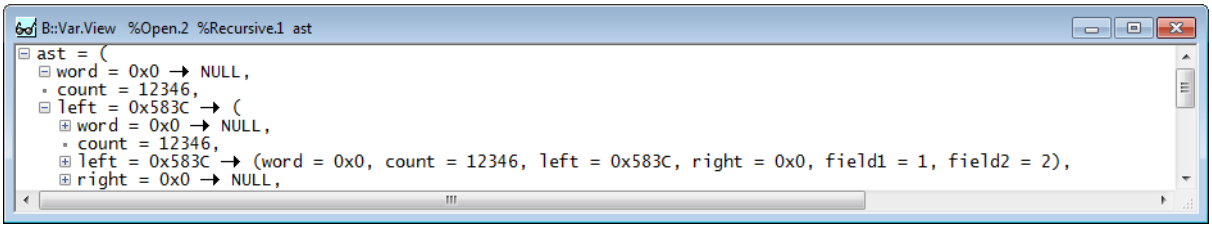
Name

Displays the name of structure elements. This is the default. It can be turned **OFF** to display more structure elements in one line.



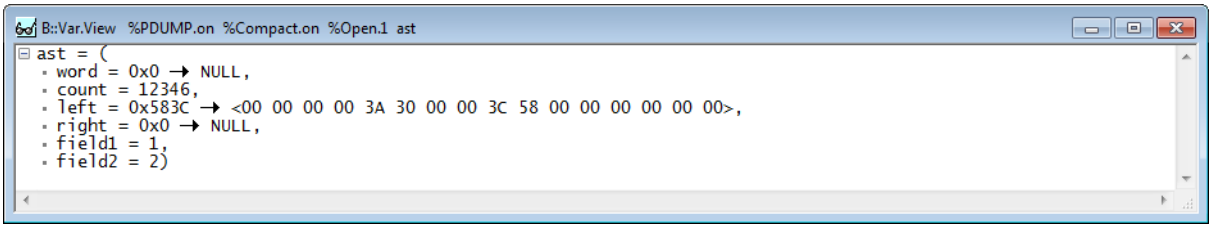
Open

Display of structures and arrays in multiple lines. The optional number defines the depth of the nesting to be displayed in multi-line mode. This option allows a clearly arranged display of multi dimensional arrays. **Open.ALL** will open nested structures respectively unions only. Pointers will not be followed.



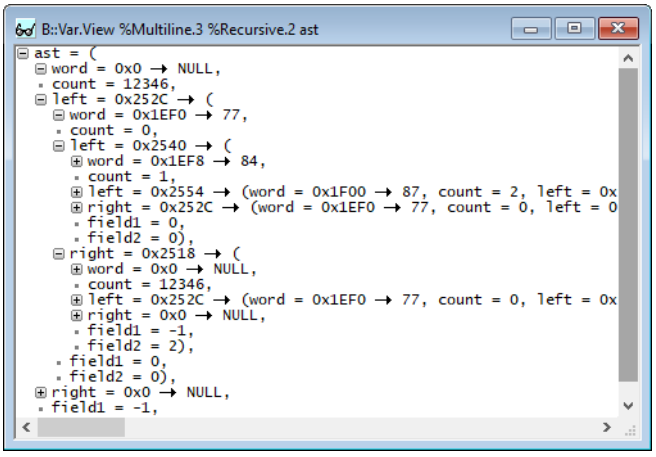
PDUMP

For pointers displays a short memory dump of the referenced memory.



Recursive

Display the contents of pointers. The optional number defines the depth of recursion to be displayed. The command **SETUP.VarPtr** defines the valid address range for pointers. The contents of pointers outside this range are not displayed.

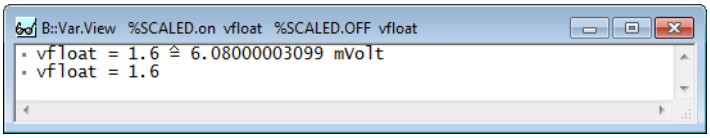


SCALED

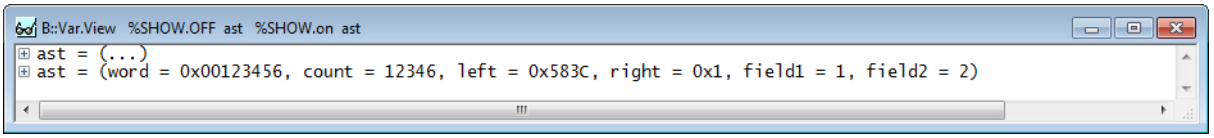
Displays the scaling information of a variable. This type of information can be added to a variable with the **sYmbol.AddInfor.Var** command.

Example:

```
;add information to a variable
;                  <variable>        <multiplier> <offset> <explanation>
sYmbol.AddInfo.Var  vfloat   Scaled     1.3          4.      " mVolt"

;display scaled variable
Var.View   %SCALED.on  vfloat   %SCALED.OFF  vfloat
```



SHOW

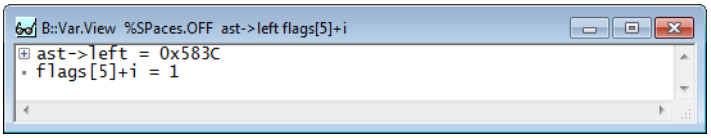


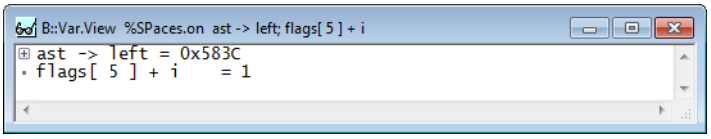
SPaces

Selects if white space characters are allowed in expressions or not. When **OFF**, expressions must be written compact and blanks separate expressions. If **on**, spaces are allowed in expressions, and only the semicolon separates expressions.

Example:

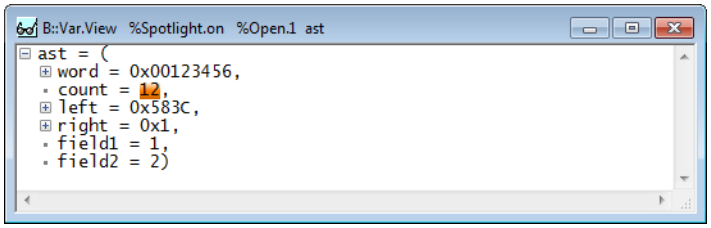
```
Var.View  %SPaces.OFF   ast->left flags[5]+i

Var.View  %SPaces.on    ast -> left; flags[ 5 ] + i
```





SpotLight

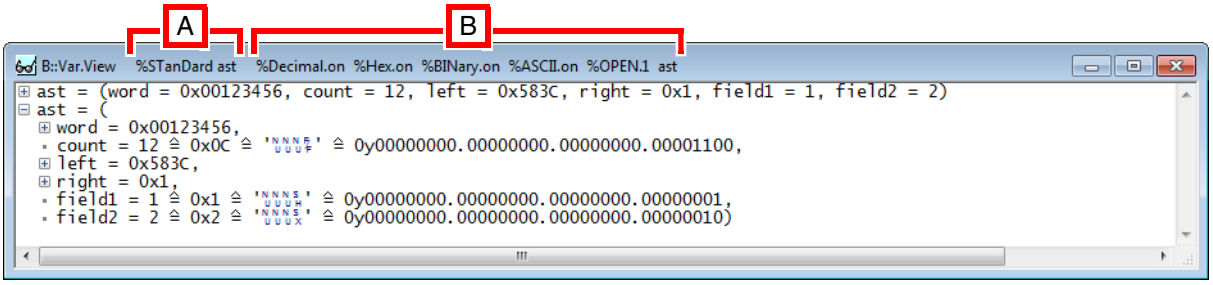
Highlights changed variable elements. This format includes the TREE format. Highlighted are only elements for the first objects of a line.



STanDard

The **STanDard** format option overrides all user-defined settings made in the **SETUP.Var** window. **STanDard** is a set of the following format options:

- **SCALED.on**
- **Name.on**
- **Compact.on**
- **TREE.on**
- **SHOW.on**
- **INherited.on**



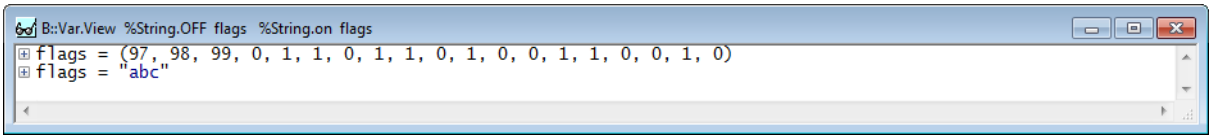
A By using just **STanDard**, you can format the display of one or more variables with all of the format options listed above.

B If you require other format options in addition to the ones included in **STanDard**, then you need to specify these format options explicitly.

See also: **all**, **DEFault**.

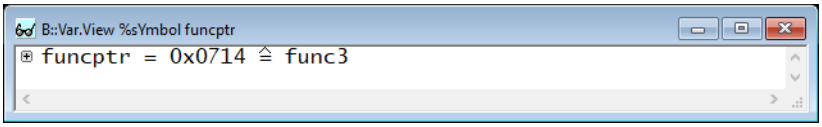
String

Displays one byte arrays or pointers to bytes as an zero-terminated ASCII string.



sYmbol

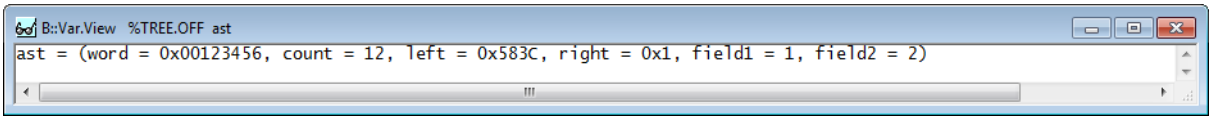
Values of pointers are displayed symbolic.

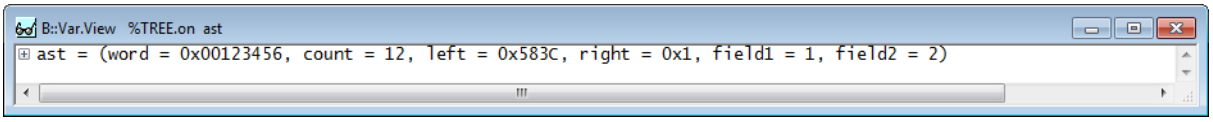


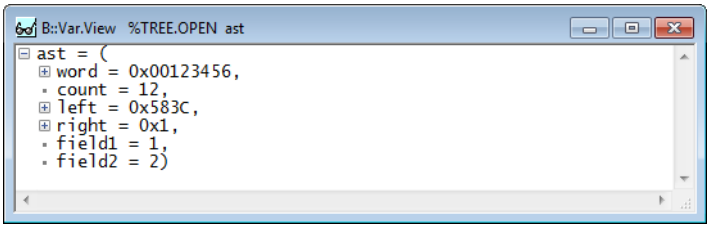
TREE

Tree view (this is the default). Allows to change some display modes for each member of a structure or array individually. This replaces the functionality of the **Open** and **Recursive** formats. Pressing the menu mouse button on the "+" or "-" sign will open a pull-down menu. This pull-down allows two choose display options for the shown elements. It is possible to show or hide the contents, display most derived classes, display the contents as ASCII string or show the first few elements of an array. **TREE.OPEN** is like **TREE.ON**, but the first element is already opened.

The TREE format is automatically selected when the SpotLight format is enabled.

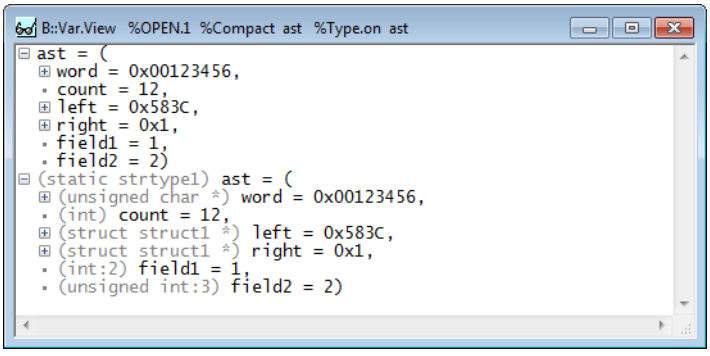






Type

Display of the variable type.



WideString

Each character is a word, e.g. for some DSPs, or unicode.

For a list of all **Var.*()** functions, see **"Var Functions"** (general_func.pdf).

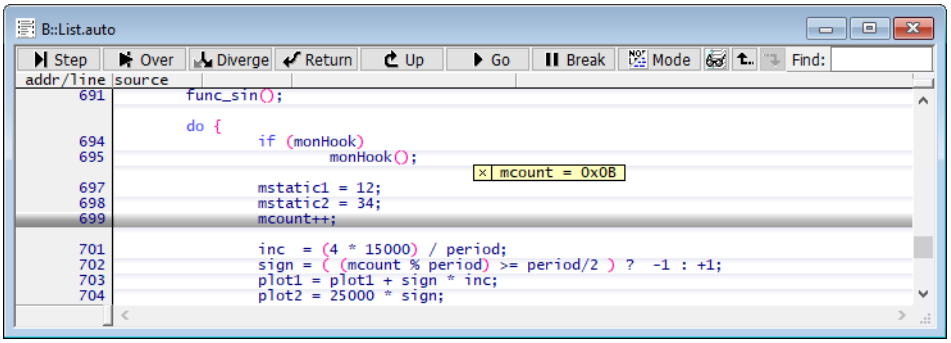
Var.AddSticker — Add variable sticker to source listing window

Format: **Var.AddSticker** *<line>* [**%***<format>*] *<var>*

Adds a variable sticker to the source listing window.

Example:

```
Var.AddSticker main\10 %Hex mcount
```



See also

- Var
- Var.set

Format: **Var.AddWatch** [**%**<*format*>] [<*variable*>] …

The specified variable is added to the top of the **Var.Watch** window. A new **Var.Watch** window is opened, if no such window exists.

<*format*> Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

See also

■ Var ■ Var.set ■ Var.View ■ Var.Watch

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

Var.AddWatchPATtern Add variables to Var.Watch window using wildcards

Format: **Var.AddWatchPATtern** [**%**<*format*>] <*symbol_pattern*> …

Adds variables to the **Var.Watch** window. For details on adding variables to a **Var.Watch** window, refer to the **Var.AddWatch** command.

<*symbol_pattern*> The wildcards '**?**' and '*****' are supported.

Example:

```
Var.AddWatchPATtern extend*
```

See also

■ Var ■ Var.set

Format:	**Var.Assign %**<*format*> <*variable*>

In contrast to **Var.set**, there is no output of the result in the message line and **AREA** window. This way you can assign values to a variable by a PRACTICE script without displaying something not of interest to be seen.

<*format*> For a description of the <*format*> parameters, click here.

See also

- Var
- Var.set

▲ 'Release Information' in 'Legacy Release History'

See also

- Var.Break.Delete
- Var.Break.direct
- Var.Break.Pass
- Var.Break.Set
- Var
- Var.set

Var.Break.Delete — Delete breakpoint on variable

[Example]

Format: **Var.Break.Delete** *<hll_expression>* [**/***<breaktype>*]

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

ProgramPass | **ProgramFail**
Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

TASK *<task_magic>* | *<task_id>* | *<task_name>*

Removes the breakpoints set to the address range specified by *<hll_expression>.*

<breaktype> — For a description of the breakpoint types and breakpoint options, see **Break.Set**.

<hll_expression> — Allows to specify the HLL expression in the syntax of the programming language used (C, C++, …).

<task_magic>, etc. — See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

Example:

```
Var.Break.Delete flags                  //deletes all breakpoints set to
                                        //the address range of variable
                                        //flags

Var.Break.Delete flags /Write           //deletes Write breakpoints set to
                                        //the address range of variable
                                        //flags
```

See also

■ Var.Break ■ Break.Delete

▲ 'Release Information' in 'Legacy Release History'
▲ 'Breakpoint Handling' in 'Training Basic Debugging'
▲ 'Breakpoint Handling' in 'Training Basic SMP Debugging'

Format: **Var.Break.direct** *<hll_expression>* [**/***<breaktype>*]

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME

Sets temporary breakpoint on address range of specified *<hll_expression>*.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.
<hll_expression>	Allows to specify the HLL expression in the syntax of the programming language used (C, C++, …).
<task_magic>, etc.	See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

See also

■ Var.Break ■ Break.direct

▲ 'Release Information' in 'Legacy Release History'

Format: **Var.Break.Pass** [*<expression>*]

When the program execution is stopped by a breakpoint, and the boolean expression is true, the program execution is automatically restarted. The feature can be cleared by entering the command without arguments.

Examples:

```
Var.Break.PASS vfloat<1.57        //automatically restart the program
                                  //execution at a breakpoint hit, if
                                  //the variable vfloat is lower then
                                  //1.57

Var.Break.Set mstatic1 /Write     //set breakpoint
Go
;…
Var.Break.PASS                    //remove the pass condition
```

The following commands shows how a condition can be directly assigned to a single breakpoint.

```
Var.Break.Set mstatic1 /Program /VarCONDition (vfloat>1.7)

Go

Var.Break.Delete mstatic1
```

See also

■ Var.Break

▲ 'Release Information' in 'Legacy Release History'

[Example]

Format: **Var.Break.Set** *<hll_expression>* [**/***<breaktype>*]

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME

Sets breakpoints to the address range specified by *<hll_expression>*. Without parameters the command opens a dialog window for setting breakpoints.

<breaktype> For a description of the breakpoint types and breakpoint options, see **Break.Set**.

<hll_expression> Allows to specify the HLL expression in the syntax of the programming language used (C, C++, …).

<task_magic>, etc. See also **“What to know about the Task Parameters”** (general_ref_t.pdf).

Example:

```
Var.Break.Set struct1                 //set Read/Write breakpoints to the
                                      //structure struct1

Var.Break.Set struct1.x /Write        //set Write Breakpoint to struct1
                                      //element x
```

See also

■ Var.Break ■ Break.Set

▲ 'Release Information' in 'Legacy Release History'

Var.Call — Call a new procedure

Format: **Var.Call** [%<*format*>] [<*expression*>]

If the expression is a function call, this function is entered and the program counter points to the first instruction of the function. The values of the CPU registers before the function call can be recalled with the **Frame.SWAP** command.

<*format*> For a description of the <*format*> parameters, click here.

Examples:

```
Var.Call func7(1.5,2.5)       //sets the PC to the start of 'func7' and
                              //pushes two floating point arguments
```

```
Var.Call (0x100)(1,2,3)       //sets the PC to 100 (hex) and pushes 3
                              //arguments
```

```
Var.Call vops+4               //assuming 'vops' is a C++ class, it sets
                              //the PC to the method function for the
                              //operator+
```

See also

■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'

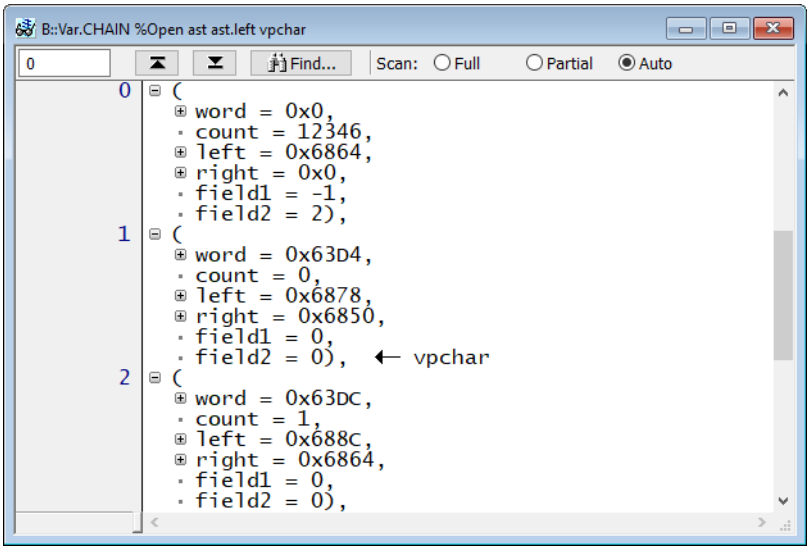
Format:	**Var.CHAIN** [**%**<*format*>] <*first*> <*next*> [<*pointer*> …]]

The first expression must be the first element of the list. The second expression specifies the pointer to the next element in the first element. The other arguments specify pointers to elements of the linked list.

<*format*> Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

Example:

```
Var.CHAIN %Open ast ast.left vpchar
```



See also

- Var
- Var.set

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

Format:	**Var.DelWatch** [*<variable>*] …

The specified formula is removed from the current **Var.Watch** window.

See also

■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'

Var.DRAW Graphical variable display

Format:	**Var.DRAW** [**%***<format>*] *<hll_expression>* [*<scale>* [*<offset>*]] [**/***<option>*]
<option>:	*<draw_option>* | **Element** *<number>* | **XY** | **YX** | **Alternate** *<number>*
<draw_option>:	**Vector** | **Points** | **Steps** | **Impulses** | **LOG**

Displays the contents of an array or a structure element graphically. The **Data.DRAW** command can be used to display memory contents graphically.

<draw_options>	**Vector:** Connects the dots for the data values by vectors (default). **Points:** Displays each data value as a dot. **Steps:** Connects the dots for the data values by steps. **Impulses:** Draws each data value as a single pulse. **LOG:** Displays the data values in a logarithmic format.
<format>	Using the *<format>* parameters, you can modify the display in various ways. For a description of the format parameters, see **"Display Formats"**, page 12.
<hll_expression>	Allows to specify the HLL expression in the syntax of the programming language used (C, C++, …).
<offset>	Offset of y-axis (floating point). Default: **0.0** See example.
<scale>	Units per pixel of y-axis (floating point). E.g. a signal has a max. height of 50 units shall be visualized window that has a height of 400 pixels: 50 units divided by 400 pixels = 0.125 By default the scale factor is set so that the window displays the complete possible value range for the selected variable. See example.

Alternate *<number>*
Split the array in *<number>* graphs.
<number>=2
first graph display even elements
second graph displays odd element.
<number>=3
first graph displays element 0, n, 2n, …
second graph displays 1, n+1, 2n+1, …
third graph display 2, n+2, 2n+2, … See example.

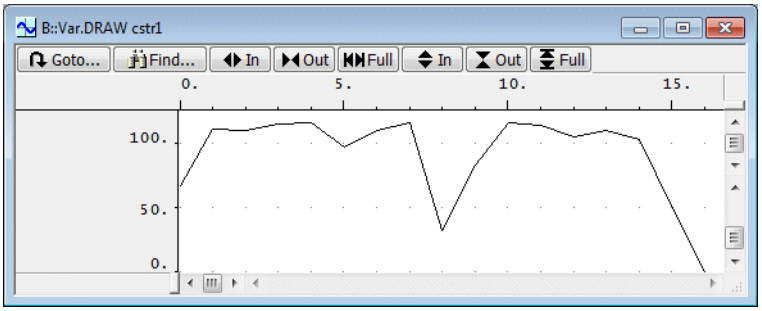
Element *<number>*
Specify the structure component to be displayed graphically. See example.

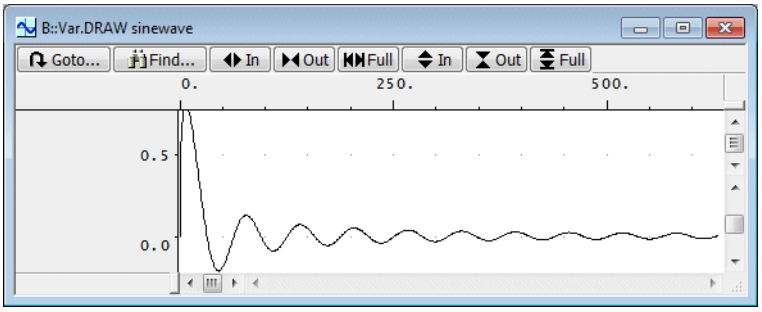
XY
Allows to display two arrays graphically.
The contents of the first array is used as x-axis.
The contents for the second array is used as y-axis. See example.

YX
Allows to display two arrays graphically.
The contents of the first array is used as y-axis.
The contents for the second array is used as x-axis. See example.

Example for arrays:

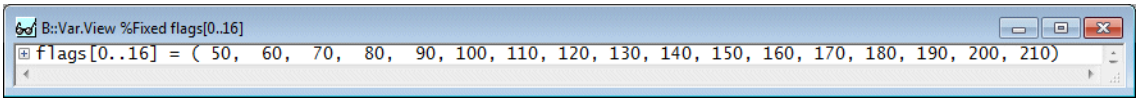
```
Var.DRAW cstr1

Var.DRAW sinewave
```

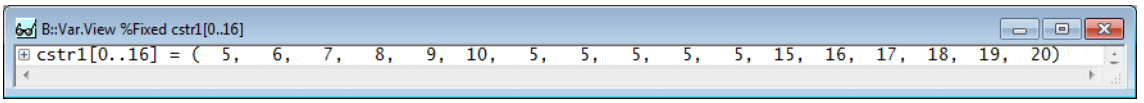


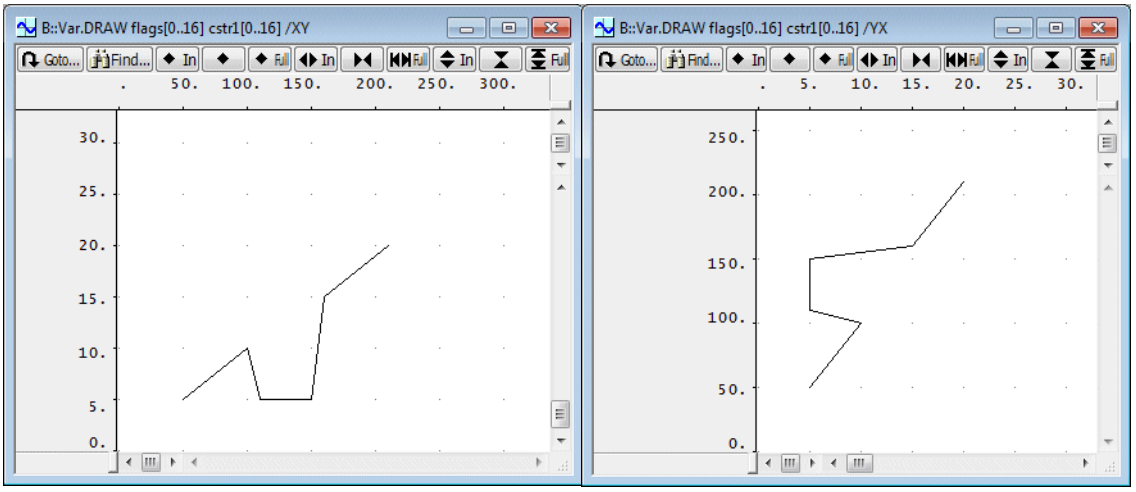


Example for two interdependent arrays:

```
Var.DRAW flags[0..16] cstr1[0..16] /XY

Var.DRAW flags[0..16] cstr1[0..16] /YX
```

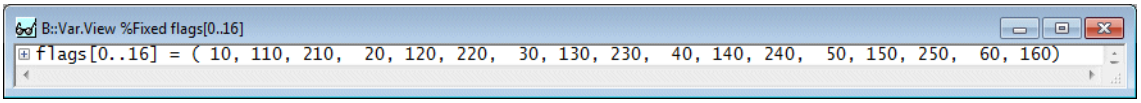


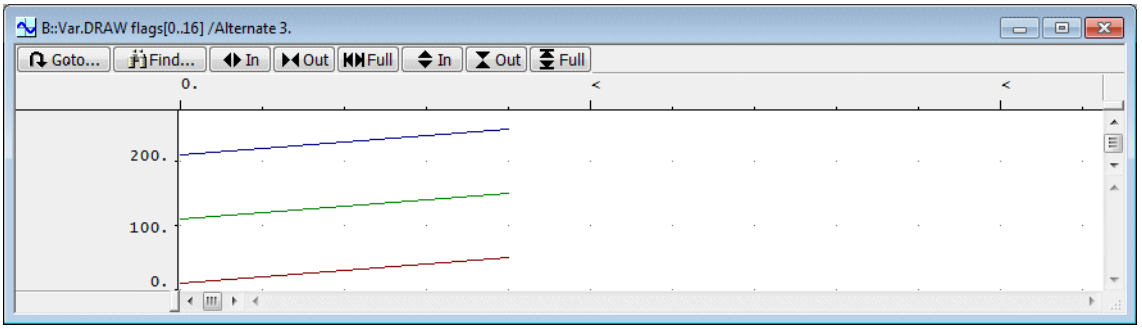




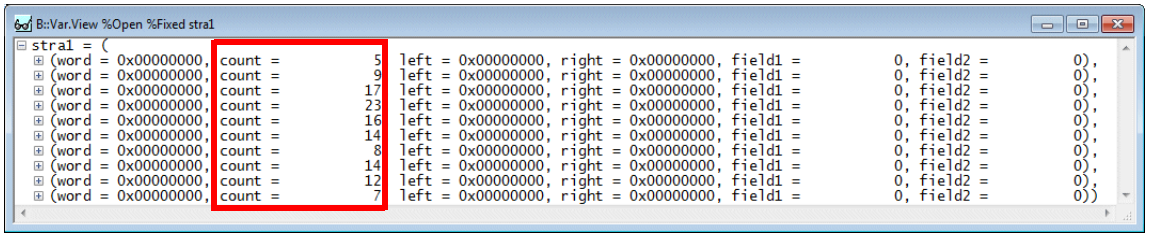
Example for array split:

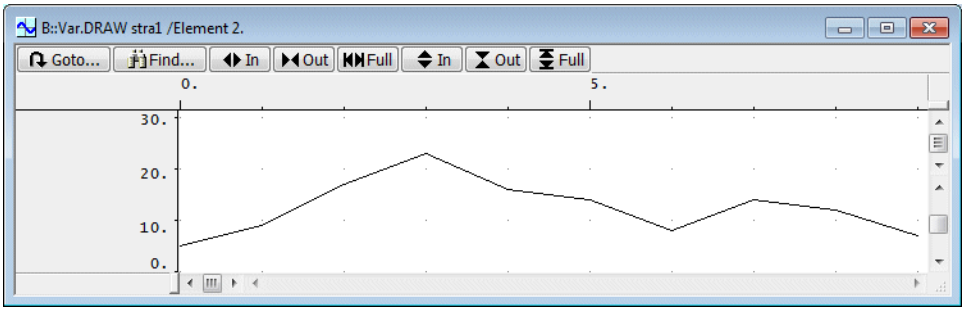
```
Var.DRAW flags[0..16] /Alternate 3.
```





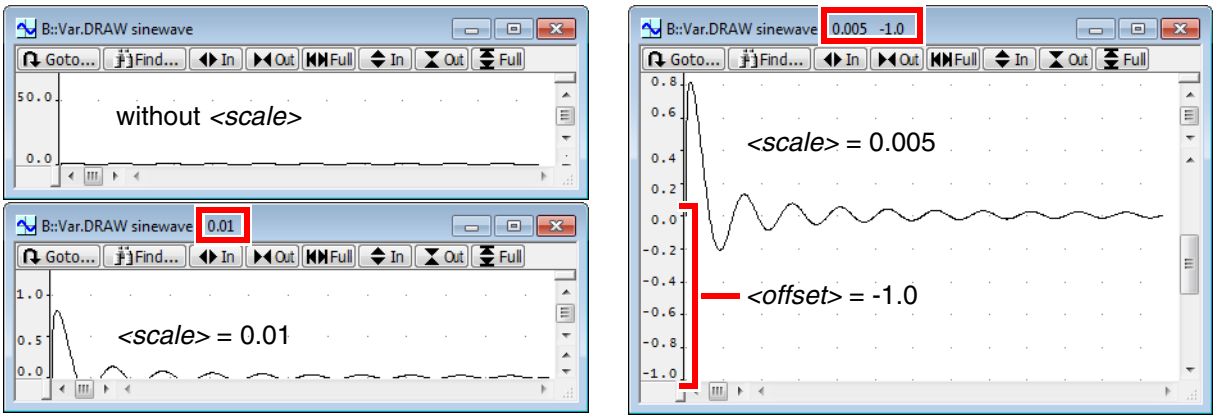
Example for structure element:






```
Var.DRAW stra1 /Element 2.
```

Example for <scale> and <offset>:



See also

- Var
- Var.PROfile
- Var.set
- <trace>.DRAW
- Data.DRAW
- Data.DRAWFFT
- Data.DRAWXY
- Data.IMAGE

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

Var.DRAWXY

Graphical variable display

Format:	**Var.DRAWXY** [**%***<format>*] *<hll_expression>* *<hll_expression>* [**/***<option>*]
<option>:	*<draw_option>* | **Element** *<number>* | **YX** | **Alternate** *<number>*
<draw_option>:	**Vector** | **Points** | **Steps** | **Impulses** | **LOG**

Displays the contents of two arrays graphically in one single window. The elements of the first array correspond to the X-axis and the elements of the second array to the Y-axis. Please refer to the **Var.DRAW** command for a description of the parameters and options.

Example:

```
Var.DRAWXY array1 array2
```

See also

- Var
- Var.set

Format:	**Var.DUMP** [**%***<format>*] [[**&**]*<variable>*] … [**/***<option>*]
<format>:	**NoHex** | **NoAscii** **Byte** | **Word** | **Long** | **Quad** | **TByte** | **HByte** **BE** | **LE** **PC8**
<option>:	**Orient** **NoOrient** **COLumns** [*<columns>*] **Mark** *<break>* **Flag** *<flag>* **Track** **CACHE**
<flag>:	**Read** | **Write** | **NoRead** | **NoWrite**
<break>:	**Program** | **Hll** | **Spot** | **Read** | **Write** | **Alpha** | **Beta** | **Charly**

The first expression defines the address of the dump. All following expressions are treated as pointers and marked in the dump.

<format> For a description of the *<format>* parameters, see **"Display Formats"**, page 12.

<option> For a description of the options, see **Data.dump**.

See also

■ Var ■ Var.set ■ SETUP.DUMP

▲ 'Release Information' in 'Legacy Release History'

Format: **Var.Eval** [**%**<*format*>] <*hll_expression*>

Evaluates a high-level language expression. The result can be returned with the **EVAL()** functions.

<*format*> Use the <*format*> parameters to export the variables in the desired format. For a description of the <*format*> parameters, click here.

See also

- Var
- Var.set
- Eval

Var.EXPORT Export variables in CSV format to file

Format: **Var.EXPORT** <*file*> [**%**<*format*>] [<*variable*>] … [**/Append**]

Exports variables in CSV format (Comma-Separated Values) for import to other applications. Existing file contents are overwritten if the file already exists.

<*file*> If path and file name are substituted for a comma, the default file name `t32.lst` is used. The file is exported to the current working directory (see **PWD** command and "**Path Prefixes**").

<*format*> Use the <*format*> parameters to export the variables in the desired format. For a description of the <*format*> parameters, click here.

Example:

```
;Export as CSV and include variable type, location, and index.
;The variables to be exported are 'flags' and 'ast'.
Var.EXPORT ~~~\export.csv %Type %Location %Index flags ast

;Optional step: display the file
TYPE ~~~\export.csv
```

NOTE:	If a text line created by this command exceeds a few thousand characters, it is clipped. To handle long lines, consider using the format specifier **%Multiline**.

See also

- Var
- Var.set
- Var.WRITE
- PRinTer.EXPORT

Format: **Var.FixedCHAIN** [%*<format>*] *<first>* *<next>* [*<pointer>* …]]

The first expression must be the first element of the list. The second expression specifies the pointer to the next element in the first element. The other arguments specify pointers to elements of the linked list. The format parameters can modify the display in various ways. Format parameters are described at the beginning of this chapter.

Example:

```
Var.FixedCHAIN %Location %Multiline ast ast.left vpchar
```

See also

- Var
- Var.set

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

Var.FixedTABle Display table

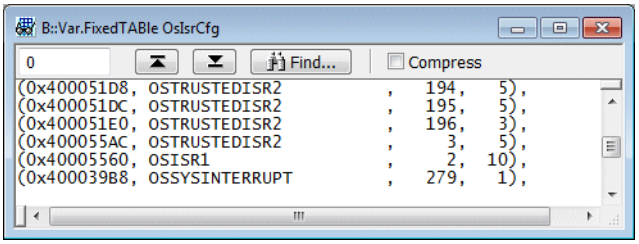
Format: **Var.FixedTABle** [%*<format>*] *<array>* {*<index>*} {*<pointer>*}

Displays the first expression as an array. The command is intended for arrays of structures or arrays of pointers to structures. The extra arguments are displayed as pointers or indexes to that array.

<format> Use the *<format>* parameters to display the variables in the desired format. For a description of the *<format>* parameters, click here.

Example 1:

```
Var.FixedTABle OsIsrCfg
```

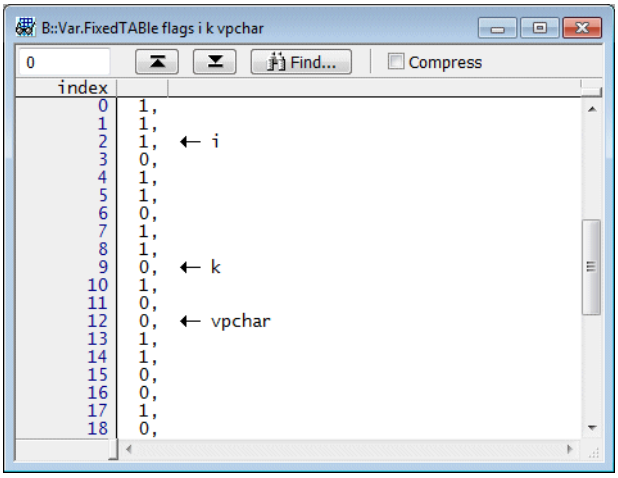


Example 2: The following command sequence allows you to save the variable content to a *.csv file.

```
PRinTer.FILE OsIsrCfg.csv  CSV        ; specify file name and select CSV
                                      ; as output format

WinPrint.Var.FixedTABle OsIsrCfg      ; WinPrint. redirects the command
                                      ; output to specified file
```

Example 3:

```
Var.FixedTABle flags i k vpchar       ; i and k are array indices
                                      ; vpchar is a pointer to the array
```



See also

- Var
- Var.set

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

See also

- Var.Go.Back
- Var.Go.Change
- Var.Go.direct
- Var.Go.Till
- Var
- Var.set

▲ 'Release Information' in 'Legacy Release History'

Var.Go.Back Re-run program backwards until variable access (CTS)

[Example]

Format: **Var.Go.Back** *<expression>* [**/***<breaktype>*]

<breaktype>:

Program | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME

Re-runs the recorded program flow backwards until the specified variable is accessed.

<breaktype>	For a description of the breakpoint types and breakpoint options, see **Break.Set**.
<task_magic>, etc.	See also **"What to know about the Task Parameters"** (general_ref_t.pdf).

Example:

```
CTS.GOTO -1209874.

Var.Go.Back flags /Write     //run program backwards until a write access
                             //to the variable flags happens
```

See also

■ Var.Go ■ Var.Go.direct

▲ 'Release Information' in 'Legacy Release History'

Var.Go.Change — Real-time emulation till expression changes

Format: **Var.Go.Change** *<expression>*

The emulation is started and after each emulation stop the given expression is evaluated. If the expression has not changed, the emulation is started again.

Example:

```
Var.Break.Set flags /Write
Var.Go.Change flags              // starts the emulation and restarts,
                                 // if the array flags has not changed
```

See also

■ Var.Go ■ Var.Go.direct

▲ 'Release Information' in 'Legacy Release History'

Format: **Var.Go.direct** *<expression>* [**/***<breaktype>*]

<breaktype>: **Program** | **ReadWrite** | **Read** | **Write**

Onchip | **HARD** | **SOFT**

ProgramPass | **ProgramFail**

MemoryReadWrite | **MemoryRead** | **MemoryWrite**
RegisterReadWrite | **RegisterRead** | **RegisterWrite**
VarReadWrite | **VarRead** | **VarWrite**
DATA[**.Byte** | **.Word** | **.Long**] *<value>* …

Alpha | **Beta** | **Charly** | **Delta** | **Echo**

WATCH | **BusTrigger** | **BusCount**
TraceEnable | **TraceData** | **TraceON** | **TraceOFF** | **TraceTrigger**

Spot
DISable | **DISableHIT** | **DeleteHIT** | **NoMark** | **EXclude**
TASK *<task_magic>* | *<task_id>* | *<task_name>*
MACHINE *<machine_magic>* | *<machine_id>* | *<machine_name>*
CORE *<number>*
COUNT *<value>*
CONDition *<expression>* [**/AfterStep**]
VarCONDition *<hll_expression>* [**/AfterStep**]
CMD *<command_string>*
RESUME

Sets breakpoints to the given variable or structure element and starts the emulation. The breakpoints are removed after the emulation has stopped again.

<breaktype> For a description of the breakpoint types and breakpoint options, see **Break.Set**.

<task_magic>, etc. See also **“What to know about the Task Parameters”** (general_ref_t.pdf).

Examples:

```
Var.Go.direct flags              // run till any element of 'flags' is
                                 // accessed

Var.Go.direct vfloat /Write      // run till a write to 'vfloat' occurs
```

See also

■ Var.Go ■ Var.Go.Back ■ Var.Go.Change ■ Var.Go.Till

Var.Go.Till Real-time emulation till expression true

Format: **Var.Go.Till** *<expression>*

The emulation is started and after each emulation stop the given boolean expression is evaluated. If the expression is false, the emulation is started again.

Example:

```
Var.Break.Set vfloat /Write      //starts the emulation and restarts, if
Var.Go.Till vfloat<=1.57         // the value
                                 //of vfloat is larger than 1.57
```

See also

■ Var.Go ■ Var.Go.direct

▲ 'Release Information' in 'Legacy Release History'

Format: **Var.IF** *<hll_condition>*

Executes the next command or command block only if the specified *<hll_condition>* is true. The **Var.IF** command is the counterpart to the PRACTICE **IF** instruction and can also be combined with the **ELSE** command.

<hll_condition> Allows to specify the condition in the syntax of the programming language used (C, C++, ...).

Example:

```
Var.IF stra2[1][0].pastruct5[0]==25
(
  PRINT "Initialization of stra2[1][0].pastruct5[0] failed."
)
```

See also

■ Var ■ Var.set

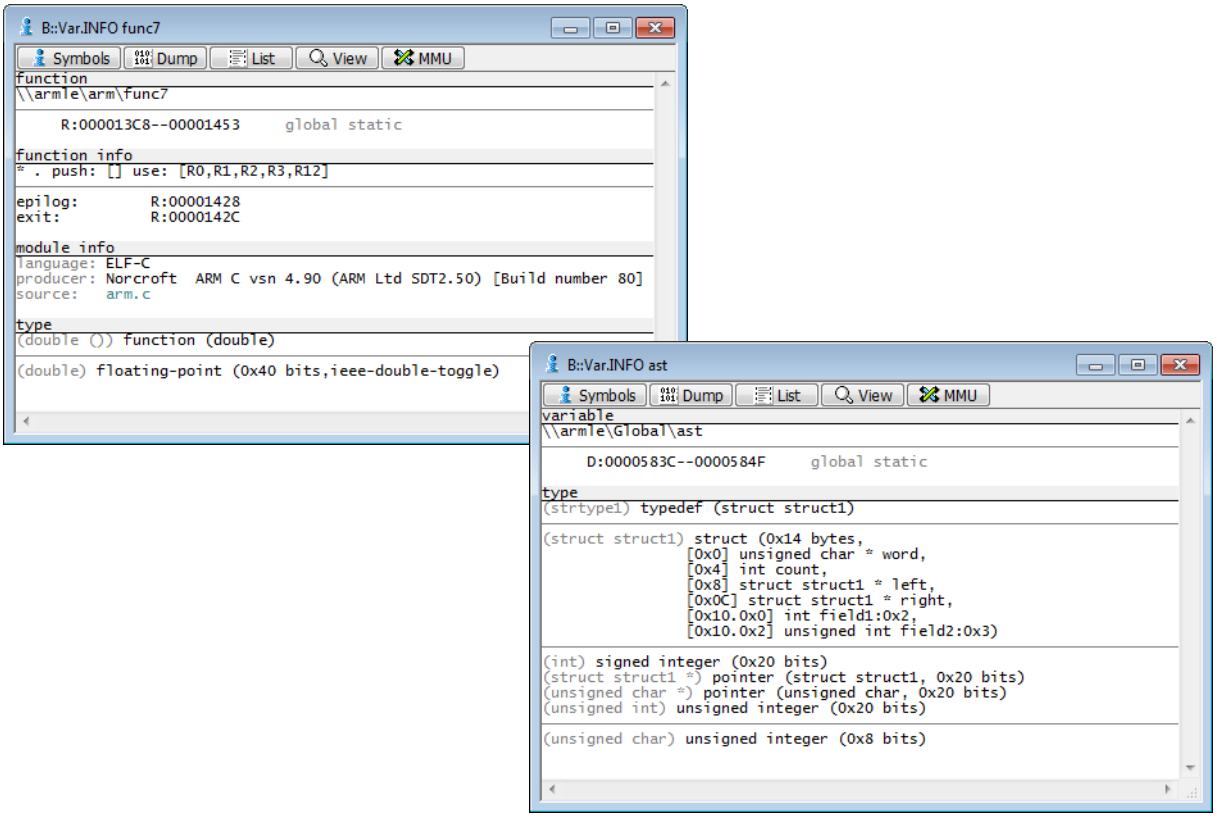
▲ 'Release Information' in 'Legacy Release History'

Format: **Var.INFO** *<variable>* | *<expression>*

Displays all information available for a symbol or the type of an HLL expression. The physical layout of HLL variables is displayed too.

Example:

```
;display information about the HLL expression func7
Var.INFO func7

;display information about the structure ast
Var.INFO ast
```



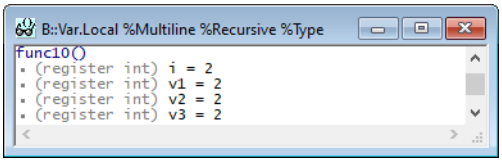
See also

- Var
- Var.set
- sYmbol.INFO

▲ 'Release Information' in 'Legacy Release History'
▲ 'The Symbol Database' in 'Training Source Level Debugging'

Format:	**Var.Local** [**%**<*format*> …]

Display of all local variables of a function. When using Pascal, the local variables of the superior functions are displayed too. The format parameters can modify the display in various ways. Format parameters are described at the beginning of this chapter.



The variables can be modified by clicking with the mouse.

See also

- Var
- Var.Ref
- Var.set
- Var.View

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

[Examples]

Format:	**Var.LOG** [**%***<format>*] {*<variable>*} {**/***<option>*}
<option>:	**ONBREAK** **ONSPOT** **ONTIME** *<time>* **TImestamp** **Changes** **AREA** *<name>*

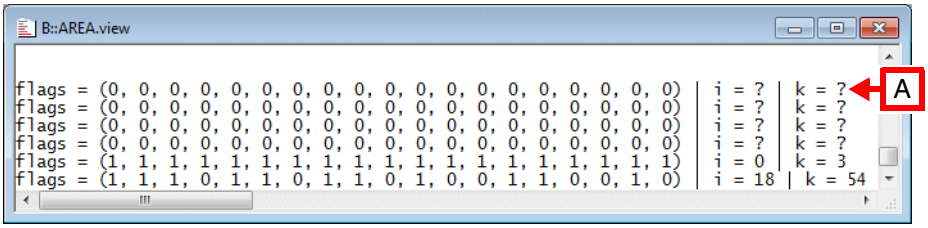
By default the specified variables are logged to the TRACE32 message **AREA** whenever the program execution is stopped. If a syntactical error is made, just a warning is received. This allows the definition of a log showing local variables not valid in the current program context.

<format> Use the *<format>* parameters to format the variables as required. For a description of the *<format>* parameters, click here.

<options>	
ONBREAK	Updates log each time the program execution is stopped. This is the default.
Changes	A log is only made when the variables have changed their value.
AREA *<name>*	Selects a different **AREA** for the logging.
ONSPOT	Update the log whenever a breakpoint specified with the Action **Spot** is hit and each time the program execution is stopped.
ONTIME	Updates the log in a fixed time interval. This option requires **run-time memory access** to the variables.
TImestamp	Adds timestamps (absolute and relative) to the log. Mainly used together with **ONTIME** option.

Example 1: **Var.LOG** without a variable definition ends the logging.

```
Var.LOG flags i k                       ; i and k are local variables
                                        ; if they are not valid in the
                                        ; current context, a warning is
                                        ; given by TRACE32 PowerView

AREA.view                               ; open TRACE32 message AREA window

;…                                      ; perform your test

Var.LOG                                 ; switch off the logging
```



A If a variable is not valid in the current context, a ? is displayed.

Example 2: Log variables to file. Remember that inline comments for **Var.*** commands must start with **//**.

```
AREA.Create my_log                      ; create my_log area

AREA.view my_log                        ; display my_log area

AREA.OPEN my_log loglist.lst            ; save all entries to area in file

Var.LOG flags ast k /AREA my_log        // enable variable log

AREA.CLOSE my_log                       ; stop saving the entries to area
                                        ; in file

Var.LOG                                 // end variable logging

TYPE loglist.lst                        ; display contents of file
```

Example 3: Log variables every second.

```
; create area named my_log
AREA.Create my_log

; display area named my_log
AREA.view my_log

; log variables every second via the run-time memory access
; log them with relative and absolute timestamp
; log only changes
Var.LOG %E flags ast /AREA my_log /ONTIME 1.0s /TImestamp /Changes

Go

;…

Break

; end variable logging
Var.LOG
```

See also

■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'

Var.NEW — Creates a TRACE32-internal variable

Format: **Var.NEW** [*<type>*] *<name>* (deprecated)
Use Var.NEWLOCAL or Var.NEWGLOBAL instead.

See also

■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'

Format:	**Var.NEWGLOBAL** [*<var_type>*] ****<*variable_name*>
<var_type>:	**int** | **char** | **long** | **short**

Creates a TRACE32-internal variable of the specified variable type and registers the variable on the *global* PRACTICE stack frame.

Global TRACE32-internal variables are visible everywhere. They are not erased when the declaring file or block ends. TRACE32-internal variables can be used to write complex PRACTICE programs which deal with expressions of the target high level language (HLL).

<var_type>	The following commands provide an overview of the supported variable types: • The **sYmbol.List.BUILTIN** command lists the supported built-in variable types. • The **sYmbol.List.Type** command lists variable types available after a target program has been loaded.
<variable_name>	The TRACE32-internal variables must begin with a '****' character, as opposed to global PRACTICE macros (variables), which begin with a '**&**' and are created with **GLOBAL**.

Example: A character array is created on the *global* PRACTICE stack frame. The array member `[5]` is initialized, and its value is printed to the TRACE32 message line using the **Var.STRing()** function. For more examples, refer to **Var.NEWLOCAL**.

```
PMACRO.list ;View the PRACTICE stack

;Create a TRACE32-internal variable: the character array \myStr
Var.NEWGLOBAL  char[10][128] \myStr

Var.set \myStr[5]="hello"    //Initialize array member [5]

PRINT Var.STRing(\myStr[5])  ;Show value in message line
```

Remember that inline comments for **Var.*** commands must start with **//**. **PRINT** is a command, whereas **Var.STRing()** is not a command, but a function(). Therefore, the above inline comment may start with a semicolon **;**.

See also

■ Var.NEWLOCAL ■ Var ■ Var.set

▲ 'In This Document' in 'General Function Reference'
▲ 'PRACTICE Script Structure' in 'PRACTICE Script Language User's Guide'
▲ 'Release Information' in 'Legacy Release History'

[Examples]

Format:	**Var.NEWLOCAL** [*<var_type>*] *****<*variable_name*>
<var_type>:	**int** | **char** | **long** | **short** | ...

Creates a TRACE32-internal variable of the specified variable type and registers the variable on the *local* PRACTICE stack frame.

Local TRACE32-internal variables exist inside the declaring block and are erased when the block ends. They are visible inside their blocks, sub-blocks (e.g. **IF**..., **RePeaT**..., **WHILE**..., etc.), subroutines (**GOSUB**...**RETURN**), and sub-scripts (**DO**...**ENDDO**).

TRACE32 internal variables can be used to write complex PRACTICE programs which deal with expressions of the target high level language (HLL).

<var_type>	The following commands provide an overview of the supported variable types: • The **sYmbol.List.BUILTIN** command lists the supported built-in variable types. • The **sYmbol.List.Type** command lists variable types available after a target program has been loaded.
<variable_name>	The debugger-internal HLL variables must begin with a '\' character, as opposed to local PRACTICE macros (variables), which begin with a '&' and are created with **LOCAL**.

Example 1:

This script shows how to create, initialize, and view the local TRACE32-internal variables. In addition, the example shows how to print their return values to the TRACE32 message line using the **Var.*()** functions (See "**Functions**").
By double-clicking a local TRACE32-internal variable in the **Var.View** window, you can change its parameter on the fly. Simply type the desired parameter in the TRACE32 command line.

```
PMACRO.list ;View the PRACTICE stack

;Create some TRACE32-internal variables: integer \val1, float \val2, and
;character array \myStr on the local PRACTICE stack frame
Var.NEWLOCAL  int   \val1
Var.NEWLOCAL  float \val2
Var.NEWLOCAL  char[10][128] \myStr

;Open the Var.View window to display these TRACE32-internal variables
Var.View %all \val1 \val2 \myStr

;Initialize the TRACE32-internal variables
Var.set \val1=0x42
Var.set \val2=197.25
Var.set \myStr[5]="Hello world!"

;Print the TRACE32-internal variables to the message bar
PRINT %Hex "0x" Var.VALUE(\val1)  ;integer
PRINT Var.FVALUE(\val2)           ;float
PRINT Var.STRing(\myStr[5])       ;string
```

Example 2:

The HLL array `flags` is manipulated based on the HLL array `vdiarray`. Remember that inline comments for **Var.*** commands must start with **//**.

```
Var.NEWLOCAL int \i //Create a TRACE32-internal variable: integer \i

Var.set \i=0        //Initialize the TRACE32-internal variable

;Open a window to watch the HLL arrays flags, ast, and vdiarray
;as well as the TRACE32-internal variable \i
Var.Watch %SpotLight flags ast vdiarray \i

;Manipulate the HLL array flags based on the HLL array vdiarray
Var.WHILE \i<sizeof(vdiarray)
    Var.set flags[\i++]=3

Var.IF \i=sizeof(vdiarray)
    PRINT "end of loop reached"
```

Example 3:

This script focusses on TRACE32-internal array variables.

```
;Create a TRACE32-internal variable: a character array
Var.NEWLOCAL char[6][20] \string_array

;Open a window to watch \string_array
Var.Watch %SpotLight \string_array

;Initialize the character array
Var.set     \string_array[0]="flashtest0" //is shown in message line
Var.ASSIGN  \string_array[2]="element2"   //is NOT shown in message line

PRINT       Var.STRing(\string_array[0])  //show value in message line

Var.IF \string_array[2][0]!='\0'
       PRINT Var.STRing(\string_array[2])
```

See also

■ Var.NEWGLOBAL ■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'

Var.OBJECT Pretty printing for C++ objects

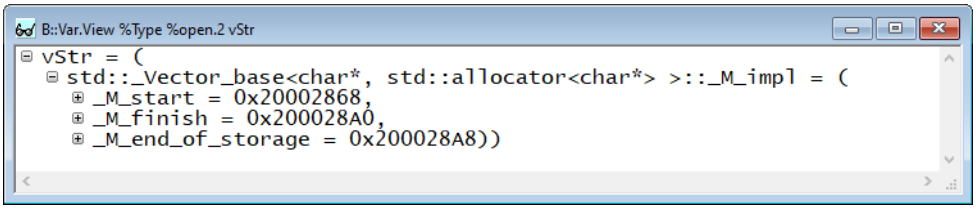
Format: **Var.OBJECT** [**%**<*format*>] [<*variable*>] …

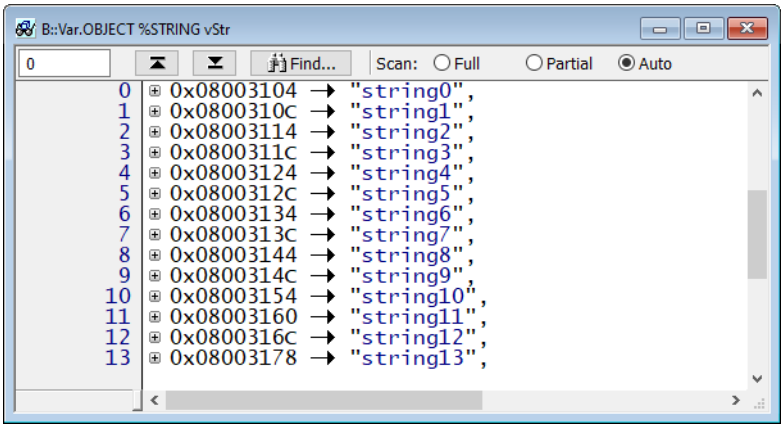
The command **Var.OBJECT** can be used together with the **sYmbol.AddInfo** command or with simulator based target calls for pretty printing of C++ objects.

Example:

```
; define the vector type information
; TABLE type
; base: _M_impl._M_start
; size = _M_impl._M_finish - _M_impl._M_start
sYmbol.AddInfo.Type std::vector<? TABLE \
"(#o)._M_impl._M_finish-(#o)._M_impl._M_start" "(#o)._M_impl._M_start"
Var.OBJECT %String %Type vStr
```

The screen shots below show the standard display with the **Var.View** command and the pretty printing with **Var.OBJECT**.





An example for displaying STL container running on the TRACE32 Instruction Set Simulator for Arm can be found under `~~/demo/etc/stl`

```
CD.DO ~~/demo/etc/stl/demo_stl_arm.cmm
```

See also

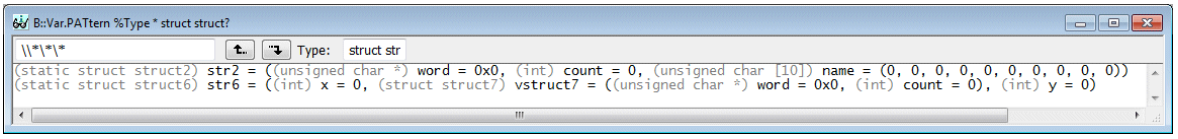
- Var
- Var.set

Format: **Var.PATtern** [%<*format*>] [<*symbol_pattern*>] [<*type_pattern*>]

Display variables allowing the wildcard ? and * in the variable name and the variable type.

Examples:

```
Var.PATtern target* struct            ; Display all variables whose name
                                      ; begins with "target" and which
                                      ; are of the type struct

Var.PATtern jpeg*                     ; Display all variables whose name
                                      ; begins with "jpeg"

Var.PATtern %Type * struct struct?    ; Display all variables which
                                      ; are of the type struct struct?
```



See also

- Var
- Var.set

▲ 'Release Information' in 'Legacy Release History'

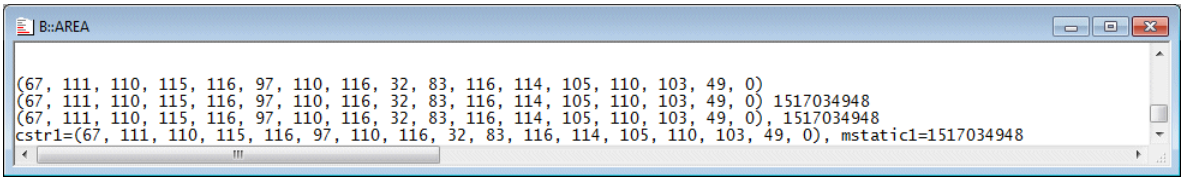
Format:	**Var.PRINT** [**%CONTinue**] {[**%**<*format*>] [<*variable*>|<*data*>]}

The specified formula is interpreted and the according values are displayed in the message line or the current output **AREA** window. The command is an HLL version of the **PRINT** command.

CONTinue	Adds the string to the current output line in the selected **AREA** window or message line without inserting a newline character.
<*format*>	Use the <*format*> parameters to print the variables in the desired format. For a description of the <*format*> parameters, click here.

Example:

```
Var.PRINT cstr1

Var.PRINT cstr1 " " mstatic1

Var.PRINT cstr1 ", " mstatic1

Var.PRINT "cstr1=" cstr1 ", mstatic1=" mstatic1
```



See also

- Var
- Var.set
- Var.WRITE
- Var.ADDRESS()
- Var.END()
- Var.RANGE()
- Var.SIZEOF()
- Var.STRing()
- Var.TYPEOF()
- Var.VALUE()

▲ 'Release Information' in 'Legacy Release History'

Format:	**Var.PROfile** [**%**<*format*>] <*variable*> [<*variable*>] [<*variable*>] [<*refresh_rate*>]
<*refresh_rate*>:	**0.1** | **1.0** | **10.0**

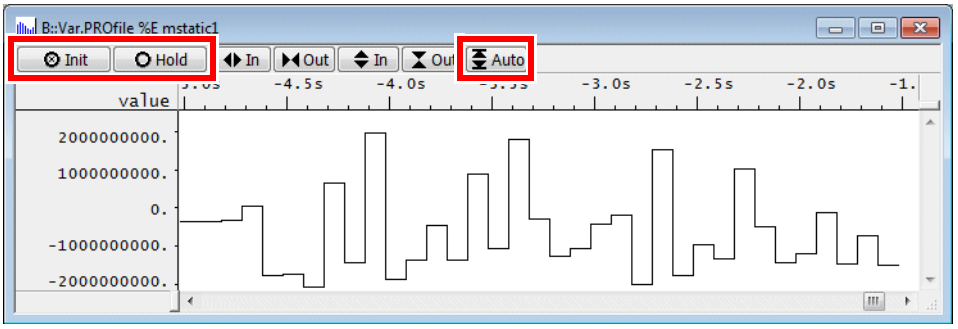
The value of the specified variable(s) is displayed graphically. The display requires **run-time memory access** if the variable value should be displayed while the program execution is running.

<*format*> Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

<*refresh_rate*> The refresh rate is measured in seconds. If no value is specified, then the display is updated and shifted every 100 ms.

Example 1:

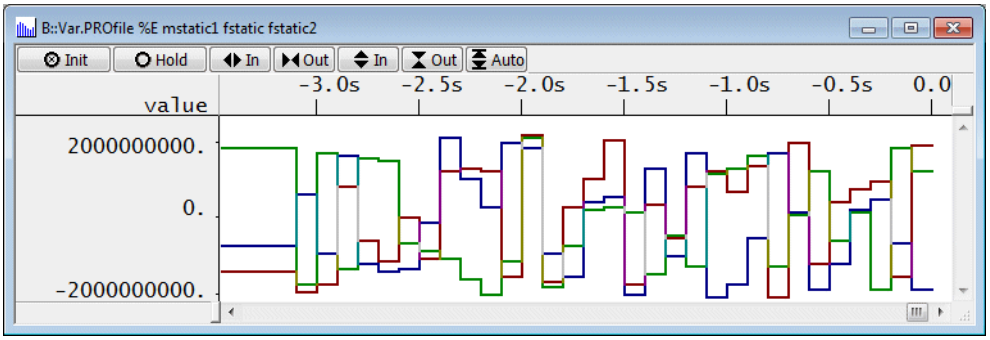
```
Var.PROfile %E mstatic1
```



Button	Description
Init	Restart display
Hold	Stop update/re-start update
AutoZoom	Best vertical scaling

Example 2: Up to three variables can be displayed. The following color assignment is used: first variable value red, second variable value green, third variable value blue.

```
Var.PROfile %E mstatic1 fstatic fstatic2
```



See also

■ Var ■ Var.DRAW ■ Var.set ■ Data.PROfile

▲ 'Release Information' in 'Legacy Release History'

Var.Ref — Referenced variables

Format:	**Var.Ref** [%*<format>*] [**/Track**]

Display of variables, similar to command **Var.Watch**. The variables referenced by the current source line are automatically added to the window.

Track	The window follows other windows. Otherwise the display is related to the next executed HLL line.
<format>	For a description of the *<format>* parameters, click here.

See also

■ Var ■ Var.Local ■ Var.set ■ Var.View

▲ 'Release Information' in 'Legacy Release History'

▲ 'Display Variables' in 'Training Source Level Debugging'

Format: **Var.set** [{**%***<format>*}] *<expression>*

It is possible to start this command by double-clicking with the mouse to the variable in the List Source window (**List**) or in a variable display window. Variable assignments done with the **Var.set** command result in a message in the TRACE32 message **AREA**. The command **Var.Assign** can be used for variable assignments without messaging to the TRACE32 message **AREA**.

<format> For a description of the *<format>* parameters, click here.

Example 1:

```
Var.set \mod1\venumvar=enum4      //assignment of value 'enum4' to the
                                  //variable 'venumvar' in module 'mod1'

Var.set charptr[4]='x'            //Content of the 5th element of array
                                  //'charptr' is set value 'x'.

                                  //complex C expression
Var.set xptr->next=(SYM_symbol*)sptr->prev->next[index]

Var.set i++                       //simple C expression

Var.set func7(1.5,2.5)            //execute a function in the target

                                  //interrupts are accepted while the
                                  //function is executed
```

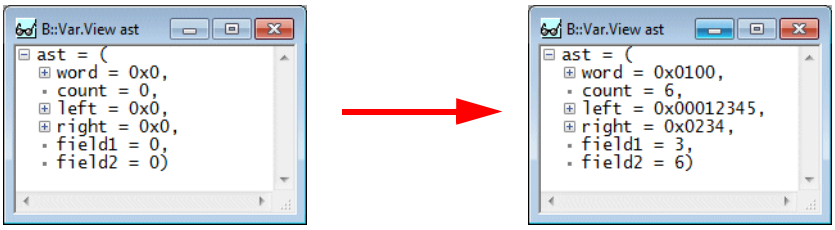
Example 2: Modification of arrays

```
Var.set flags[3..7]=12                //set flags[3..7] to 12

Var.set flags[3..7]=(1,2,3,4,5)       //set flags[3..7] to specified
                                      //values
```

```
//PRACTICE script for array comparison
Var.IF flags[3..7]==(1,2,3,4,5)
       PRINT "Array elements already initialized"
ELSE
       Var.set flags[3..7]=(1,2,3,4,5)
ENDDO
```

Example 3: Modifications of structures

```
Var.set ast=(0x100,6,0x12345,0x234,3,6)
```



Example 4: Assigning the result of TRACE32 functions to variables requires a special syntax.

```
// Assign result of TRACE32 function FILE.EXIST(<file>)to variable k

Var.set k=\FILE_EXIST("t32.men")

// multiply variable k with the result of the TRACE32 function
// Register(<name>) and assign the result to variable i

Var.set i=k*\Register(R10)

// assign the result of the TRACE32 function Data.Byte(<hex_address>) to
// the variable flags[3]

Var.set flags[3]=\Data_Byte(0x40004000)

// The following syntax is required if an Access Class is required
// here the Access Class NoCache

Var.set flags[3]=\Data_Byte((NC:0x40004000))
```

Example 5: If no assignment is made, the variable value will be displayed in the message line.

```
Var.set % i                         //displays value of variable 'i' in
                                    //decimal, hex and ASCII

Var.set %String struct1             //displays structure 'struct1'.
                                    //displays character array included
                                    //in the structure as strings
```

See also

- Var
- Var.Assign
- Var.DelWatch
- Var.Eval
- Var.Go
- Var.LOG
- Var.AddSticker
- Var.Break
- Var.DRAW
- Var.EXPORT
- Var.IF
- Var.NEW
- Var.AddWatch
- Var.Call
- Var.DRAWXY
- Var.FixedCHAIN
- Var.INFO
- Var.NEWGLOBAL
- Var.AddWatchPATtern
- Var.CHAIN
- Var.DUMP
- Var.FixedTABle
- Var.Local
- Var.NEWLOCAL

- Var.OBJECT
- Var.Ref
- Var.TYPE
- Var.WRITE
- Var.END()
- Var.RANGE()
- Var.VALUE()
- Var.PATtern
- Var.Step
- Var.View
- Var.ADDRESS()
- Var.EXIST()
- Var.SIZEOF()
- Var.PRINT
- Var.TABle
- Var.Watch
- Var.BITPOS()
- Var.FVALUE()
- Var.STRing()
- Var.PROfile
- Var.TREE
- Var.WHILE
- Var.BITSIZE()
- Var.ISBIT()
- Var.TYPEOF()

▲ 'Release Information' in 'Legacy Release History'
▲ 'Testing of Functions' in 'Training Source Level Debugging'

See also

- Var.Step.BackChange
- Var.Step.BackTill
- Var.Step.Change
- Var.Step.Till
- Var
- Var.set
- Step.single

Var.Step.BackChange Step back till expression changes

Format: **Var.Step.BackChange** [*<expression>*]

Steps back till the expression changes. The command will stop also if the expression cannot be evaluated.

Example:

```
Var.Step.BackChange k          // steps till variable k changes

Var.Step.BackChange ptr->x     // steps till the contents of the
                               // structure pointed to by 'ptr'
                               // changes

Var.Step.BackChange flags      // steps till one element of the array
                               // 'flags' changes
```

See also

- Var.Step

▲ 'Release Information' in 'Legacy Release History'

Var.Step.BackTill Step back till expression true

Format: **Var.Step.BackTill** [*<expression>*]

Steps back till the boolean expression becomes true (i.e. not zero). The command will stop also, if the expression cannot be evaluated.

```
Var.Step.BackTill i>0x10     //steps till variable 'i' is larger than 10
```

See also

- Var.Step

▲ 'Release Information' in 'Legacy Release History'

Format: **Var.Step.Change** [*<expression>*]

Steps till the expression changes. The command will stop also if the expression cannot be evaluated.

Examples:

```
Var.Step.Change k           // steps till variable k changes

Var.Step.Change ptr->x      // steps till the contents of the structure
                            // pointed to by 'ptr' changes

Var.Step.Change flags       // steps till one element of the array
                            // 'flags' changes
```

See also

- Var.Step

▲ 'Release Information' in 'Legacy Release History'

Var.Step.Till — Step till expression true

Format: **Var.Step.Till** [*<expression>*]

Steps till the boolean expression becomes true (i.e. not zero). The command will stop also if the expression cannot be evaluated.

Example:

```
Var.Step.Till i>0x10        //steps till variable 'i' is larger than 10
```

See also

- Var.Step

▲ 'Release Information' in 'Legacy Release History'

Format:	**Var.TABle** [%<*format*>] [<*variable*> [<*pointer*> …]]

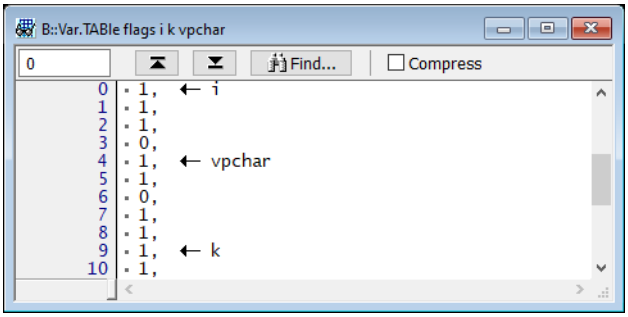
Displays the first expression as an array. The extra arguments are displayed as pointers or indexes to that array.

<*format*>	Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

Examples:

```
Var.TABle %Location flags i k vpchar
Var.TABle vpchar[0..100]                    //'artificial' array build on
                                            // a pointer
```

Displays the first expression as an array.



See also

■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

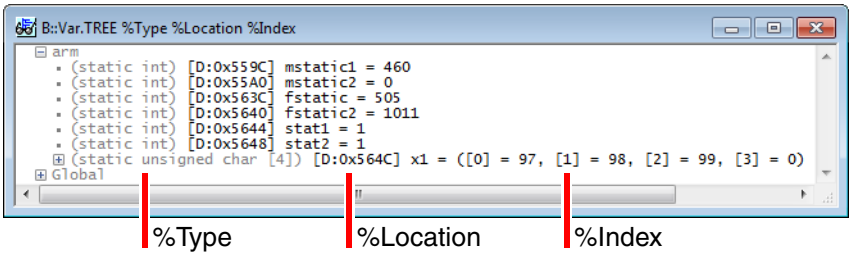
Format: **Var.TREE** [**%***<format>*]

Displays the HLL variables in the form of a tree structure. The tree structure is broken down by program and module.

<format> Use the *<format>* parameters to display the variables in the desired format. For a description of the *<format>* parameters, click here.

Example:

```
Var.TREE %Type %Location %Index
```



See also

■ Var ■ Var.set

▲ 'Release Information' in 'Legacy Release History'

Format: **Var.TYPE** [**%**<*format*>] [<*expression*>] …

<*format*>:
all
DEFault
Type [**.on** | **.OFF**]
Open [**.on** | **.OFF** | **.2** | **.3** | **.4**]
Location [**.on** | **.OFF**]
HIdden [**.on** | **.OFF**]
Recursive [**.on** | **.OFF** | **.2** | **.3** | **.4**]
…

The specified formula is interpreted and the types of the according values are displayed. By the options of this command the way of display may be modified in wide range.

<*format*> Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

See also

■ Var ■ Var.set

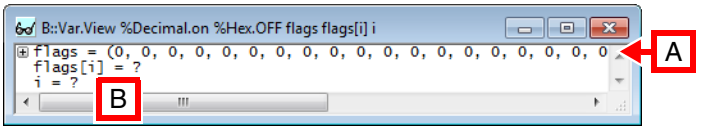
▲ 'Release Information' in 'Legacy Release History'

Format:	**Var.View** [%<*format*>] [<*variable*>] …

The specified formula is interpreted and the according values are displayed. If a syntactical error is made, just a warning will be received. That's why the definition of a window showing local variables is allowed, without the program counter being valid to the according procedure.

<*format*> Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

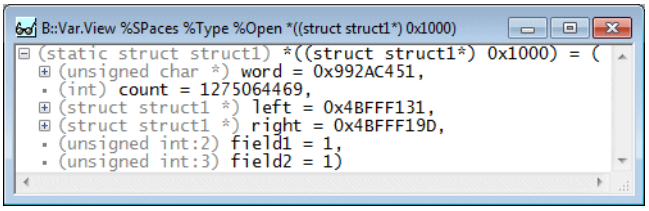
- Display variables:



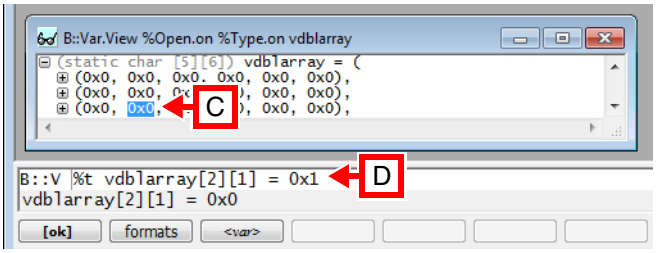
A Static variable.

B ? indicates that the local variable is not valid in the current program context.

- Interpret and display memory content in HLL format:



- Modify a variable value or structure element:



C Double-click the value you want to modify.

D Double-clicking inserts the current value into the TRACE32 command line. Simply enter a new value, e.g. `0x1`

NOTE:	A variable value is displayed in red color, then the assigned value is not within the defined range of the variable type. The typical reason in C programs is that an enumeration variable has a value that does not correspond to one of the defined enumerators.

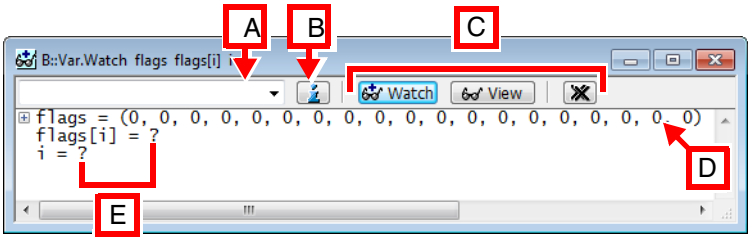
See also

- Var
- Var.AddWatch
- Var.Local
- Var.Ref
- Var.set
- Var.Watch
- Frame.view

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

Format:	**Var.Watch** [%<*format*>] [<*variable*>] …

Opens a **Var.Watch** window, displaying the specified variables. Further variables can be added on the fly to the window by using the window buttons or the **Var.AddWatch** command.



A Recall already used variables

B Browse symbols

C Var.AddWatch, Var.View, Var.DelWatch

D Double-click to modify or use Var.set

E ? indicates that the variable is not valid in the current program context.

<*format*>	Use the <*format*> parameters to display the variables in the desired format. For a description of the <*format*> parameters, click here.

Example 1: If the command **Var.Watch** is used with format parameters only, these format parameters are applied to all variables added to the window.

```
Var.Watch %Decimal %Hex

Var.AddWatch mstatic1                  ; mstatic1 is displayed in decimal
                                       ; and hex format
```

Example 2: If the command **Var.Watch** is used with variable names and format parameters, the format parameters apply only to the specified variables. Variables added to the window are formatted in the default way.

```
Var.Watch %Index flags %Hex mstatic1 %Hex.OFF %SpotLight ast

Var.AddWatch mstatic2                  ; mstatic1 is displayed in the
                                       ; default format
```

Example 3: The **Var.Watch** window can also evaluate HLL expressions.

```
Var.Watch  mstatic1 ast.count enumvar mstatic1+ast.count/enumvar
```

See also

■ Var ■ Var.AddWatch ■ Var.set ■ Var.View

▲ 'Release Information' in 'Legacy Release History'
▲ 'Display Variables' in 'Training Source Level Debugging'

Var.WHILE PRACTICE loop construction

Format: **Var.WHILE** *<hll_condition>*

Repeats the next command or command block while *<hll_condition>* is true. **Var.WHILE** is the counterpart to the PRACTICE **WHILE** instruction.

<hll_condition> Allows to specify the condition in the syntax of the programming language used (C, C++, …).

Example:

```
Var.WHILE ast.count<1238
(
  Var.set ast.count++
  …
)
```

See also

■ Var ■ Var.set ■ WHILE

▲ 'Release Information' in 'Legacy Release History'

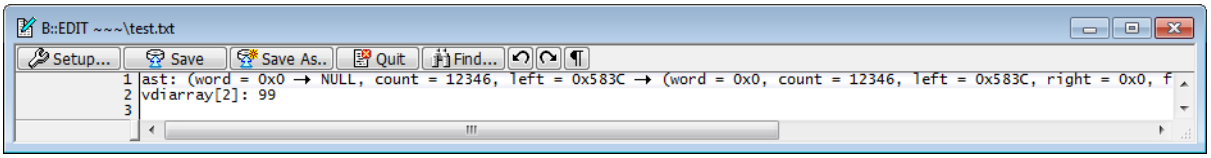
Format:	**Var.WRITE #***<file_number>* [**%CONTinue**] [**%***<format>*] [*<variable>*] …

Writes the values of the specified variables to file. The command is an HLL version of the **WRITE** command.

CONTinue	Adds the string to the current output line in the selected file without inserting a newline character.
<format>	Using the *<format>* parameters, you can modify the output in various ways. For a description of the *<format>* parameters, click here.

Example:

```
;create and open a file for writing
OPEN #1 ~~~\test.txt /Create

;write the variable name 'ast' to the file
WRITE #1 "ast: "

;continue with the values of 'ast' in the same line
Var.WRITE #1 %CONTinue %Recursive.on ast

;write the array name and a selected index 'vdiarray[2]' to the file
WRITE #1 "vdiarray[2]: "

;continue with the value of the array index 2 in the same line
Var.WRITE #1 %CONTinue vdiarray[2]

;close the file for writing
CLOSE #1

;open the file for editing in TRACE32
EDIT ~~~\test.txt
```



See also

- Var
- Var.EXPORT
- Var.PRINT
- Var.set
- CLOSE
- OPEN

▲ 'Release Information' in 'Legacy Release History'

VCO — Clock generator

Simulator only

See also

- VCO.BusFrequency
- VCO.Down
- VCO.Frequency
- VCO.Rate
- VCO.RESet
- VCO.state
- VCO.TimeBaseFrequency
- VCO.Up

VCO.BusFrequency — Control bus clock

Simulator only

Format: **VCO.BusFrequency** *<frequency>*

Sets the bus clock frequency.

See also

- VCO
- VCO.state

VCO.Down — Frequency down

Simulator only

Format: **VCO.Down** [*<frequency>*]

<frequency>: **50000.** …

Step down with the VCO frequency.

```
VCO.Down                                ; frequency down 50 kHz
```

See also

- VCO
- VCO.Frequency
- VCO.state
- VCO.Up

VCO.Frequency

Control VCO clock

Simulator only

Format:	**VCO.Frequency** *<frequency>*
<frequency>:	**1000Hz** ... **2GHz**

In the TRACE32 Instruction Set Simulator, the value sets the number of cycles (instruction fetch and load/store) after which the simulation time is increased by one second.

<frequency>	All frequency definitions may be done in Hz, kHz, MHz or GHz.

```
VCO.Frequency 20MHz                     ; set VCO clock to 20 MHz

VCO.Frequency 10.5MHz                   ; set VCO clock to 10.5 MHz

VCO.Frequency 1800KHz                   ; set VCO clock to 1.8 MHz
```

See also

- VCO
- VCO.Down
- VCO.state
- VCO.Up
- VCO()

VCO.Rate

VCO rate

Simulator only

Format:	**VCO.Rate** *<rate>*

Defines the rate between VCO clock and internal CPU clock.

See also

- VCO
- VCO.state

VCO.RESet — VCO reset

Simulator only

Format:	**VCO.RESet**

The VCO is initialized to the default frequency.

See also

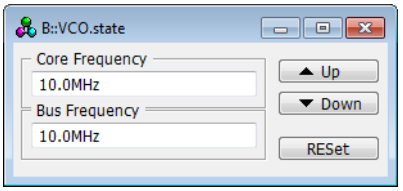
- VCO
- VCO.state

VCO.state — State display

Simulator only

Format:	**VCO.state**

Displays the state of the VCO.



See also

- VCO
- VCO.BusFrequency
- VCO.Down
- VCO.Frequency
- VCO.Rate
- VCO.RESet
- VCO.TimeBaseFrequency
- VCO.Up
- VCO()

VCO.TimeBaseFrequency — Set the time base clock

Simulator only

Format:	**VCO.TimeBaseFrequency** *<frequency>*
<frequency>:	**1000Hz** ... **2GHz**

Sets the time base clock.

See also

- VCO
- VCO.state

Simulator only

Format:	**VCO.Up** [*<frequency>*]
<frequency>:	**1000Hz** ... **2GHz**

Step up with VCO clock.

Example:

```
VCO.Up                                  ; frequency up by 50 kHz

VCO.Up 1MHz                             ; frequency up by 1 MHz
```

See also

- VCO
- VCO.Down
- VCO.Frequency
- VCO.state

VCU — VCU registers (Vector Computational Unit)

Ceva-X only

VCU commands refer to the Vector Computational Unit which is an optional unit available only to the new Ceva-XC devices. In addition these commands must be made available by specifying the actual number of implemented VCUs (see **SYStem.VCU.INSTances**).

See also

- VCU.Init
- VCU.RESet
- VCU.Set
- VCU.view

VCU.Init — Initialize VCU registers

Ceva-X only

Format: **VCU.Init**

Sets the VCU registers to their default values.

See also

- VCU
- VCU.view

VCU.RESet — Reset VCU registers

Ceva-X only

Format: **VCU.RESet** [**/VCU***<instance>*]

Default: VCU0

Resets all registers of the selected instance to zero.

See also

- VCU
- VCU.view

VCU.Set — Set VCU register

Ceva-X only

Format: **VCU.Set** *<register>* *<value>* [*/<option>*] [**/VCU***<instance>*]

Default: VCU0

Modifies the selected *<register>* of the according VCU instance. MLD registers become available if **SYStem.VCU.MLD** is **ON**.

See also

- VCU
- VCU.view

VCU.view — Display VCU registers

Ceva-X only

Format: **VCU.view** [*/<option>*] [**/VCU***<instance>*]

Default: VCU0

Control panel to display and modify VCU registers of the corresponding VCU instance. MLD registers become available if **SYStem.VCU.MLD** is **ON**.

See also

- VCU
- VCU.Init
- VCU.RESet
- VCU.Set

VE — Virtual execution mode

VE is the virtual execution mode of TRACE32. The virtual execution mode can be used to run and debug a target application even if no target memory is available. This can be useful to run initialization code for the target.

After turning on the VE, all program code will be simulated by the debugger's instruction set simulator. The simulator will cause instruction fetches/loads and stores according the program. The target of the fetch/load/store depends on the TRACE32 virtual memory (VM:). If an address is fetched/loaded/stored which has been set using **Data.LOAD** or **Data.Set**, the simulator will access simulator memory. All other addresses will be forwarded to the processor.

See also

- VE.OFF
- VE.ON

▲ 'Release Information' in 'Legacy Release History'

VE.OFF — Turn off virtual execution mode

Format: **VE.OFF**
VM.OFF (deprecated)

Turns off the virtual execution mode.

See also

- VE

VE.ON — Turn on virtual execution mode

Format: **VE.ON**
VM.ON (deprecated)

Turns on the virtual execution mode.

See also

- VE

VPU — Vector Processing Unit (VPU)

Not all core architectures supported

The **VPU** command group is used to display and modify the VPU (Vector Processing Unit) registers. These commands do not support all core architectures.

See also

■ VPU.Init ■ VPU.Set ■ VPU.view ❑ VPU()

▲ 'VPU Functions' in 'General Function Reference'

VPU.Init — Initialize VPU registers

Not all core architectures supported

Format: **VPU.Init** [/*<option>*]
VPU.RESet (deprecated)

Resets all Vector Processing Unit (VPU) registers.

<option> For a description of the options, see **Register.view**.

See also

■ VPU ■ VPU.view

Not all core architectures supported

Format:	**VPU.Set** *<register>* *<value>* [*/<option>*]
<register>: (PowerPC 74xx/86xx only)	**VR0**..**VR31** **VRSAVE** **VSCR**
<register>: (APEX only)	**V0...V7** **VC0...VC3** **VCS0...VCS7** **OVV**
<register>: (TPC only)	**VPE0_V0**…**VPE0_V44** **VPE1_V0**…**VPE1_V44** … **VPE63_V0**…**VPE63_V44** **VPE0_VP0**…**VPE0_VP15** **VPE1_VP0**…**VPE1_VP15** … **VPE63_VP0**…**VPE63_VP15**

Sets the value of the specified Vector Processing Unit register.

<register>: (PowerPC 74xx/86xx only)	To modify the ALTIVEC vector registers VR0-VR31, split the value in four 32-bit values. If less than four values are given, the values will be aligned to LSB and undeclared values will be set to zero.
<register>: (TPC only)	The register name describes the Vector Processing Element number and the register class and number. The first part of the register name specifies the Vector Processing Element (VPE) number. The second part specifies the register class, Vector (V) or Vector Predicate (VP), and the respective register number.
<option>:	For a description of the options, see **Register.view**.

Example: Brief example for PowerPC

```
VPU.Set VR2 0x11111111 0x22222222 0x33333333 0x44444444
VPU.Set VRSAVE 0x0000003F
VPU.Set VSCR 0x00010000

PRINT VPU(VR2.W3)
PRINT VPUCR(VRSAVE)
```

See also

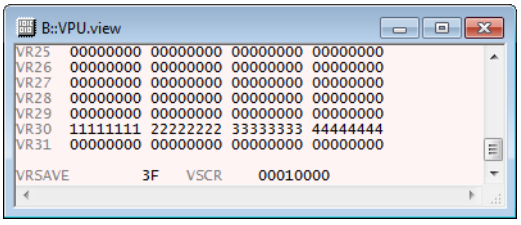
■ VPU ■ VPU.view

VPU.view — Display ALTIVEC register window

Not all core architectures supported

Format: **VPU.view** [*/<option>*]

Opens a window displaying the register contents of the Vector Processing Unit.



<option> For a description of the options, see **Register.view**.

See also

■ VPU ■ VPU.Init ■ VPU.Set

▲ 'Release Information' in 'Legacy Release History'