

Training Simulator and Demo Software


Release 09.2023

MANUAL

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Training	
Training Simulator and Demo Software	1
About the Demo	3
Starting the TRACE32 Simulator	3
User Interface - TRACE32 PowerView	4
TRACE32 Command Line and Softkeys	6
Window Captions - What Makes Them Special in TRACE32	7
Debugging the Program	8
Basic Debug Commands	8
Debug Modes	9
Displaying the Stack Frame	11
Breakpoints	12
Setting Breakpoints	12
Listing all Breakpoints	13
Setting Read/Write Breakpoints	14
Variables	15
Displaying Variables	15
Displaying Variables of the Current Program Context	16
Using the Symbol Browser	16
Formatting Variables	17
Modifying Variables	18
Memory	19
Displaying Memory	19
Modifying Memory	20

About the Demo

What is this? This is a guided tour through TRACE32 - a tutorial. We use a simple program example in C to illustrate the most important debug features, and give lots of helpful tips & tricks for everyday use.

How long does this tutorial take? 0.5 to 1 hrs.

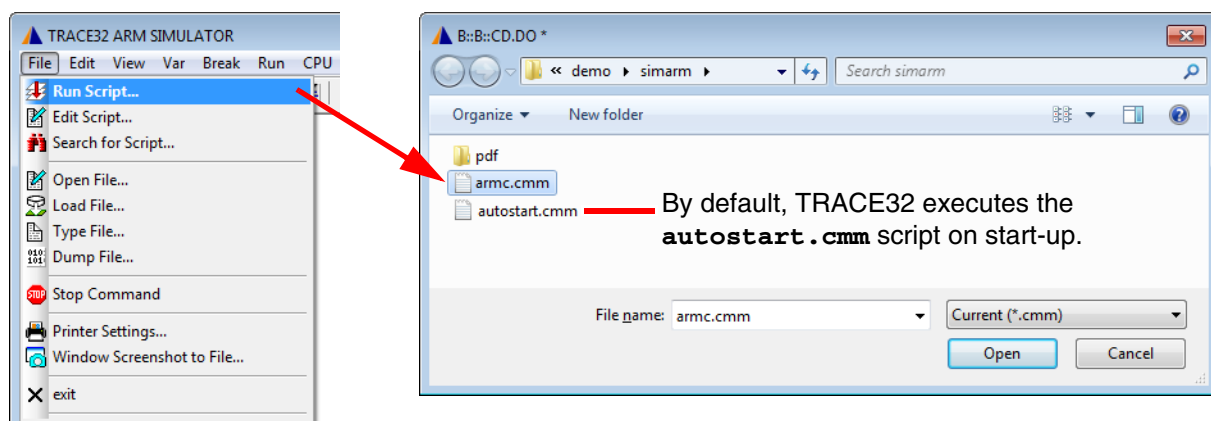
How can I learn most from this tutorial? Work completely through all chapters in sequence and then do the quiz at the end.

Where can I download the TRACE32 Simulator for the hands-on debug session? From: <https://www.lauterbach.com/download.html>. You do not need any hardware for this tutorial.

Starting the TRACE32 Simulator

1. Unzip the downloaded file. You do not need to install the TRACE32 Simulator.
2. Double-click the `t32m<architecture>.exe` file (e.g. `t32marm.exe`) to start the demo debug session. When the TRACE32 Instruction Set Simulator starts, a start-up PRACTICE script that sets up a debug session is automatically executed.

You can manually execute the same start-up PRACTICE script by choosing **File menu > Run Script**.



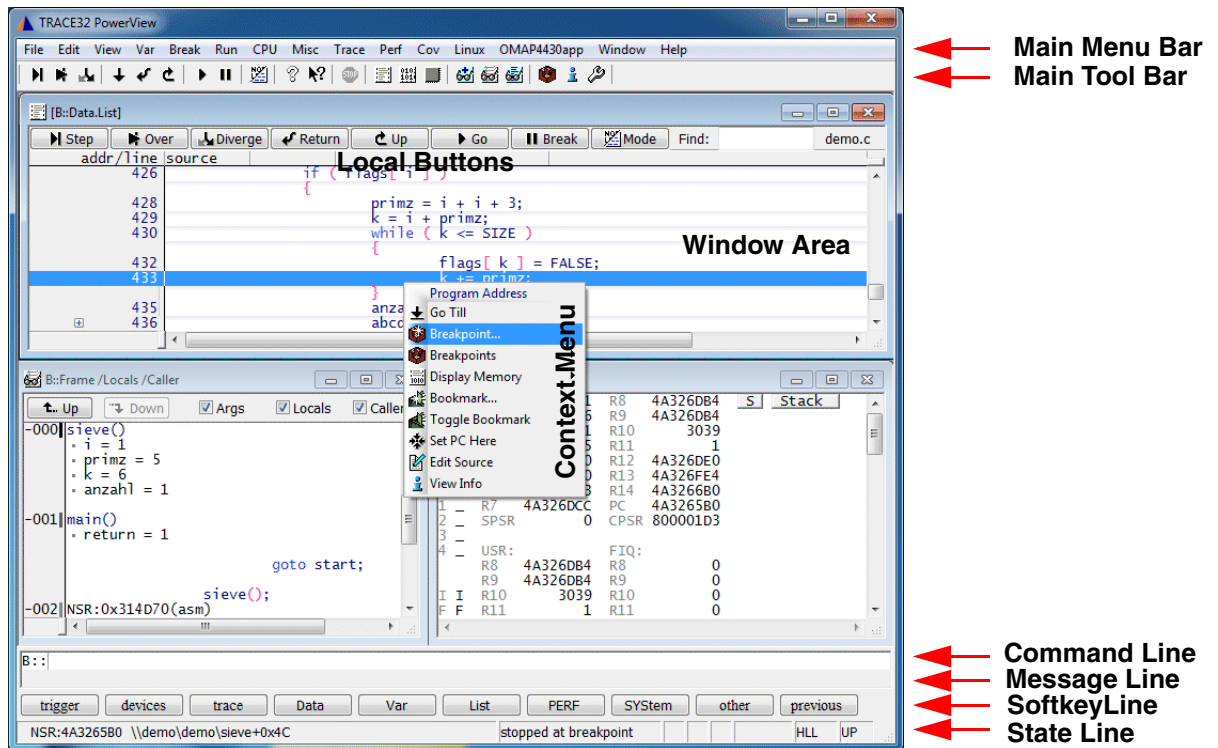
PRACTICE, the Lauterbach script language, is used for automating tests, configuring the TRACE32 PowerView GUI and your debug environment.

For our demo debug session, the PRACTICE start-up script `armc.cmm` loads the application program `armle.axf` and generates a TRACE32 internal symbol database out of the loaded information.

User Interface - TRACE32 PowerView

The graphical user interface (GUI) of TRACE32 is called TRACE32 PowerView.

The following screen shot presents the main components of the user interface.



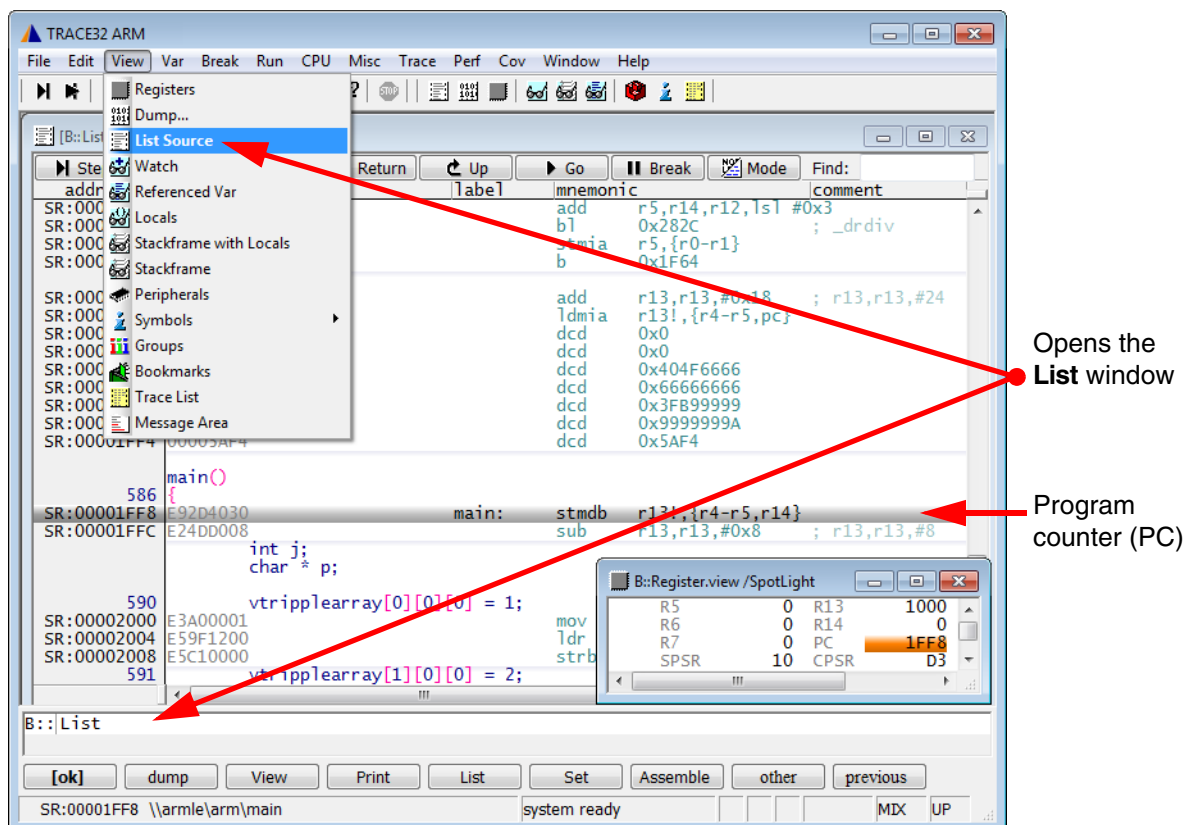
We'll briefly explain the GUI using the **List** command and **List** window as an example.

A video tutorial about the TRACE32 PowerView GUI is available here:
support.lauterbach.com/kb/articles/introduction-to-trace32-gui

Do one of the following to open the **List** window:

- Choose **View** menu > **List Source**
- or, at the TRACE32 command line, type: **List** (or **L**)

The **List** window displays the code in assembler mnemonic and HLL (HLL stands for High-Level Language and means the programming language of your source code).



In the **List** window, the gray bar indicates the position of the program counter (PC). Right now, it is located on the symbolic address of the label **main**.

A video tutorial about the source code display in TRACE32 is available here:
support.lauterbach.com/kb/articles/displaying-the-source-code

To summarize it, you can execute commands in TRACE32 PowerView via the usual suspects:

1. Menus on the menu bar
2. Buttons on the main toolbar and the buttons on the toolbars of TRACE32 windows
3. Context menus in TRACE32 windows

Additionally in TRACE32, you can execute commands via the TRACE32 command line.

TRACE32 Command Line and Softkeys

TRACE32 commands are **not** case sensitive: **register.view** is the same as **Register.view**

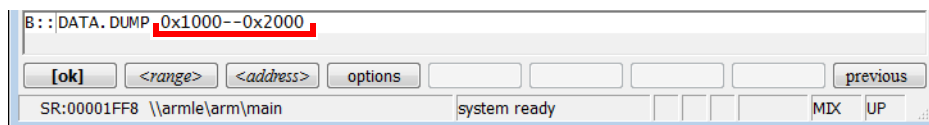
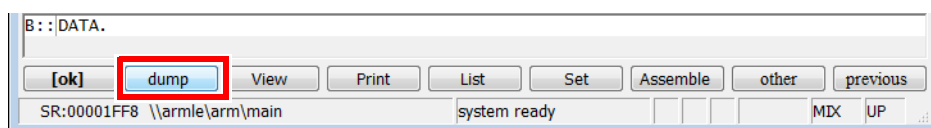
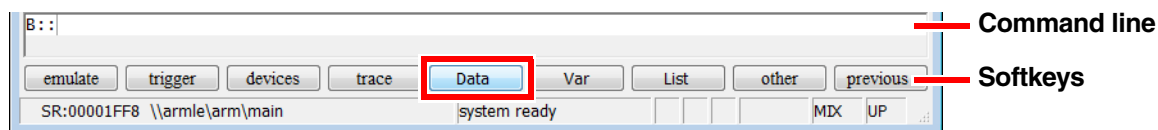
UPPER CASE letters indicate the short forms of commands and must not be omitted. All lower case letters can be omitted. This makes short forms an efficient time saver when you are entering frequently-used commands in the command line. Examples:

- Instead of the long form **Register.view** type just the short form **r** or **R**
- Instead of the long form **List** type just the short form **l** or **L**

The softkeys are below the command line. The camel casing (i.e. upper and lower case letters) on any softkey tells you the long form of a command. The softkeys guide you through the command input, displaying all possible commands and parameters.

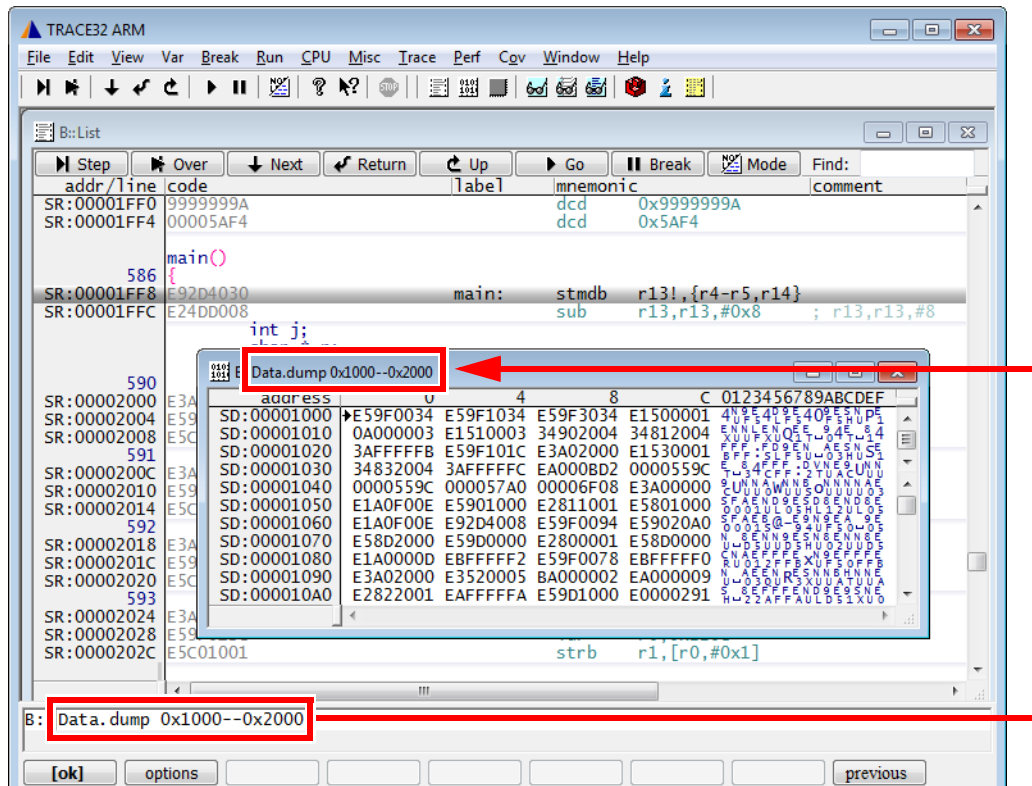
Example - To assemble the Data.dump command using the softkeys:

1. Click **Data**.
2. Click **dump**.
3. Type the **<range>** or **<address>** you want to dump. For example, **0x1000--0x2000**
4. Click **[ok]** to execute the command. The **Data.dump** window opens.



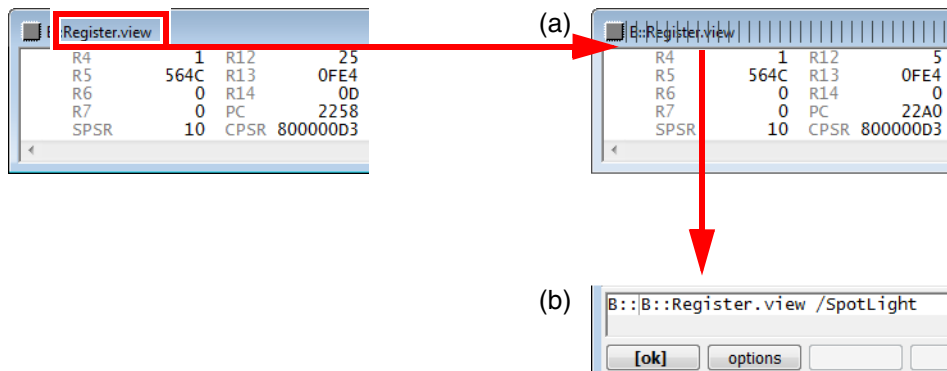
Window Captions - What Makes Them Special in TRACE32

The command with which you open a window will be shown as the window caption. The parameters and options are also included in the window caption.



You can **re-insert** a command from a window caption (a) into the command line (b) in order to modify the command. Let's do this with the **Register** window.

1. Choose **View** menu > **Register**.
2. Right-click the window caption (a).
3. Modify the command, e.g. by adding the **/SpotLight** option: It will highlight changed registers.




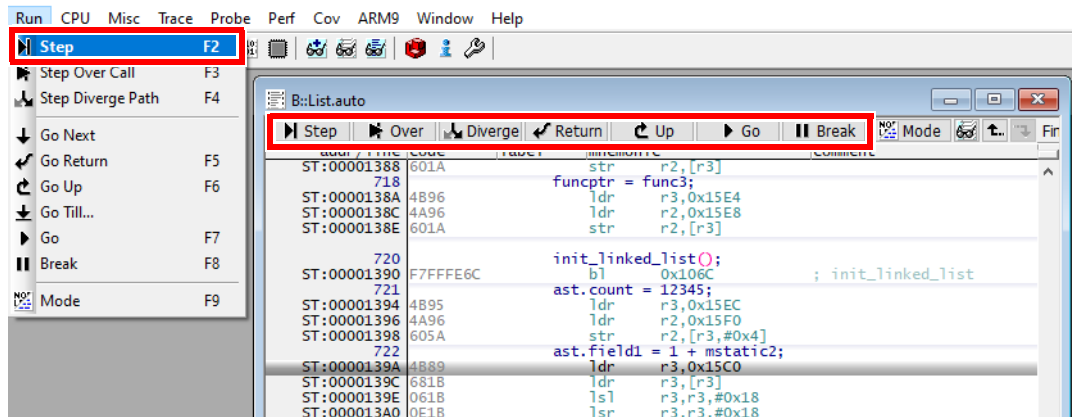
4. Click **[ok]** to execute the modified command.
5. Click **Single Step** on the TRACE32 toolbar. Changed registers are highlighted immediately.

Debugging the Program

Basic Debug Commands

The basic debug commands are available via the **Run** menu, the toolbar of the **List** window, the main toolbar, and via the TRACE32 command line.

Single stepping  is one of the basic debug commands.

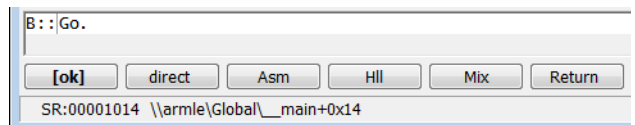
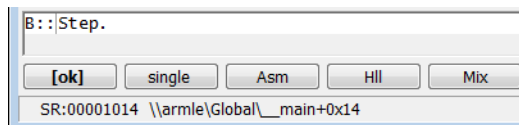
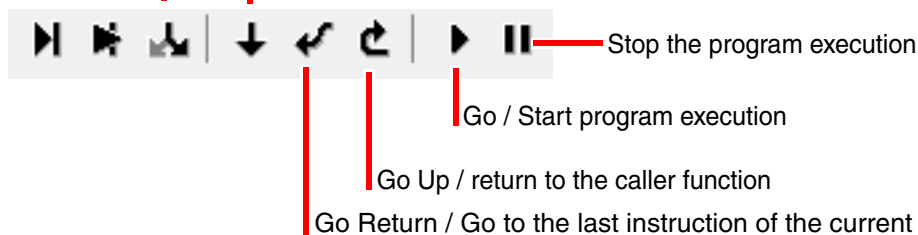


Single Step

Step over function calls or subroutines

Step till next unreachable line

Go to the next code line written in the program listing
Useful e.g. to leave loops

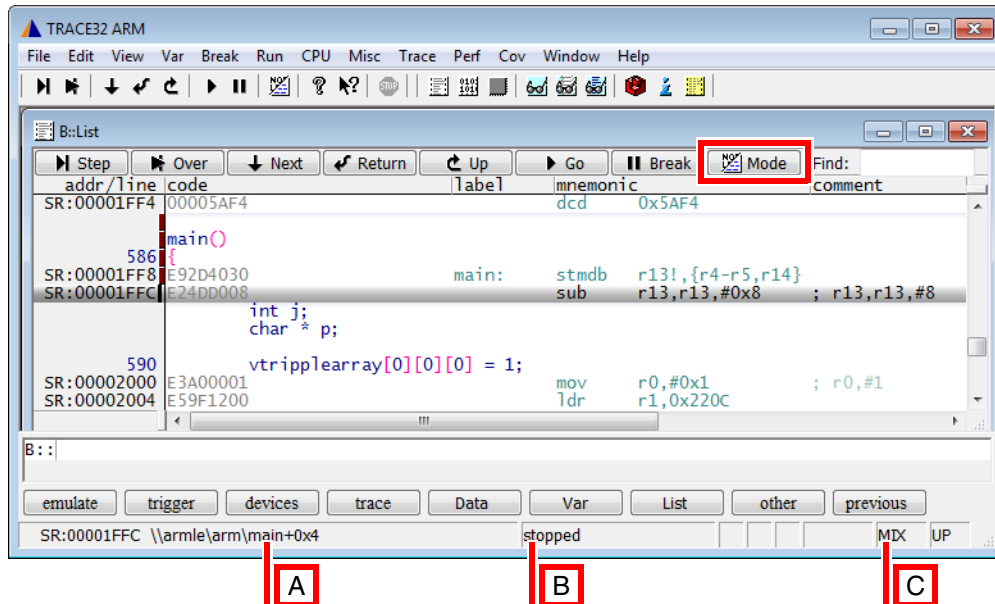


TRACE32 provides also more complex debug control commands. You can step until an expression changes or becomes true.

Example: **Var.Step.Till i>11.** single-steps the program until the variable **i** becomes greater than **11**. Please note that TRACE32 uses a dot to denote decimal numbers.

Debug Modes

Take a look at the state line at the bottom of the TRACE32 main window:

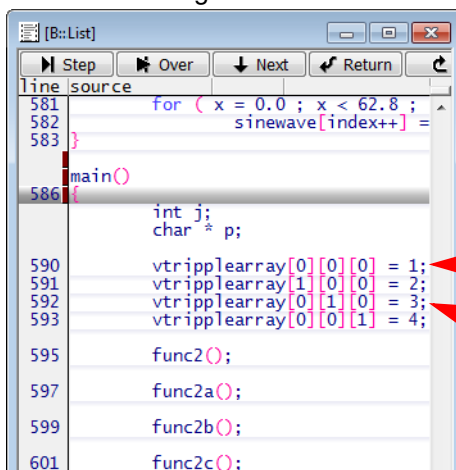


The state line tells you:

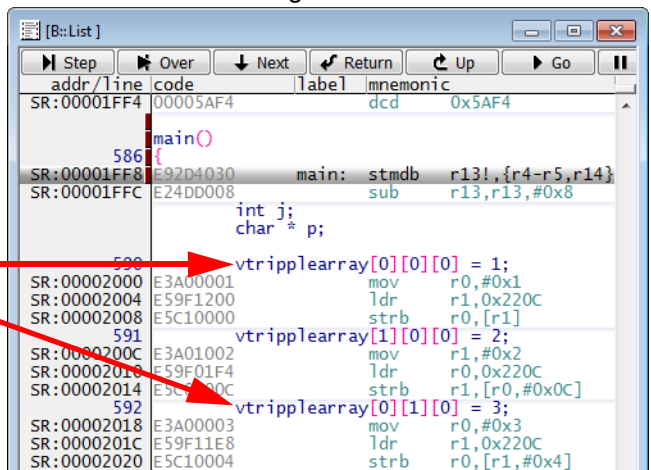
- A The (symbolic) address of the current cursor position. The program counter (PC) is highlighted in gray.
- B The state of the debugger: **stopped** means program execution is stopped. You can now, for example, inspect or change memory.
- C The state line displays the currently selected debug mode: The code display will be **HLL** (High Level Language) or **ASM** (assembler) or a **MIXed** mode with HLL and its corresponding assembler mnemonic.




6. On the toolbar of the **List** window, click  **Mode** to toggle the debug mode to **HLL**.

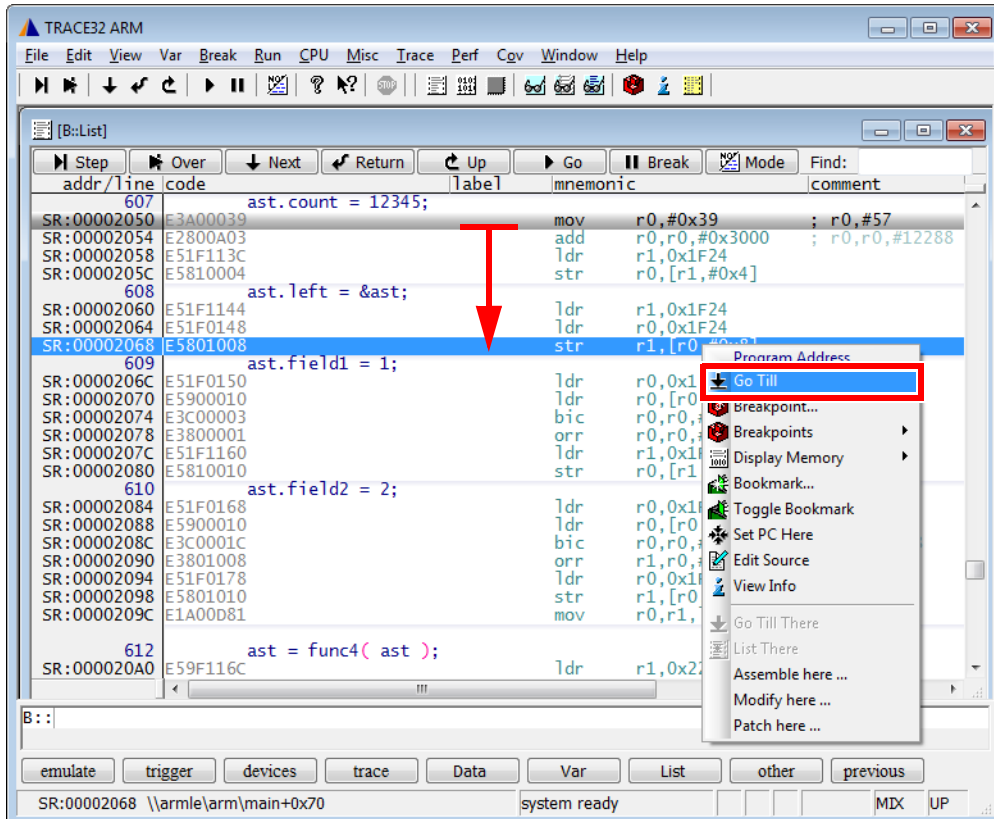
Debug mode HLL



Debug mode MIX

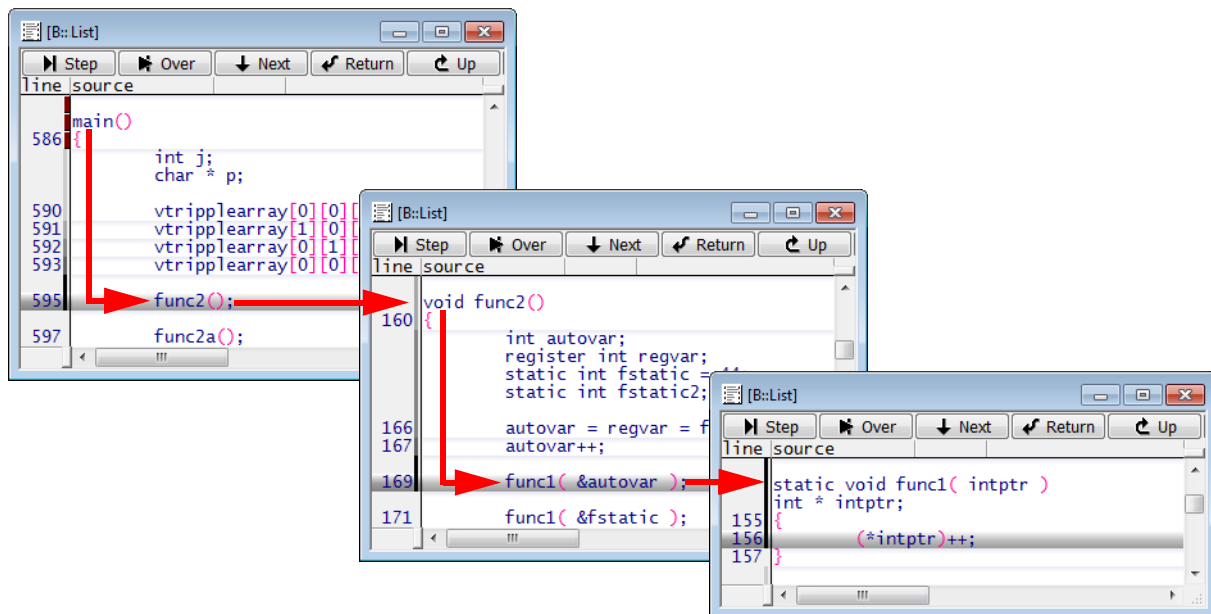


7. Click  **Step**.
The step you are taking is debug mode HLL goes to the next source code line.
8. Click  **Mode** again to toggle the debug mode to **MIX**.
9. Click  **Step**.
This time, the step executes one assembler line.
10. Right-click a code line, and then select **Go Till**.
The program execution starts. It stops when the program reaches the selected code line.



Displaying the Stack Frame

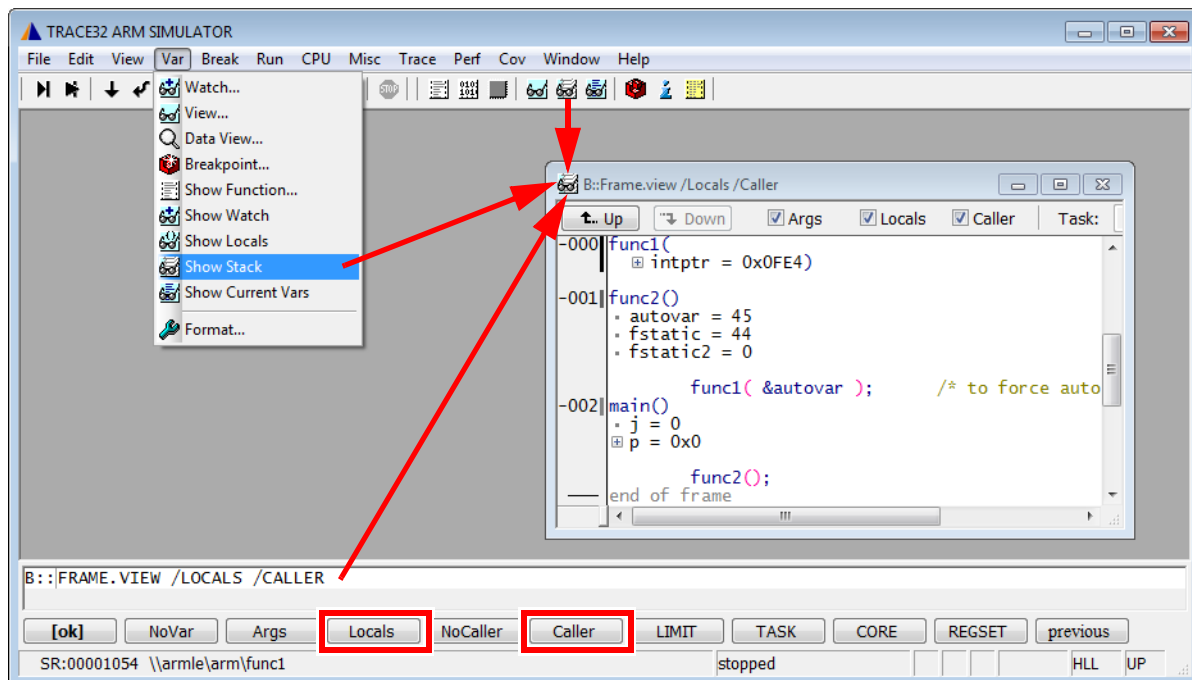
For the following example, let's assume we have the following call hierarchy: **main()** calls **func2()** and **func2()** calls **func1()**:



Choose **Show Stack** in the **Var** menu. The **Frame.view** window displays the call hierarchy.

- The **/Locals** option shows the local variables of each function.
- The **/Caller** option shows a few source code lines to indicate where the function was called.

This screenshot corresponds to the calling hierarchy shown above.



Breakpoints

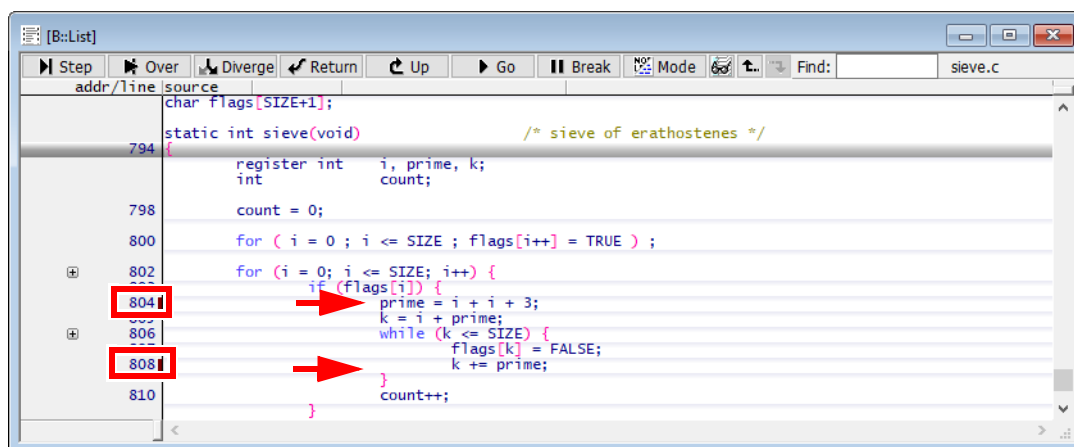
Video tutorials about breakpoints in TRACE32 are available here:
support.lauterbach.com/kb/articles/using-breakpoints-in-trace32

Setting Breakpoints

Let's set a breakpoint to the instruction `prime = i + i + 3` and the instruction `k += prime`

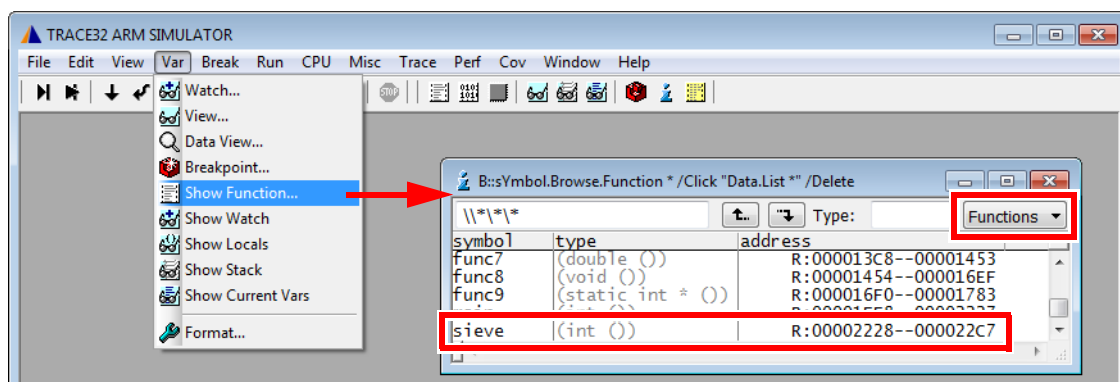
1. Double-click a code line to set a program breakpoint.
2. Make sure to click the white space in the code line, and not the code literal.

All code lines with a program breakpoint are marked with a red vertical bar.



To set a breakpoint to an instruction that is not in the focus of the current source listing

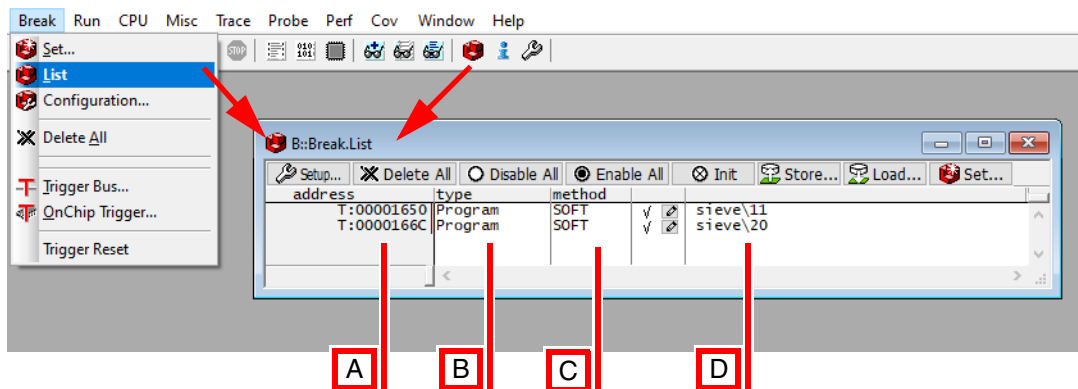
1. Choose **Var** menu > **Show Function**.
The **sYmbol.Browse.Function** window opens.



2. Select the function you are interested in e.g. **sieve**.
The **List** window opens, displaying this function. This window is now fixed to the start address of the function sieve and does not move with the program counter cursor.

Listing all Breakpoints

1. Choose **Break** menu > **List** to list all breakpoints.
The **Break.List** window opens, providing an overview of the set breakpoints.



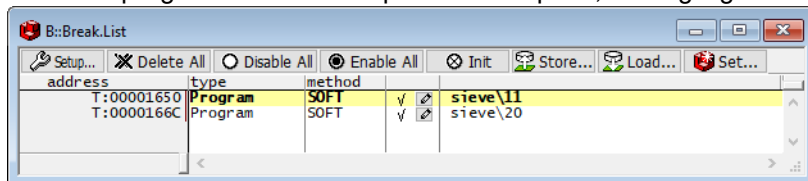
- A Address of the breakpoint.

B Breakpoint type.

C Breakpoint method: SOFTWARE, ONCHIP or DISABLED.

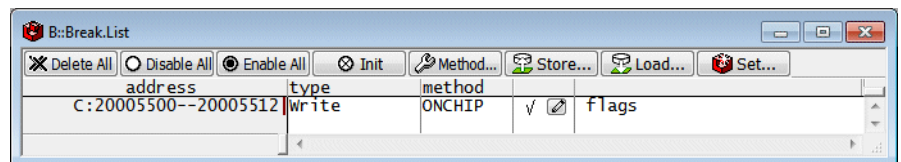
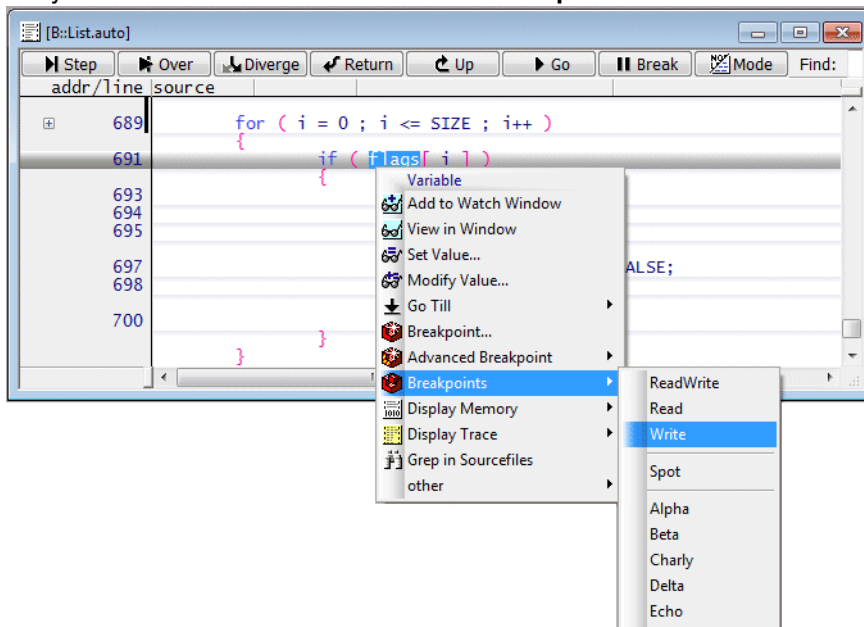
D Symbolic address of the breakpoint. Example:

 - **sieve\11** means source code line 11 in function **sieve**.
2. On the toolbar, click **Go** to start the program execution.
3. When the program execution stops at a breakpoint, it is highlighted in the **Break.List** window.



Setting Read/Write Breakpoints

You can set a breakpoint that stops the program execution at a read or write access to a memory location (e.g. global variable). To set a breakpoint on the array `flags` for instance, do a right mouse click on the array name in the **List** window then select **Breakpoints > Write**.



Variables

Video tutorials about variable display in TRACE32 are available here:

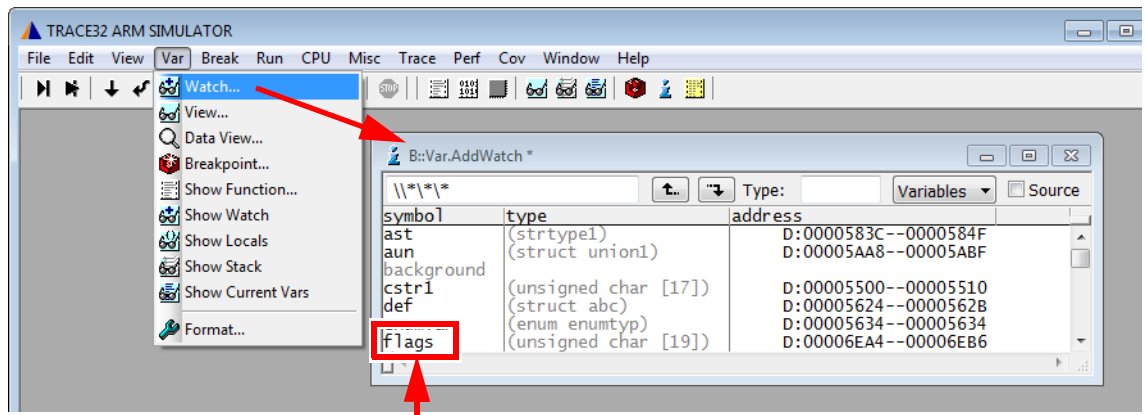
support.lauterbach.com/kb/articles/variable-logging-and-monitoring-in-trace32

Displaying Variables

Let's display the variables **flags**, **def**, and **ast**.

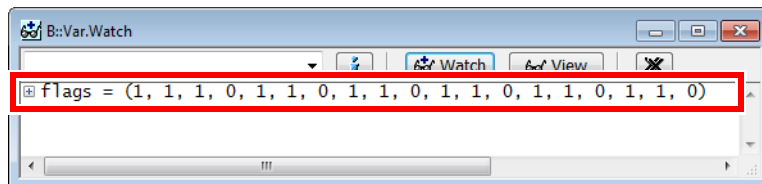
1. Choose **Var** menu > **Watch...**

The **Var.AddWatch** window opens, displaying the variables known to the symbol database.



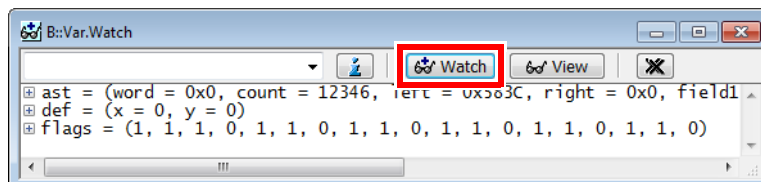
2. Double-click the variable **flags**.

The **Var.Watch** window opens, displaying the selected variable.



3. **Alternative steps:**

- In the **Var.Watch** window, click **Watch**, and then double-click the variables **def** and **ast** to add them to the **Var.Watch** window.



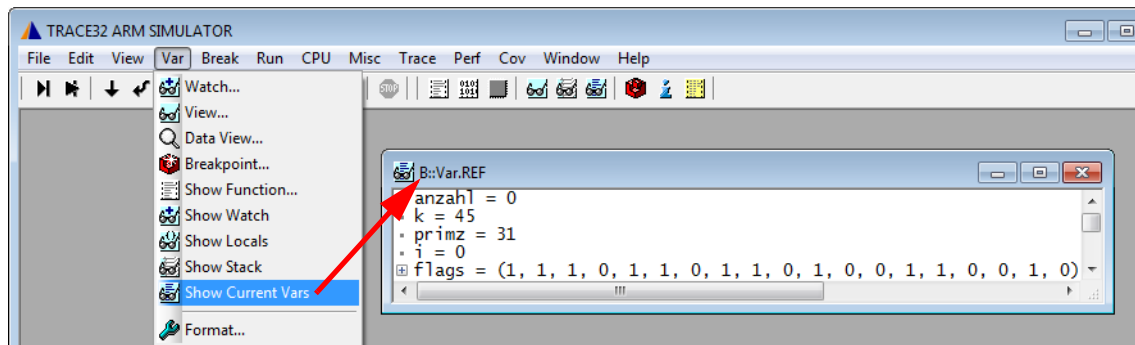
- From a **List** window, drag and drop any variable you want into the **Var.Watch** window.
- In a **List** window, right-click any variable, and then select **Add to Watch window** from the context menu.
- If you want to display a more complex structure or an array in a separate window, choose **Var** menu > **View**.

Displaying Variables of the Current Program Context

1. Set the program counter (PC) to `sieve()` by typing at the TRACE32 command line:

```
Register.Set PC sieve ;The command short form is: r.s pc sieve
```

2. Choose **Var** menu > **Show Current Vars**.
The **Var.REF** window opens, displaying all variables accessed by the current program context.

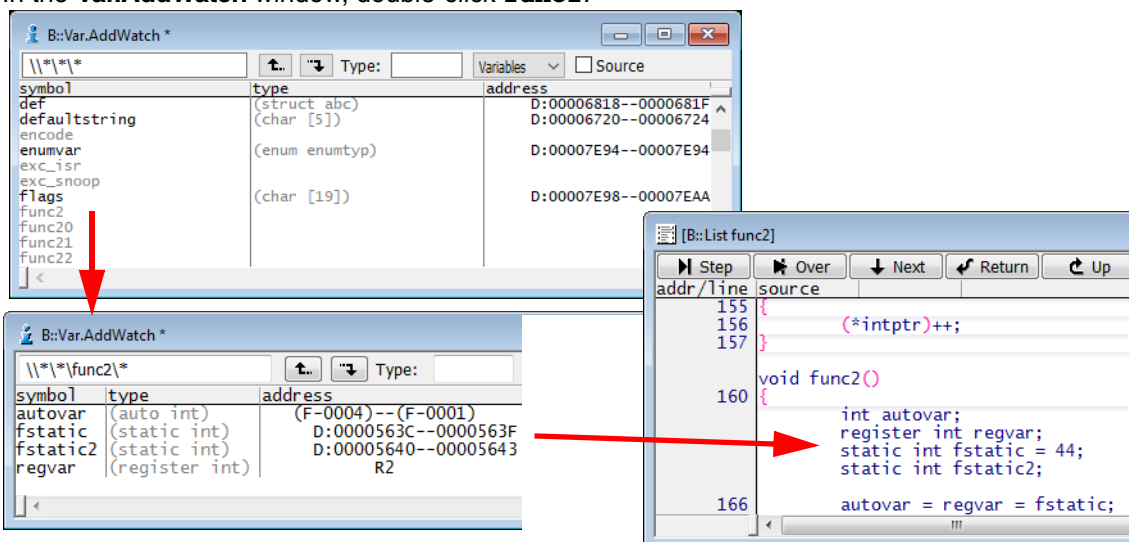


3. Click **Step** on the TRACE32 toolbar to execute a few single steps.
The **Var.REF** window is updated automatically.

Using the Symbol Browser

The symbol browser provides an overview of the variables, functions, and modules currently stored in the symbol database.

1. Choose **Var** menu > **Watch...**
The **Var.AddWatch** window lets you browse through the contents of the symbol database. Global variables are displayed in black and functions in gray. By double-clicking a function, its local variables are displayed.
2. In the **Var.AddWatch** window, double-click **func2**.



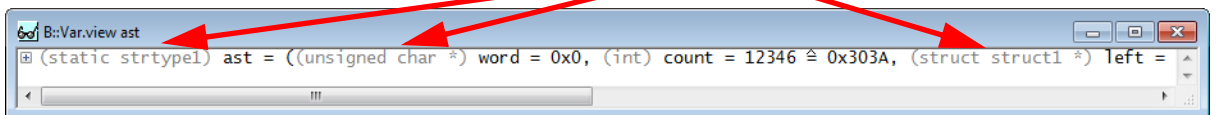
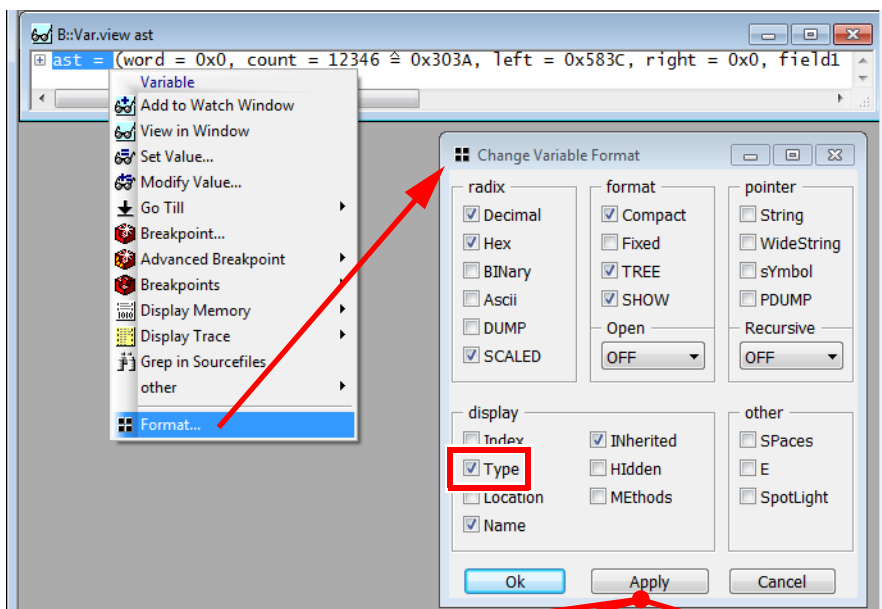
Formatting Variables

To format the display of variables - global settings:

1. Choose **Var** menu > **Format**.
2. In the **SETUP.Var** window, make your settings. **Decimal** and **Hex** are useful global settings. TRACE32 applies your settings to all **Var.view** windows that you open *afterwards*.

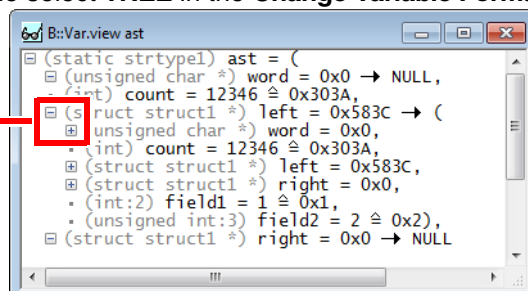
To format the display of an individual variable:

1. At the command line, type: **Var.view ast** (The variable **ast** is included in this demo.)
2. In the **Var.view** window, right-click **ast**, and then click **Format**. The **Change Variable Format** dialog opens.
3. Select the **Type** check box to display the variable **ast** with the complete type information.
4. Click **Apply**. The format of **ast** in the **Var.view** window is updated immediately.



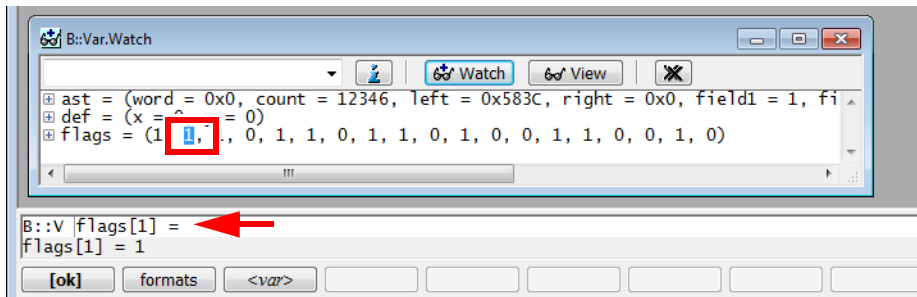
5. For more complex variable select **TREE** in the **Change Variable Format** dialog box.

Click + and - to expand and collapse the tree.





Modifying Variables

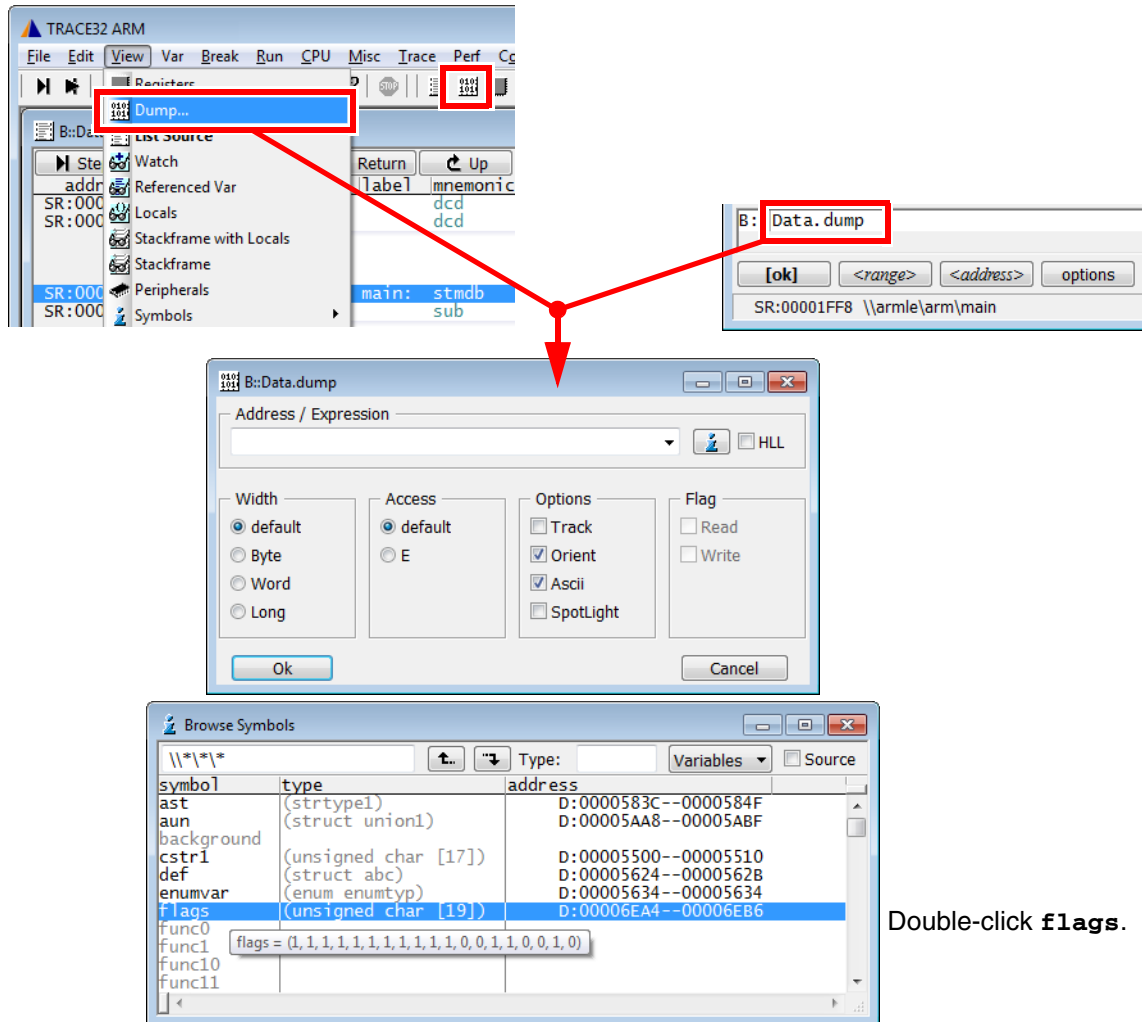
1. Double-click the variable value to modify the value. The **Var.set** command will be displayed in the command line. The short form of the command is **v** or **▼**



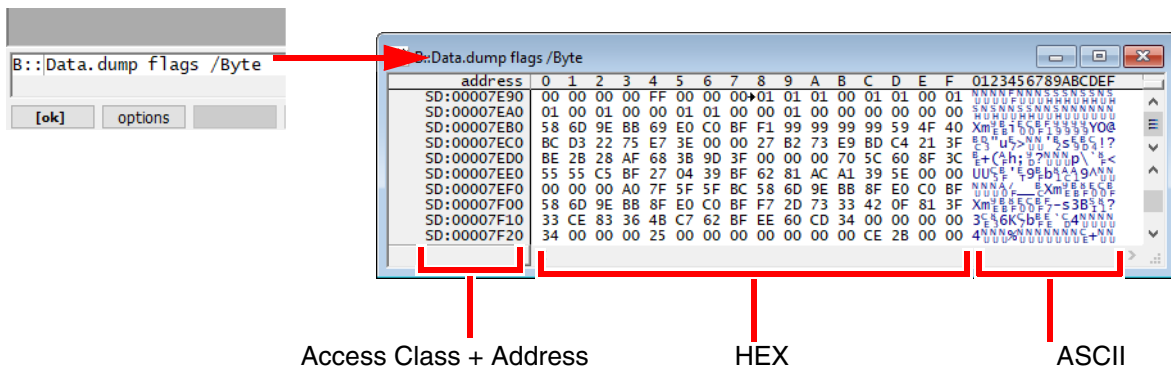
2. Enter the new value directly after the equal sign and confirm with **[ok]**.

Displaying Memory

1. To display a memory dump in a **Data.dump** window, do one of the following:
 - Choose **View** menu > **Dump**,
 - or click  **Memory Dump** on the toolbar,
 - or, at the TRACE32 command line, type: **Data.dump**
You can also specify an address or symbol directly, e.g.: **Data.dump flags**
2. In the **Data.dump** dialog, enter the data item, e.g. **flags**
 - Alternatively click  to browse through the symbol database.
3. In the **Browse Symbols** window, double-click the symbol **flags** to select it, and then click **OK**.



In the following screenshot, the **Data.dump** window is called via the TRACE32 command line.



There are different ways to define an address range:

- `<start_address>--<end_address>` (SD is an [access class](#))

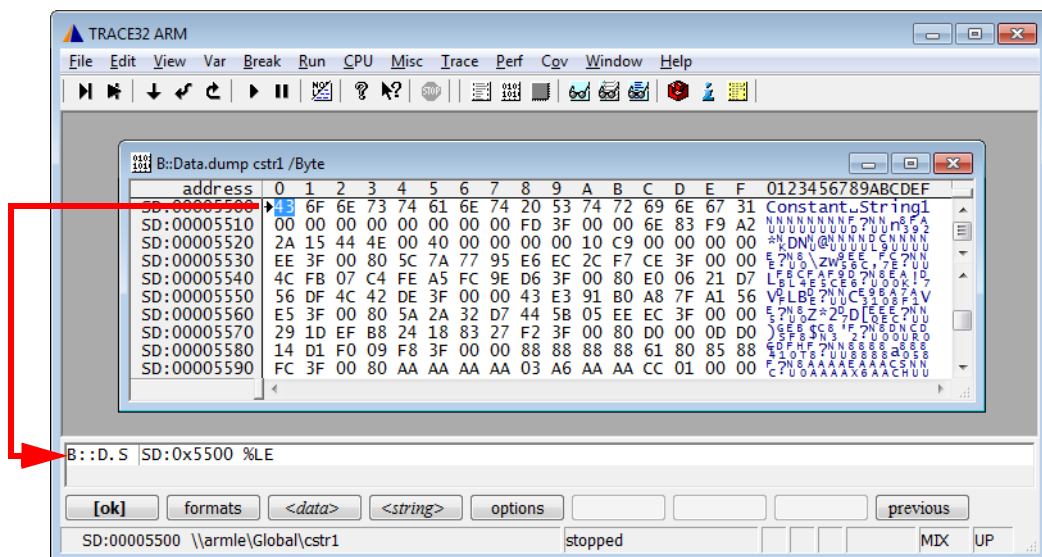
```
Data.dump SD:0x5530--SD:0x554F
```

- `<start_address>++<offset>`

```
Data.dump cstr1++0x1f /Byte ;start at cstr1 plus the next 0x1f bytes
```

Modifying Memory

1. In a **Data.dump** window, double-click the value you want to modify. A **Data.Set** command for the selected address is displayed in the command line. The short form of the command is **D.S** or **d.s**



2. Enter the new value directly after **%LE**, and then confirm with **[ok]**. (**%LE** stands for Little Endian).

