## General Commands Reference Guide V

### TRACE32 Online Help

### TRACE32 Directory

### TRACE32 Index

### TRACE32 Documents  ............................................................................................................

### General Commands  ............................................................................................................

### General Commands Reference Guide V  ................................................................................

### History  .............................................................................................................

### Var  .............................................................................................................................................

### Overview Var

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>HLL variables and expressions</td>
</tr>
<tr>
<td>Var.AddWatch</td>
<td>Add variable to Var.Watch window</td>
</tr>
<tr>
<td>Var.AddWatchPATtern</td>
<td>Add variables to Var.Watch window using wildcards</td>
</tr>
<tr>
<td>Var.Assign</td>
<td>Assignment to a variable</td>
</tr>
<tr>
<td>Var.Break</td>
<td>Breakpoint on variable</td>
</tr>
<tr>
<td>Var.Break.Delete</td>
<td>Delete breakpoint on variable</td>
</tr>
<tr>
<td>Var.Break.direct</td>
<td>Set temporary breakpoint on HLL expression</td>
</tr>
<tr>
<td>Var.Break.Pass</td>
<td>Define pass condition for breakpoint</td>
</tr>
<tr>
<td>Var.Break.Set</td>
<td>Set breakpoint to HLL expression</td>
</tr>
<tr>
<td>Var.Call</td>
<td>Call a new procedure</td>
</tr>
<tr>
<td>Var.CHAIN</td>
<td>Display linked list</td>
</tr>
<tr>
<td>Var.DelWatch</td>
<td>Delete variable from watch</td>
</tr>
<tr>
<td>Var.DRAW</td>
<td>Graphical variable display</td>
</tr>
<tr>
<td>Var.DUMP</td>
<td>Memory dump</td>
</tr>
<tr>
<td>Var.EXPORT</td>
<td>Export variables in CSV format to file</td>
</tr>
<tr>
<td>Var.FixedCHAIN</td>
<td>Display linked list</td>
</tr>
<tr>
<td>Var.FixedTABle</td>
<td>Display table</td>
</tr>
<tr>
<td>Var.Go</td>
<td>Real-time emulation</td>
</tr>
<tr>
<td>Var.Go.Back</td>
<td>Re-run program backwards until variable access (CTS)</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Var.Go.Change</td>
<td>Real-time emulation till expression changes</td>
</tr>
<tr>
<td>Var.Go.direct</td>
<td>Real-time emulation with breakpoint</td>
</tr>
<tr>
<td>Var.Go.Till</td>
<td>Real-time emulation till expression true</td>
</tr>
<tr>
<td>Var.IF</td>
<td>PRACTICE conditional branching</td>
</tr>
<tr>
<td>Var.INFO</td>
<td>View information about HLL variable or HLL expression</td>
</tr>
<tr>
<td>Var.Local</td>
<td>Local variables</td>
</tr>
<tr>
<td>Var.LOG</td>
<td>Log variables</td>
</tr>
<tr>
<td>Var.NEW</td>
<td>Creates a TRACE32-internal variable</td>
</tr>
<tr>
<td>Var.NEWGLOBAL</td>
<td>Creates a global TRACE32-internal variable</td>
</tr>
<tr>
<td>Var.NEWLOCAL</td>
<td>Creates a local TRACE32-internal variable</td>
</tr>
<tr>
<td>Var.PATtern</td>
<td>Display variables allowing wildcards for symbol name and type</td>
</tr>
<tr>
<td>Var.PRINT</td>
<td>Display variables</td>
</tr>
<tr>
<td>Var.PROfile</td>
<td>Graphical display of variable</td>
</tr>
<tr>
<td>Var.Ref</td>
<td>Referenced variables</td>
</tr>
<tr>
<td>Var.set</td>
<td>Modify variable</td>
</tr>
<tr>
<td>Var.Step</td>
<td>Step</td>
</tr>
<tr>
<td>Var.Step.BackChange</td>
<td>Step back till expression changes</td>
</tr>
<tr>
<td>Var.Step.BackTill</td>
<td>Step back till expression true</td>
</tr>
<tr>
<td>Var.Step.Change</td>
<td>Step till expression changes</td>
</tr>
<tr>
<td>Var.Step.Till</td>
<td>Step till expression true</td>
</tr>
<tr>
<td>Var.TABLE</td>
<td>Display table</td>
</tr>
<tr>
<td>Var.TREE</td>
<td>Display variables in the form of a tree structure</td>
</tr>
<tr>
<td>Var.TYPE</td>
<td>Display variable types</td>
</tr>
<tr>
<td>Var.View</td>
<td>Display variables</td>
</tr>
<tr>
<td>Var.Watch</td>
<td>Open Var.Watch window</td>
</tr>
<tr>
<td>Var.WHILE</td>
<td>PRACTICE loop construction</td>
</tr>
<tr>
<td>Var.WRITE</td>
<td>Write variables to file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VCO</th>
<th>Clock generator</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview VCO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VCO.BusFrequency</td>
<td>Control bus clock</td>
<td>70</td>
</tr>
<tr>
<td>VCO.Clock</td>
<td>Control emulation clock</td>
<td>71</td>
</tr>
<tr>
<td>VCO.Down</td>
<td>Frequency down</td>
<td>71</td>
</tr>
<tr>
<td>VCO.Frequency</td>
<td>Control VCO clock</td>
<td>72</td>
</tr>
<tr>
<td>VCO.Rate</td>
<td>VCO rate</td>
<td>72</td>
</tr>
<tr>
<td>VCO.RESet</td>
<td>VCO reset</td>
<td>73</td>
</tr>
<tr>
<td>VCO.state</td>
<td>State display</td>
<td>73</td>
</tr>
<tr>
<td>VCO.TimeBaseFrequency</td>
<td>Set the time base clock</td>
<td>74</td>
</tr>
<tr>
<td>VCO.Up</td>
<td>Frequency up</td>
<td>74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VCU</th>
<th>VCU registers (Vector Computational Unit)</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCU.Init</td>
<td>Initialize VCU registers</td>
<td>76</td>
</tr>
<tr>
<td>VCU.RESet</td>
<td>Reset VCU registers</td>
<td>76</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>VCU.Set</td>
<td>Set VCU register</td>
<td>77</td>
</tr>
<tr>
<td>VCU.view</td>
<td>Display VCU registers</td>
<td>77</td>
</tr>
<tr>
<td>VE</td>
<td>Virtual execution mode</td>
<td>78</td>
</tr>
<tr>
<td>VE.OFF</td>
<td>Turn off virtual execution mode</td>
<td>78</td>
</tr>
<tr>
<td>VE.ON</td>
<td>Turn on virtual execution mode</td>
<td>78</td>
</tr>
<tr>
<td>VPU</td>
<td>Vector Processing Unit (VPU)</td>
<td>79</td>
</tr>
<tr>
<td>VPU.Init</td>
<td>Initialize ALTIVEC registers</td>
<td>79</td>
</tr>
<tr>
<td>VPU.Set</td>
<td>Modify ALTIVEC registers</td>
<td>79</td>
</tr>
<tr>
<td>VPU.view</td>
<td>Display ALTIVEC register window</td>
<td>80</td>
</tr>
</tbody>
</table>

Usage:

(B) command only available for ICD
(E) command only available for ICE
(F) command only available for FIRE
History

16-Jul-18    Renamed the placeholder `<magic_number>` to `<task_magic>` to be able to distinguish between `<task_magic>` as well as `<machine_magic>` and `<space_magic>`.

13-Jun-18    Description for commands `Var.AddWatchPATtern` and `Var.PATtern` added.
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.

<table>
<thead>
<tr>
<th>Symbol Table</th>
<th>Entry</th>
<th>HLL-Windows</th>
<th>Assembler windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>_vfloat</td>
<td>_vfloat or</td>
<td>vfloat</td>
<td>_vfloat</td>
</tr>
<tr>
<td></td>
<td>vfloat</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

▲ 'Var Functions’ in ‘General Function Reference’
▲ ‘Release Information’ in ‘Release History’
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.

```
E::d.l `class1::method1`  //access to same method (ANSI mangled)
E::d.l method1

E::d.l `class1::class1`  //creator of class class1
E::d.l `class1::~class1`  //destructor of class class1
E::d.l `class1::operator++`  //overloaded operator
E::d.l `class1::operator+(int)`  //overloaded operator with prototype
E::v `class1::operator+(int)`  //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:

```
v *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.

```
v __HEAP  //displays nothing (if __HEAP is a label)
v *__HEAP  //assumes __HEAP as a pointer to character
v (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'.

```
v (0x2000)(1,2,3)  //calls the function at 2000 (hex)
v __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.

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

```
v flags[0..19]=0        //clears the array to 0
v 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.

```
v vpchar = 0        //will call the 'malloc' function
v vpchar = "abc"    //copy the string "abc" to location 0x100
```

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

```
v.g.t pname=="TEST"   //execute program till string equal
```

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

```
v 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.

```
v.type %m 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.

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

```c
w.v.v signal.content[signal.type]
```

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

```c
v.ast=(1,2,3) //assigns the first three members values
v.if point==(0,0) //condition is true when first elements are zero
v.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.

```c
v *this //displays the “regular” object
v *[this] //displays the most derived class of the object
```

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

```c
v.v flags[0] := 12H //standard MODULA hexadecimal syntax
v.v flags[0] := 0x12 //also accepted (like ‘C’)

v.v CARDINAL(1.24) //typecast like ‘C’: (CARDINAL) 1.23
v.v ^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:

```plaintext
[<format>] ... all
Ascii [.on | .OFF]
BINary [.on | .OFF]
Compact [.on | .OFF]
Decimal [.on | .OFF]
DEFault
DUMP [.on | .OFF]
E [.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 affects 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.

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.

![Fixed width fields example](image)

**Hex**

Display of values in hex format.

![Hex display example](image)

**Hidden**

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

```cpp
E::w.v.v %c %m X12 %hi X12
X12 = (j = 0)
X12 = (j = 0,
     i__2X1 = 0,
     OX2 = (i = 0))
```

**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.

![Index display example](image)
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.

```
E::w.v.v %m.3 %r.2 ast
ast = (word = 0x123456 ➔ INVALID,
   count = 12346,
   left = 0x2540 ➔ (word = 0x0 ➔ NULL,
      count = 12346,
      left = 0x2540 ➔ (word = 0x0,
         count = 12346,
         left = 0x2540,
         right = 0x1,
         field1 = 1,
         field2 = 2),
      right = 0x1 ➔ (word = 0x0,
         count = 0,
         left = 0x0,
         right = 0x0,
         field1 = 0,
         field2 = 0),
   field1 = 1,
   field2 = 2),
right = 0x1 ➔ (word = 0x9123458 ➔ INVALID,
   count = 0,
   right = 0x1 ➔ (word = 0x9123458 ➔ INVALID,
      count = 0,
      pointer outside valid range
```

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
```
**SPaces**

Selectors 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 and Fixed formats. Highlighted are only elements for the first objects of a line.
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.
Values of pointers are displayed symbolic.

```
E::w.v.v %c %m %y vpchar ast
vpchar = 0x2774 = \MCC\flags
ast = (  
    word = 0x0 = \MCC\stack,
    count = 12346,
    left = 0x2540 = \MCC\ast,
    right = 0x1 = \MCC\__simulated_output,
    field1 = 1,
```

**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.
Type

Display of the variable type.

WideString

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

Var.ADDRESS (<expression>)
Returns the first address, occupied by the expression. (e.g. function or variable).

Var.RANGE (<expression>)
Returns the address range occupied by the expression.

Var.END (<expression>)
Returns the last address occupied by the expression.

Var.SIZEOF (<expression>)
Returns the size occupied by the expression in memory.

Var.FVALUE (<expression>)
Returns the contents of the HLL expression as floating point value.

Var.VALUE (<expression>)
Returns the contents of the HLL expression.a

PRINT Var.VALUE("ast->left*i-5");

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

Var.AddWatch
Add variable to Var.Watch window

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
▲ 'HLL Structures’ in ’ICE User’s Guide’
▲ ‘Release Information’ in ’Release History’
▲ ‘Display Variables’ in ’Training HLL Debugging’
Var.AddWatchPatterns  
Add variables to Var.Watch window using wildcards

| Format: | Var.AddWatchPatterns [%<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.AddWatchPatterns extend*

See also
- Var
- Var.set

Var.Assign  
Assignment to a variable

| 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 'Release History'

©1989-2019 Lauterbach GmbH
Var.Break.Delete

Delete breakpoint on variable

Format:  
Var.Break.Delete \(<\text{hll}\text{-expression}>\) [\(<\text{breaktype}>\)]

\(<\text{breaktype}>\):
- Program | ReadWrite | Read | Write
- ProgramPass | ProgramFail
- Alpha | Beta | Charly | Delta | Echo
- WATCH | BusTrigger | BusCount
- TraceEnable | TraceData | TraceON | TraceOFF | TraceTrigger

\(<\text{task_magic}>\) | \(<\text{task_id}>\) | \(<\text{task_name}>\)
- Program | Hll | Spot | Read | Write (E)
- Alpha | Beta | Charly (E)
- Data (E)
- DFEault (E)
- ALL
- AllNotHll (E)

Removes the breakpoints set to the address range specified by \(<\text{hll}\text{-expression}>\).

\(<\text{breaktype}>\)
For a description of the breakpoint types and breakpoint options, see Break.Set.

\(<\text{hll}\text{-expression}>\)
Allows to specify the HLL expression in the syntax of the programming language used (C, C++, …).

\(<\text{task_magic}>\), 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

- 'Breakpoint Memory' in 'ICE User's Guide'
- 'Release Information' in 'Release History'
- 'Breakpoint Handling' in 'Debugger Basics - Training'
- 'Breakpoint Handling' in 'Debugger Basics - SMP Training'
- 'Breakpoint Handling' in 'Training FIRE Basics'
- 'Breakpoints' in 'Training ICE Basics'
Var.Break.direct  
Set temporary breakpoint on HLL expression

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
DIISable | DIISableHIT | 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 'Release History'
- 'Breakpoints' in 'Training ICE Basics'
Var.Break.Pass

Define pass condition for breakpoint


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
- 'Real-time Emulation' in 'ICE User's Guide'
- 'Release Information' in 'Release History'
Var.Break.Set  

Set breakpoint to HLL expression

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 | 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.

For a description of the breakpoint types and breakpoint options, see Break.Set.

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

See also “What to know about the Task Parameters” (general_ref_t.pdf).
**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.

**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+
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 %m %l ast ast.left vpchar

The softkey « [rebase]» takes the first entry displayed in the window as the new base of the linked list. This function can be used, if the window update gets slow, due to the long time required to walk along the linked list. The softkey « [first]» resets the linked list to the first element defined in the command line.

<table>
<thead>
<tr>
<th>E::w.v.chain %m %l ast ast.left vpchar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0 (0) [D:2A16] ([D:2A16] word = 0x0,</td>
</tr>
<tr>
<td>[D:2A1A] count = 12346,</td>
</tr>
<tr>
<td>[D:2A1E] left = 0x2A16,</td>
</tr>
<tr>
<td>[D:2A22] right = 0x0,</td>
</tr>
<tr>
<td>[D:2A26.0] field1 = 1,</td>
</tr>
<tr>
<td>[D:2A26.2] field2 = 2), ← vpchar</td>
</tr>
<tr>
<td>0x1 (1) [SD:2A16] ([SD:2A16] word = 0x0,</td>
</tr>
<tr>
<td>[SD:2A1A] count = 12346,</td>
</tr>
<tr>
<td>[SD:2A1E] left = 0x2A16,</td>
</tr>
<tr>
<td>[SD:2A22] right = 0x0,</td>
</tr>
<tr>
<td>[SD:2A26.0] field1 = 1,</td>
</tr>
<tr>
<td>[SD:2A26.2] field2 = 2), ← vpchar</td>
</tr>
<tr>
<td>0x2 (2) [SD:2A16] ([SD:2A16] word = 0x0,</td>
</tr>
<tr>
<td>[SD:2A1A] count = 12346,</td>
</tr>
<tr>
<td>[SD:2A1E] left = 0x2A16,</td>
</tr>
<tr>
<td>[SD:2A22] right = 0x0,</td>
</tr>
</tbody>
</table>

See also

- Var
- Var.set

- 'HLL Structures' in 'ICE User's Guide'
- 'Release Information' in 'Release History'
- 'Display Variables' in 'Training HLL Debugging'
Var.DelWatch
Delete variable from watch

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

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

See also

- Var
- Var.set
- ‘Release Information’ in ‘Release History’

Var.DRAW
Graphical variable display

Format: **Var.DRAW [%<format>] <hll_expression> [scale] [offset] [l<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 10.

- **<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:**

```plaintext
Var.DRAW cstrl
Var.DRAW sinewave
```

![Graph of cstrl](image1)

![Graph of sinewave](image2)
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.set
- <trace>.DRAW
- Data.DRAW
- Data.DRAWFFT
- Data.DRAWXY
- Data.IMAGE

▲ 'HLL Structures' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
▲ 'Display Variables' in 'Training HLL Debugging'
### Var.DUMP

#### Memory dump

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

**Format:**    
```
Var.DUMP [%<format>] [[&]<variable>] … [I<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 | Hill | Spot | Read | Write | Alpha | Beta | Charly

For a description of the `<format>` parameters, see “Display Formats”, page 10.

For a description of the options, see `Data.dump`.

See also
- Var
- Var.set
- SETUPDUMP

▲ 'Release Information' in 'Release History'

©1989-2019 Lauterbach GmbH
Export variables in CSV format (Comma-Separated Values) for import to other applications. Existing file contents are overwritten if the file already exists.

**Format:**

```
Var.EXPORT <file> [%<format>] [<variable>] … [/Append]
```

- **<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:**

```plaintext
; 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`.

`tbd.`

**See also**

- `Var`
- `Var.set`
- `Var.WRITE`
- `PRinTer.EXPORT`
Var.FixedCHAIN

Display linked list

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 %1 %m ast ast.left vpchar
```

See also
- Var
- Var.set
- ‘Release Information’ in ‘Release History’
- ‘Display Variables’ in ‘Training HLL Debugging’

Var.FixedTABle

Display table

The first expression must be 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.

```
Var.FixedTABle [%<format>] <array> {<index>} {<pointer>}
```

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 'Release History'
- 'Display Variables' in 'Training HLL Debugging'
Var.Go

Re-run program backwards until variable access (CTS)

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
- 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.
For a description of the breakpoint types and breakpoint options, see Break.Set.

See also

- Var.Go
- Var.Go.direct

▲ 'Release Information' in '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:**

CTS.GOTO -1209874.

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

Var.Go.Change <expression>

v.b.s flags /w
v.g.c flags  //starts the emulation and restarts, if the array
//flags has not changed

**See also**

- Var.Go
- Var.Go.direct

▲ 'Real-time Emulation' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
Var.Go.direct

Real-time emulation with breakpoint

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

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

<brktype>: 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 | 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

For a description of the breakpoint types and breakpoint options, see Break.Set.

See also “What to know about the Task Parameters” (general_ref_t.pdf).
Examples:

```
v.g flags  //run till any element of 'flags' is accessed
v.g vfloat /w  //run till a write to 'vfloat' occurs
```

See also

- Var.Go
- Var.Go.Back
- Var.Go.Change
- Var.Go.Till

▲ ’Breakpoint Memory’ in ’ICE User's Guide’
▲ ’Real-time Emulation’ in ’ICE User's Guide’

---

### 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:

```
v.b.s vfloat /w  //starts the emulation and restarts, if the value
v.g.t vfloat<=1.57  //of vfloat is larger than 1.57
```

See also

- Var.Go
- Var.Go.direct

▲ ’Real-time Emulation’ in ’ICE User's Guide’
▲ ’Release Information’ in ’Release History’
Var.IF

PRACTICE conditional branching

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:

```c
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 'Release History'
Var.INFO

View information about HLL variable or HLL expression

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 'Release History'
- ▲ 'The Symbol Database' in 'Training HLL Debugging'
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.

<table>
<thead>
<tr>
<th>E68::w.v.l</th>
<th>%m</th>
<th>%r</th>
<th>%t</th>
</tr>
</thead>
<tbody>
<tr>
<td>sieve()</td>
<td>(register int) i = 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(register int) anzahl = 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Local variable with type declaration

Display of all local variables of a function.

<table>
<thead>
<tr>
<th>E68::w.v.l</th>
<th>%m</th>
<th>%r</th>
</tr>
</thead>
<tbody>
<tr>
<td>sieve()</td>
<td>i = 6</td>
<td></td>
</tr>
<tr>
<td>anzahl = 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Local variables without type declaration

<table>
<thead>
<tr>
<th>E::w.v.l</th>
<th>%m.3</th>
<th>%r.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>func4(str = (word = 0x0 → NULL, count = 12345, left = 0x360 → (word = 0x0 → NULL, count = 12345, left = 0x360 → (word = 0x0, count = 12345, left = 0x360, right = 0x0, field1 = 1, field2 = 2), right = 0x0 → NULL)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Structured Variables

<table>
<thead>
<tr>
<th>E::w.v.l</th>
<th>%m</th>
<th>%t</th>
</tr>
</thead>
<tbody>
<tr>
<td>func6((auto double) a = 2.0, (auto double) b = 3.0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Function parameters

<table>
<thead>
<tr>
<th>E::w.v.l</th>
</tr>
</thead>
<tbody>
<tr>
<td>func5(?, ?, ?)</td>
</tr>
<tr>
<td>return = 11</td>
</tr>
</tbody>
</table>

Return value of a function

The variables can be modified by clicking with the mouse.

See also

- 'HLL Structures' in 'ICE User's Guide'
- 'Release Information' in 'Release History'
- 'Display Variables' in 'Training HLL Debugging'
Var.LOG

Log variables

Format: Var.LOG [%<format>] {<variable>} {<option>}

<option>:
ONBREAK
ONSPOT
ONTIME <time>
Timestamp
Changes
AREA <name>

ONFLAG (TRACE-ICE, TRACE32-FIRE only)
ONFLAG2 (TRACE-ICE, TRACE32-FIRE only)

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>

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONBREAK</td>
<td>Updates log each time the program execution is stopped. This is the default.</td>
</tr>
<tr>
<td>Changes</td>
<td>A log is only made when the variables have changed their value.</td>
</tr>
<tr>
<td>AREA &lt;name&gt;</td>
<td>Selects a different AREA for the logging.</td>
</tr>
<tr>
<td>ONSPOT</td>
<td>Update the log whenever a breakpoint specified with the Action Spot is hit and each time the program execution is stopped.</td>
</tr>
<tr>
<td></td>
<td>On the TRACE32-ICE the spot point system must be in Always stop mode.</td>
</tr>
<tr>
<td>ONTIME</td>
<td>Updates the log in a fixed time interval. This option requires run-time memory access to the variables.</td>
</tr>
<tr>
<td>Timestamp</td>
<td>Adds timestamps (absolute and relative) to the log. Mainly used together with ONTIME option.</td>
</tr>
<tr>
<td>Options for TRACE32-ICE and TRACE32-FIRE</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td></td>
</tr>
<tr>
<td><strong>ONFLAG</strong></td>
<td></td>
</tr>
<tr>
<td>Updates the log when the write flags of one of the variables are set. The flags are cleared after the variable is displayed. This allows an update of the log after a variable has been changed by the target processor.</td>
<td></td>
</tr>
<tr>
<td><strong>ONFLAG2</strong></td>
<td></td>
</tr>
<tr>
<td>Like <strong>ONFLAG</strong>, but the update is only made when the data was not overwritten during the read out of the variables. This makes that no wrong values are displayed when the values are changing during the dual-port memory read.</td>
<td></td>
</tr>
</tbody>
</table>

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
```

![Image: Log variable flags](image)

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
```

©1989-2019 Lauterbach GmbH
Example 3: Log variables every second.

```plaintext
; 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 '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 'Release History'
Var.NEWGLOBAL

Creates a global 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).

<table>
<thead>
<tr>
<th><code>&lt;variable_type&gt;</code></th>
<th>The following commands provide an overview of the supported variable types:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• The <code>sYmbol.List.BUILTIN</code> command lists the supported built-in variable types.</td>
</tr>
<tr>
<td></td>
<td>• The <code>sYmbol.List.Type</code> command lists variable types available after a target program has been loaded.</td>
</tr>
</tbody>
</table>

| `<variable_name>` | The TRACE32-internal variables must begin with a '\', 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


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'
Var.NEWLOCAL

Creates a local TRACE32-internal variable

Format: Var.NEWLOCAL [<variable_type>] \<variable_name>

|<variable_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).

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.

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.

```plaintext
;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 'Release History'
Var.PATtern

Display variables allowing wildcards for symbol name and type.

**Format:**

```markdown
Var.PATern [%<format>] [<symbol_pattern>] [<type_pattern>]
```

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

**Examples:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Var.PATtern target* struct</code></td>
<td>Display all variables whose name begins with &quot;target&quot; and which are of the type struct</td>
</tr>
<tr>
<td><code>Var.PATtern jpeg*</code></td>
<td>Display all variables whose name begins with &quot;jpeg&quot;</td>
</tr>
<tr>
<td><code>Var.PATtern %Type * struct struct?</code></td>
<td>Display all variables which are of the type struct struct?</td>
</tr>
</tbody>
</table>

![Example Image](image.png)

**See also**

- `Var`
- `Var.set`
- 'Release Information' in 'Release History'
Var.PRINT

Display variables

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.

\begin{description}
\item[CONTinue] Adds the string to the current output line in the selected AREA window or message line without inserting a newline character.
\item[<format>] Use the <format> parameters to print the variables in the desired format. For a description of the <format> parameters, click here.
\end{description}

Example:

\begin{verbatim}
Var.PRINT cstr1
Var.PRINT cstr1 " " mstatic1
Var.PRINT cstr1 ", " mstatic1
Var.PRINT "cstr1=" cstr1 ", mstatic1=" mstatic1
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{area.png}
\caption{AREA window displaying variables.}
\end{figure}

See also

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

▲ 'Release Information' in 'Release History'

©1989-2019 Lauterbach GmbH
Var.PROfile

Graphical display of variable

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

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>Restart display</td>
</tr>
<tr>
<td>Hold</td>
<td>Stop update/re-start update</td>
</tr>
<tr>
<td>AutoZoom</td>
<td>Best vertical scaling</td>
</tr>
</tbody>
</table>
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.set
- Data.PROfile

■ 'Release Information' in '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

■ 'HLL Structures' in 'ICE User's Guide'
■ 'Release Information' in 'Release History'
■ 'Display Variables' in 'Training HLL Debugging'
Var.set

Modify variable

<table>
<thead>
<tr>
<th>Format:</th>
<th><strong>Var.set</strong> [[%&lt;format&gt;]] &lt;expression&gt;</th>
</tr>
</thead>
</table>

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.

Example 1:

Example 2: Modification of arrays

Example 2:

//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

©1989-2019 Lauterbach GmbH
Example 3: Modifications of structures

\[
\text{Var.set } \text{ast} = (0x100, 6, 0x12345, 0x234, 3, 6)
\]

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

\[
\begin{align*}
\text{// Assign result of TRACE32 function } & \text{FILE.EXIST}(<\text{file}) \text{to variable } k \\
\text{Var.set } & k = \text{FILE.EXIST}("t32.men") \\
\text{// multiply variable } & k \text{ with the result of the TRACE32 function} \\
\text{// Register(<name>) and assign the result to variable } i \\
\text{Var.set } & i = k * \text{Register(R10)} \\
\text{// assign the result of the TRACE32 function } & \text{Data.Byte(<hex_address>) to} \\
\text{// the variable flags[3]} \\
\text{Var.set } & \text{flags[3]} = \text{Data.Byte(0x40004000)} \\
\text{// The following syntax is required if an } & \text{Access Class} \text { is required} \\
\text{// here the Access Class NoCache} \\
\text{Var.set } & \text{flags[3]} = \text{Data.Byte((NC:0x40004000))}
\end{align*}
\]

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

\[
\begin{align*}
\text{Var.set } & \% i \quad \text{//displays value of variable 'i' in} \\
& \text{//decimal, hex and ASCII} \\
\text{Var.set } & \% \text{String struct1} \quad \text{//displays structure 'struct1'.} \\
& \text{//displays character array included in the structure as strings}
\end{align*}
\]

See also
- Var
- Var.Break
- Var.DRAW
- Var.FixedTABle
- Var.Local
- Var.NEWLOCAL
- Var.AddWatch
- Var.Call
- Var.Dump
- Var.Go
- Var.LOG
- Var.PATtern
- Var.AddWatchPATtern
- Var.CHAIN
- Var.EXPORT
- Var.IF
- Var.NEW
- Var.PRINT
- Var.Assign
- Var.DelWatch
- Var.FixedCHAIN
- Var.INFO
- Var.NEWGLOBAL
- Var.PROfile
Var.Ref
Var.TYPE
Var.WRITE
Var.END()
Var.RANGE()
Var.VALUE()

Var.Step
Var.View
Var.ADDRESS()
Var.EXIST()
Var.SIZEOF()

Var.TABle
Var.Watch
Var.BITPOS()
Var.FVALUE()
Var.STRING()

Var.TREE
Var.WHILE
Var.BITSIZE()
Var.ISBIT()
Var.TYPEOF()

▲ 'HLL Structures' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
▲ 'Testing of Functions' in 'Training HLL Debugging'
Var.Step

- **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:**

\[
\text{Var.Step.BackChange} \ [<\text{expression}>]
\]

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

**Example:**

- \( \text{v.s.bc \ k} \) //steps till variable k changes
- \( \text{v.s.bc \ ptr->x} \) //steps till the contents of the structure pointed to by 'ptr' changes
- \( \text{v.s.bc \ flags} \) //steps till one element of the array 'flags' changes

---

### Var.Step.BackTill

**Step back till expression true**

**Format:**

\[
\text{Var.Step.BackTill} \ [<\text{expression}>]
\]

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

- \( \text{v.s.bt \ i>0x10} \) //steps till variable 'i' is larger than 10

---

- **See also**
  - Var.Step
  - 'Release Information' in 'Release History'
**Var.Step.Change**

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

**Examples:**

\[
\begin{align*}
\text{v.s.c k} & \quad \text{//steps till variable k changes} \\
\text{v.s.c ptr->x} & \quad \text{//steps till the contents of the structure pointed to} \\
& \quad \text{//by 'ptr' changes} \\
\text{v.s.c flags} & \quad \text{//steps till one element of the array 'flags' changes}
\end{align*}
\]

**See also**

- Var.Step
- ‘Real-time Emulation’ in 'ICE User's Guide'
- ‘Release Information’ in 'Release History'

---

**Var.Step.Till**

Step till expression true

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

**Example:**

\[
\begin{align*}
\text{v.s.t i>0x10} & \quad \text{//steps till variable 'i' is larger than 10}
\end{align*}
\]

**See also**

- Var.Step
- ‘Real-time Emulation’ in 'ICE User's Guide'
- ‘Release Information’ in 'Release History'
Var.TAble

Display table

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:

v.tab %l flags i j k vpchar
v.tab vpchar[0..100]  //'artificial' array build on a pointer

Displays the first expression as an array.

E::v.v.tab flags i j k vpchar

<table>
<thead>
<tr>
<th>Value</th>
<th>Flags</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0 (0)</td>
<td>1,</td>
<td></td>
</tr>
<tr>
<td>0x1 (1)</td>
<td>1,</td>
<td>← i</td>
</tr>
<tr>
<td>0x2 (2)</td>
<td>1,</td>
<td></td>
</tr>
<tr>
<td>0x3 (3)</td>
<td>0,</td>
<td></td>
</tr>
<tr>
<td>0x4 (4)</td>
<td>1,</td>
<td></td>
</tr>
<tr>
<td>0x5 (5)</td>
<td>1,</td>
<td>← vpchar</td>
</tr>
<tr>
<td>0x6 (6)</td>
<td>0,</td>
<td></td>
</tr>
<tr>
<td>0x7 (7)</td>
<td>1,</td>
<td></td>
</tr>
<tr>
<td>0x8 (8)</td>
<td>1,</td>
<td></td>
</tr>
<tr>
<td>0x9 (9)</td>
<td>0,</td>
<td></td>
</tr>
<tr>
<td>0x0A (10)</td>
<td>1,</td>
<td></td>
</tr>
<tr>
<td>0x0B (11)</td>
<td>1,</td>
<td>← k</td>
</tr>
</tbody>
</table>

pointers and indexes

See also

Var  Var.set

▲ ‘HLL Structures’ in 'ICE User’s Guide'
▲ ‘Release Information’ in 'Release History'
▲ ‘Display Variables’ in 'Training HLL Debugging'
Var.TREE
Display variables in the form of a tree structure

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 'Release History'
Var.TYPE

Display variable types

Format:

Var.TYPE [%<format>] [<expression>] …

<format>:

all
DEFault
Type [ .on  |  .OFF]
Open  [ .on  |  .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.

E::w.v.type %m struct1
struct1 struct1 struct(char * word,
   int count,
struct1 * left,
struct1 * right,
int field1:2,
unsigned int field2:3)

Display types of variables.

E::w.v.type %m %r %l %t ast
struct1 ast struct(20 bytes, [0] char * word (pointer to char, 32 bits),
   [4] int count (signed 32 bits),
   [8] struct1 * left (pointer to struct1, 32 bits
   [12] struct1 * right (pointer to struct1, 32 bi
   [16.0] int field1:2 (signed 32 bits),
   [16.2] unsigned int field2:3 (unsigned 32 bits)

See also

■ Var
■ Var.set

▲ 'HLL Structures' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'

©1989-2019 Lauterbach GmbH
Var.View

Display variables

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:

  ![Variable Display Window](image1)

  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:

  ![Memory Interpretation Window](image2)

- Modify a variable value or structure element:

  ![Variable Modification Window](image3)

  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

See also

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

▲ ‘HLL Structures’ in ‘ICE User’s Guide’
▲ ‘Release Information’ in ‘Release History’
▲ ‘Display Variables’ in ‘Training HLL Debugging’

©1989-2019 Lauterbach GmbH
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.

### 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

▲ 'HLL Structures' in 'ICE User's Guide'
▲ 'Release Information' in 'Release History'
▲ 'Display Variables' in 'Training HLL 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 'Release History'
Var.WRITE

Write variables to file

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 'Release History'

©1989-2019 Lauterbach GmbH
**Overview VCO**

The VCO may be used as a clock generator to support the emulation CPU, or may be used separately from the emulation system. The frequency range is from 2 to 70 (50) MHz. In response of the emulation adapter this frequency is divided by a fixed rate generating the CPU clock. The frequency control is made either on VCO or CPU clock frequency level. The step rate is 50 kHz on the VCO level. The VCO clock is ready on the BNC connector in rear of the ECU32 or ECC8 modul.

---

**NOTE:**

The VCO commands are not only available for FIRE and ICE, but also for the TRACE32 Instruction Set Simulator.

The **VCO.TimeBaseFrequency**, however, is only available for the TRACE32 Instruction Set Simulator.
**VCO.BusFrequency**  
Control bus clock  

FIRE / ICE only  

Format:  

```
VCO.BusFrequency <frequency>
```

Tbd.

See also  

- VCO  
- VCO.state

**VCO.Clock**  
Control emulation clock  

FIRE / ICE only  

Format:  

```
VCO.Clock <frequency> (deprecated)
```

Use **VCO.Frequency** instead.

`<frequency>`:  

```
1000Hz ... 2GHz
```

Sets the CPU clock frequency.

Examples:

```
VCO.Clock 16MHz ; set VCO clock to 16 MHz
VCO.Clock 4.77MHz ; set VCO clock to 4.77 MHz
VCO.Clock 8000KHz ; set VCO clock to 8 MHz
```

See also  

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

△ 'Emulator Functions’ in ‘FIRE User's Guide’  
△ 'Frequency Generator’ in ‘ICE User's Guide’

©1989-2019 Lauterbach GmbH
**VCO.Down**

**Frequency down**

**Format:**  
\[ \text{VCO.Down } [\text{<frequency>}] \]

\[ \text{<frequency>}: \quad 50000. \ldots \]

Step down with the VCO frequency.

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

**See also**

- \[ \text{VCO} \]
- \[ \text{VCO.Clock} \]
- \[ \text{VCO.Frequency} \]
- \[ \text{VCO.state} \]
- \[ VCO.Down \]
- \[ VCO.Up \]

- ‘Emulator Functions’ in ‘FIRE User’s Guide’
- ‘Frequency Generator’ in ‘ICE User’s Guide’

**VCO.Frequency**

**Control VCO clock**

**Format:**  
\[ \text{VCO.Frequency } \text{<frequency>} \]

\[ \text{<frequency>}: \quad 1000Hz \cdots 2GHz \]

**FIRE/ICE:** Sets the VCO frequency to the specified value.

**Simulator:** 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.

<table>
<thead>
<tr>
<th>&lt;frequency&gt;</th>
<th>All frequency definitions may be done in Hz, kHz, MHz or GHz.</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCO.Frequency 20MHz</td>
<td>; set VCO clock to 20 MHz</td>
</tr>
<tr>
<td>VCO.Frequency 10.5MHz</td>
<td>; set VCO clock to 10.5 MHz</td>
</tr>
<tr>
<td>VCO.Frequency 1800KHz</td>
<td>; set VCO clock to 1.8 MHz</td>
</tr>
</tbody>
</table>

**See also**

- \[ \text{VCO} \]
- \[ \text{VCO.Clock} \]
- \[ \text{VCO.Down} \]
- \[ \text{VCO.state} \]
- \[ \text{VCO.Up} \]
- \[ \text{VCO()} \]

- ‘Frequency Generator’ in ‘ICE User’s Guide’

©1989-2019 Lauterbach GmbH
VCO.Rate  
FIRE / ICE only

<table>
<thead>
<tr>
<th>Format:</th>
<th>VCO.Rate &lt;rate&gt;</th>
</tr>
</thead>
</table>

Defines the rate between VCO clock and internal CPU clock.

See also
- VCO
- VCO.state

VCO.RESet  
FIRE / ICE only

<table>
<thead>
<tr>
<th>Format:</th>
<th>VCO.RESet</th>
</tr>
</thead>
</table>

The VCO is initialized to the default frequency.

See also
- VCO
- VCO.state
- 'Emulator Functions' in 'FIRE User's Guide'
- 'Frequency Generator' in 'ICE User's Guide'
State display

**Format:** VCO.state

Displays the state of the VCO.

**See also**
- VCO
- VCO.BusFrequency
- VCO.Clock
- VCO.Down
- VCO.Frequency
- VCO.Rate
- VCO.RESet
- VCO.Up
- VCO.TimeBaseFrequency
- VCO.state

---

**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
VCO.Up

FIRE / ICE only

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

<frequency>: 1000Hz ... 2GHz

Step up with VCO clock.

**Example:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCO.Up</td>
<td>frequency up by 50 kHz</td>
</tr>
<tr>
<td>VCO.Up 1MHz</td>
<td>frequency up by 1 MHz</td>
</tr>
</tbody>
</table>

**See also**

- VCO
- VCO.Clock
- VCO.Down
- VCO.Frequency
- 'Emulator Functions’ in ‘FIRE User’s Guide’
- ‘Frequency Generator’ in ‘ICE User’s Guide’
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

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>VCU.Set &lt;register&gt; &lt;value&gt; [/&lt;option&gt;] [VCU&lt;instance&gt;]</code></th>
</tr>
</thead>
</table>

**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

<table>
<thead>
<tr>
<th>Format:</th>
<th><code>VCU.view [/&lt;option&gt;] [VCU&lt;instance&gt;]</code></th>
</tr>
</thead>
</table>

**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 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 to 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

'Ve.OFF' in '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
The **VPU** command group is used to display and modify the VPU (Vector Processing Unit) registers for PowerPC.

**See also**
- **VPU.Init**
- **VPU.Set**
- **VPU.view**
- **VPU()**
- ▲ 'VPU Functions’ in 'General Function Reference’

### VPU.Init
**Initialize ALTIVEC registers**

**Format:**

```plaintext
VPU.Init
VPU.RESet (deprecated)
```

Sets the ALTIVEC vector registers VR0-VR31, VRSAVE and VSCR to zero.

**See also**
- **VPU**
- **VPU.view**

### VPU.Set
**Modify ALTIVEC registers**

**Format:**

```plaintext
VPU.Set <register> [/<option>]
```

**<register>:**
- **VR0..VR31** `<value1>` [.. `<value4>`]
- **VRSAVE** `<value>`
- **VSCR** `<value>`

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.

**<option>**

For a description of the options, see Register.view.
Example:

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

PowerPC 74xx only

Format:  

VPU.view [/<option>]

Opens a window displaying the ALTIVEC vector registers VR0 to VR31, VRSAVE and VSCR.

For a description of the options, see Register.view.

See also
- VPU
- VPU.Init
- VPU.Set
  - ‘Release Information’ in ‘Release History’