Debugger Tutorial

Release 02.2024

Debugger Tutorial

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Debugger Getting Started ....

Debugger Tutorial 1

About the Tutorial 4

Set up the TRACE32 Environment 6

Set Up the Debug Environment 7

The Welcome Dialog 7

A Typical Set Up Procedure 8

How to Generate TRACE32 Support Information 10

Start-Up Scripts 11

Write a Start-Up Script 11

Run a Start-up Script 12

Automated Start-up Scripts 13

User Interface - TRACE32 PowerView 14

TRACE32 Command Line and Softkeys 16

Window Captions - What Makes Them Special in TRACE32 17

Debugging the Program 18

Basic Debug Commands 18

Debug Modes 19

Displaying the Stack Frame 21

Breakpoints 22

Setting Breakpoints 22

Listing all Breakpoints 23

Setting Read/Write Breakpoints 24

Variables 25

Displaying Variables 25

Displaying Variables of the Current Program Context 26

Using the Symbol Browser 26

Formatting Variables 27

Modifying Variables 28

Memory 29

Displaying Memory 29

Modifying Memory 30

Exit TRACE32 31
Getting Online Help **32**

Debugger Tutorial

Version 04-Mar-2024

About the Tutorial

What is it about?

This is a tutorial for all hardware-assisted TRACE32 Debuggers (TRACE32-ICD) that are implemented using an on-chip debug interface. The most common used on-chip debug interface is JTAG.

Preconditions:

The tutorial assumes that the TRACE32 debugger software is already installed. Please refer to **"TRACE32 Installation Guide"** (installation.pdf) for information about the installation process. Also a basic knowledge of software debugging and the C-programming language is helpful. This is required in order to be able to follow the example code found in this tutorial. Moreover, a basic knowledge of the target processor and used assembler/compiler is necessary to get your debug system running.

To work with a hardware-assisted TRACE32 Debugger (ICD) a working target system is required. Please also take care of the proper sequence on powering up/down:

- Power Up: debugger - target
- Power down: target - debugger

Purpose of this tutorial:

The purpose of this tutorial is to do the basic steps in setting up the debug environment and to make you familiar with the basic features of TRACE32. For simplicity, we use in this tutorial a single-core system example.

How to use this tutorial:

The tutorial contains a guided debug session. It uses a simple C-program example to show you the basic debug features. You should perform a number of exercises as you read this tutorial. We recommend to go completely through all chapters.

Where can I get more information:

A detailed overview of all debug features offers **"Training Basic Debugging"** (training_debugger.pdf).

The TRACE32 PowerView GUI contains a detailed online help that offers descriptions of all debug features. Refer to the "**Online Help"** chapter on how to start and use the online help system.

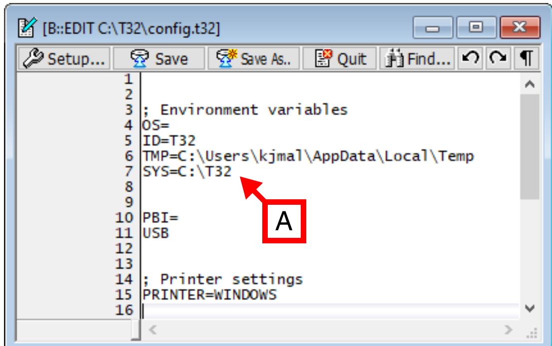
How long does it take?

60 minutes

Set up the TRACE32 Environment

After installing the TRACE32 on your host PC, a default environment is set up. This configuration can be adapted to your debugging environment.

By default the configuration file **`config.t32`** in the system directory (e.g. **`C:\T32`** or **`/opt/t32`**) is used. The option **`-c`** allows you to define your own location and name for the configuration file.



The TRACE32 executables are named *t32m<architecture>*[*.exe*] (e.g. t32marm.exe in our example) and are placed in the TRACE32 system directory (**SYS=**, see **[A]**) under **`bin\`*`<os>`***
(e.g. **`bin\windows64`** or **`bin/pc_linux64`**).

Example for Windows:

```
C:\T32\bin\windows64\t32marm.exe -c C:\workspace\config_user.t32
```

Example for Linux:

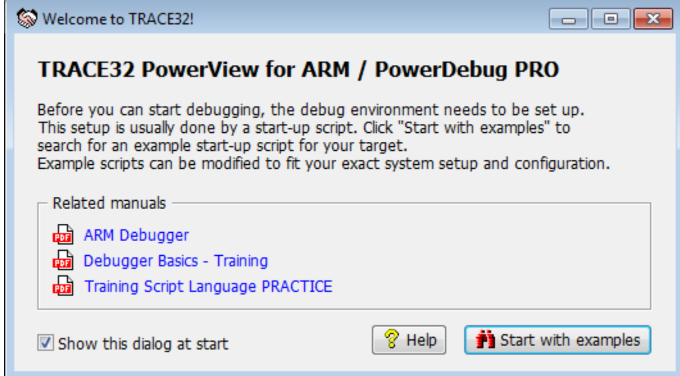
```
/opt/t32/bin/pc_linux64/t32marm -c /home/user/config_user.t32
```

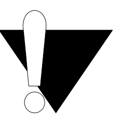
Set Up the Debug Environment

In order to set up your debugger, you need some knowledge about your CPU and your target configuration. A basic start-up procedure and the CPU specific setting for the debugger are described in the **Processor Architecture Manual** that can be opened by selecting the menu **Help > Processor Architecture Manual**.

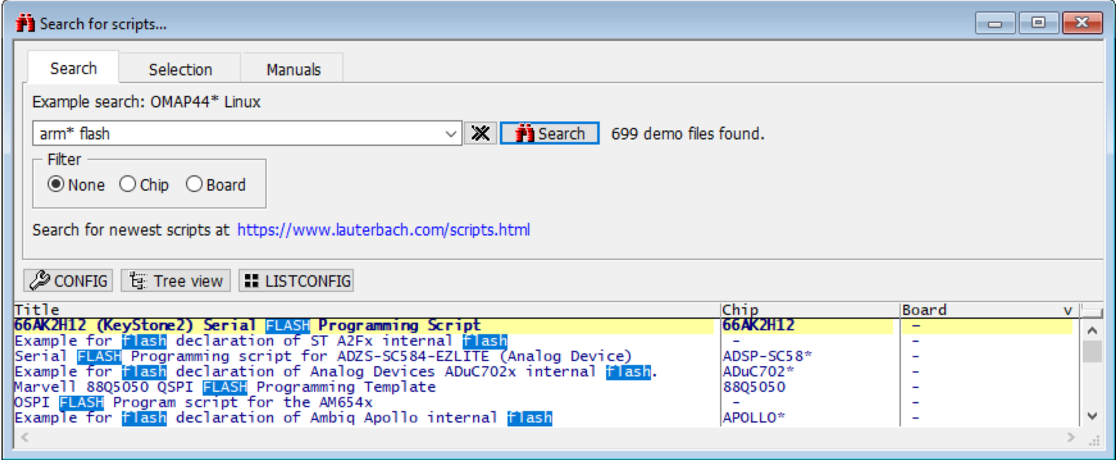
The Welcome Dialog

Per default a **Welcome to TRACE32!** dialog is displayed when TRACE32 PowerView is started. This dialog shows the target architecture and debug module. Additionally, the dialog includes links to most important manuals.



	Please make sure that you have not started, by mistake, a **TRACE32 Instruction Set Simulator** instead of an **Debugger**. You would see in this case the message **"TRACE32 PowerView for ... / Simulator"** in the **Welcome to TRACE32!** dialog.

Lauterbach provides ready-to-run PRACTICE start-up scripts for public known architecture hardware. You can search for PRACTICE scripts by pushing the **Start with examples** button from the **Welcome to TRACE32!** dialog.

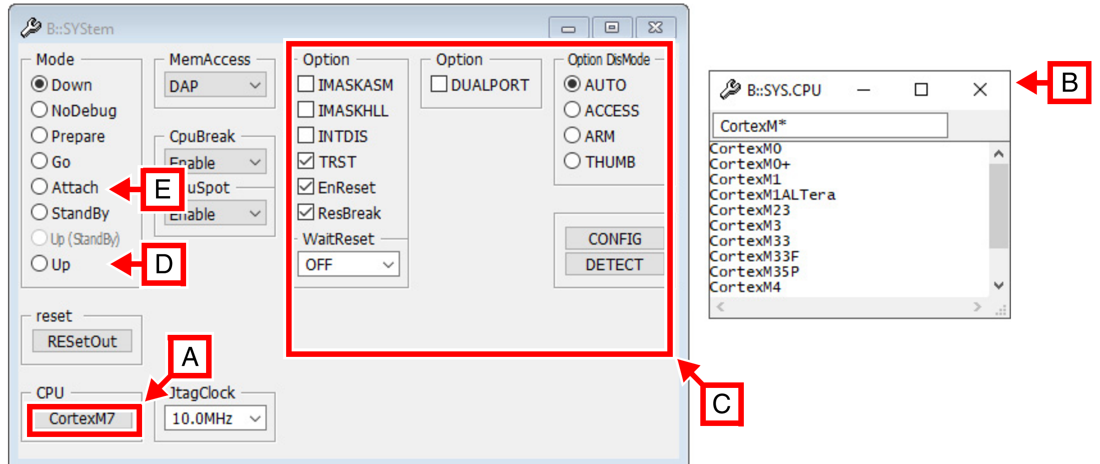


You can also inspect the demo directory manually from the TRACE32 system directory.

A Typical Set Up Procedure

This chapter describes a typical start-up procedure for the debugger. To demonstrate the steps needed, we will do a manual setup. Later on we will show you how to use PRACTICE scripts (*.cmm) for this purpose. For simplicity, we use here a single-core system example. The ***SYStem*** Window provides all CPU specific settings. You can open this window by selecting the menu **CPU > SYStem Settings...**

1. Inform TRACE32 about the core/chip on your target **[A, B]**, if an automatic detection is not possible.



SYStem.DETECT CPU	Auto detection of CPU
SYStem.CPU *<cpu>*	Select the CPU/chip

2. Some cores require additional settings before the communication can be established. You can set these options from the **SYStem** windows **[C]**. For details, refer to the **Processor Architecture Manual**.

3. Establish the communication between the debugger and the core. The most common way is to select the mode ***Up*** **[D]**.

   If ***Up*** is selected, the following steps are performed:

   - Reset of the core.
   - Initialization of the communication between the debugger and the core.
   - Stop of the core at the reset vector, if supported by the core in use.

SYStem.Up	Establish the communication between the debugger and the core

   A second useful way to establish the communication between the debugger and the core is ***Attach*** **[E]**. ***Attach*** allows to connect the debugger to an already running core.

SYStem.Mode Attach	Establish the communication between the debugger and the target core (without reset)

   If you get an error after selecting ***Up*** or ***Attach***, refer to the **Processor Architecture Manual**.

4. The next step is to download your application into the target:

   - if the application should run out of RAM then you can directly use the **Data.LOAD** command. Just type **Data.LOAD.*** then select the file you want to download.
   - For on-chip and off-chip NOR as well as memory-mapped serial flash programming, refer to the **FLASH** command group and to the **"Onchip/NOR FLASH Programming User's Guide"** (norflash.pdf).

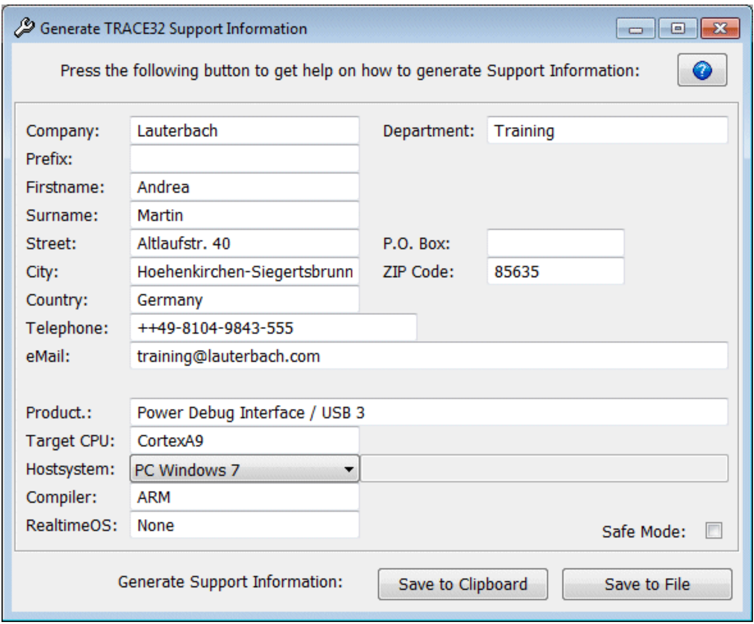
     A video tutorial about programming the processor internal FLASH in TRACE32 is available here:
     support.lauterbach.com/kb/articles/flash-programming
   - For non-memory-mapped flash programming (NAND, SPI, eMMC) refer to the **FLASHFILE** command group. Depending on your flash device, you can find more details in **"NAND FLASH Programming User's Guide"** (nandflash.pdf), **"Serial FLASH Programming User's Guide"** (serialflash.pdf) or **"eMMC FLASH Programming User's Guide"** (emmcflash.pdf).

The different debug scenarios are described in details in **"Establish Your Debug Session"** (tutor_setup.pdf).

How to Generate TRACE32 Support Information

If you need assistance in setting up the debugging environment, be sure to include detailed system information.

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



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

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

Start-Up Scripts

It is strongly recommended to summarize the commands, that are used to set up the debug environment, in a start-up script. The script language PRACTICE is provided for this purpose.

The standard extension for a script file is `.cmm`.

Write a Start-Up Script

The debugger provides an ASCII editor, that allows to write, to run and to debug a start-up script.

PEDIT *<file>*	Open *<file>* with the script editor

```
PEDIT my_startup
```

The debugger provides two commands, that allow you to convert debugger configuration information to a script.

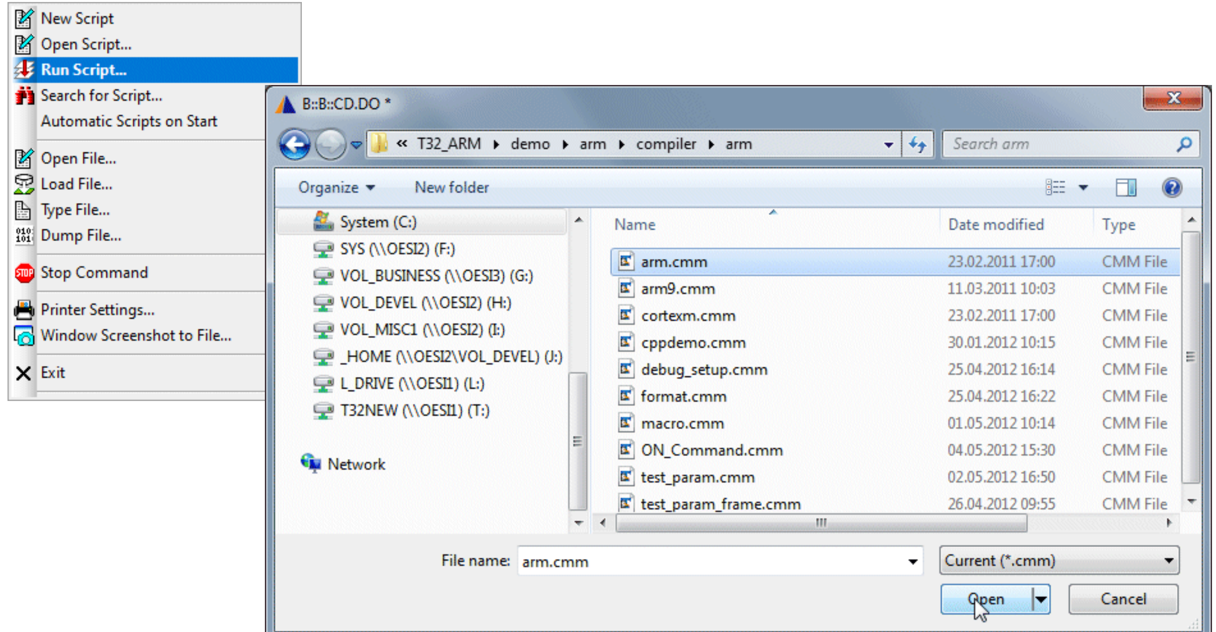
STOre <file> [*<item>*]	Generate a script that allows to reproduce the current settings
ClipSTOre [*<item>*]	Generate a command list in the clip-text that allows to reproduce the current settings

```
STOre system_settings SYStem          ; Generate a script that allows you
                                      ; to reproduce the settings of the
                                      ; SYStem window at any time

PEDIT system_settings                 ; Open the file system_settings
```

```
ClipSTOre SYStem                      ; Generate a command list that
                                      ; allows you to reproduce the
                                      ; settings of the SYStem window
                                      ; at any time
                                      ; The generated command list can be
                                      ; pasted in any editor
```

Run a Start-up Script



ChDir.DO *<file>*	Change directory and run script

```
ChDir.DO my_startup.cmm
```

Automated Start-up Scripts

There are two ways to define a start-up script, that is automatically started, when the debugger is started.

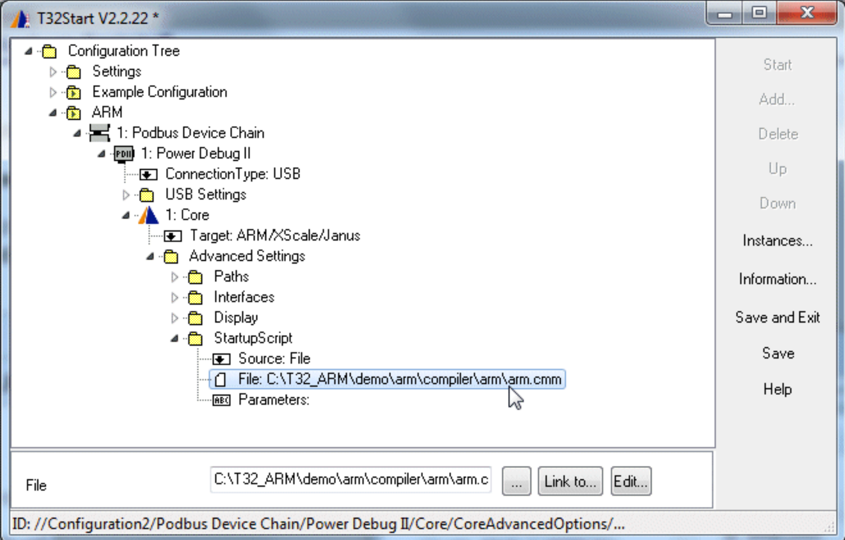
1. Define start-up script in conjunction with the executable

The debugger-executable can be started with the start-up script as parameters.

```
c:\t32\t32arm.exe -s g:\and\arm\start.cmm
```

2. Use T32Start to define an automated start-up script

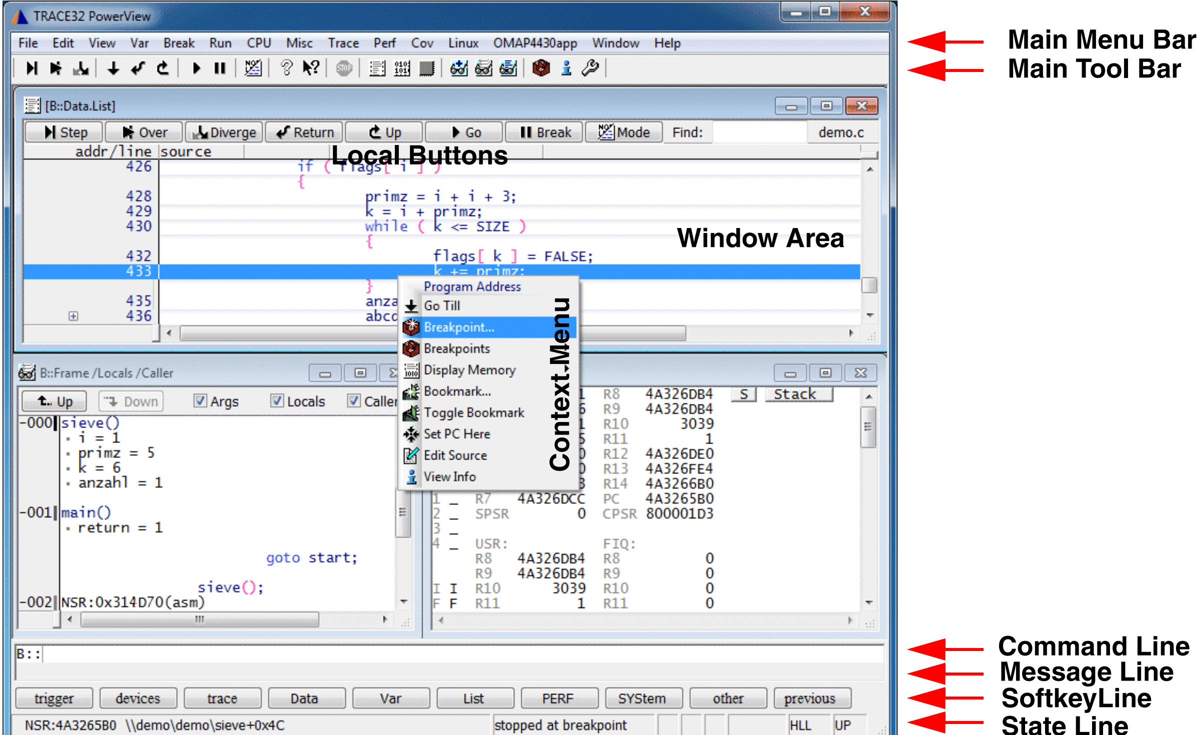
If you use T32Start to start the debugger, an automated start-up script can be defined.



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.



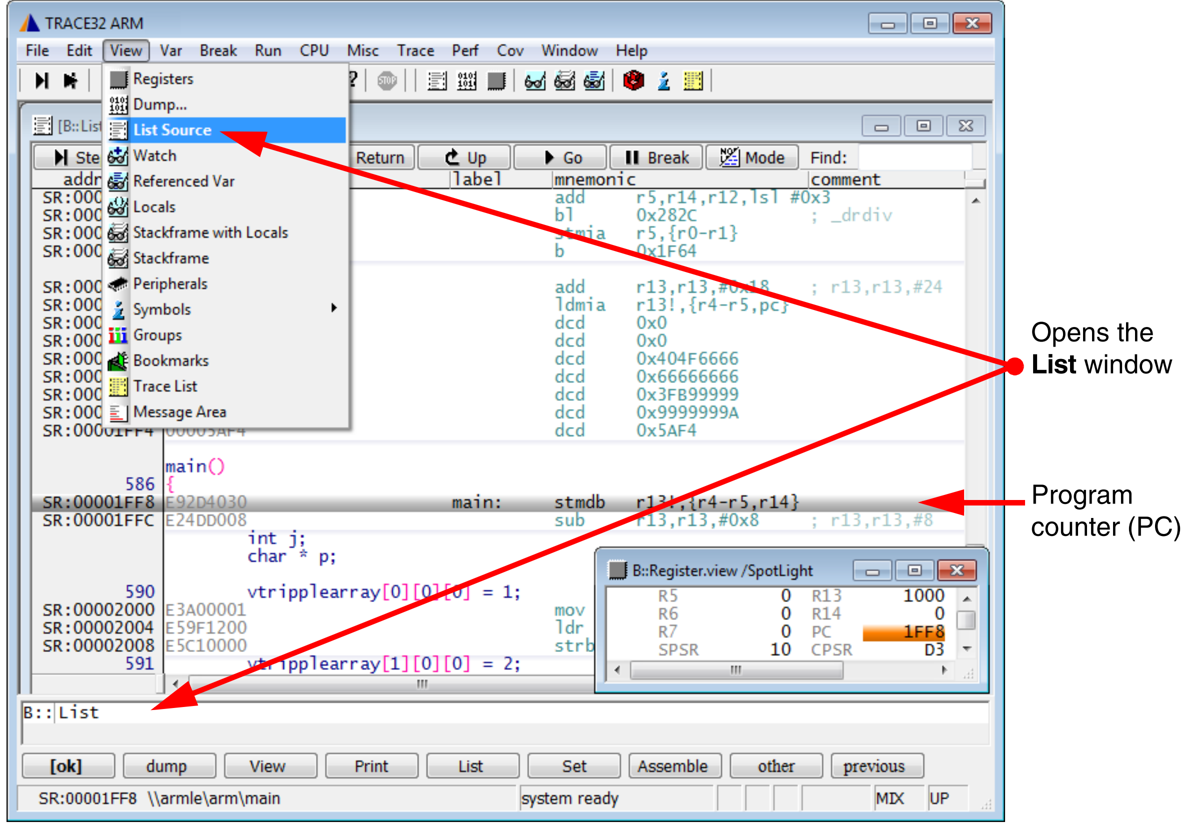
We'll briefly explain the GUI using the **List** command and **List** window as an example.

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`

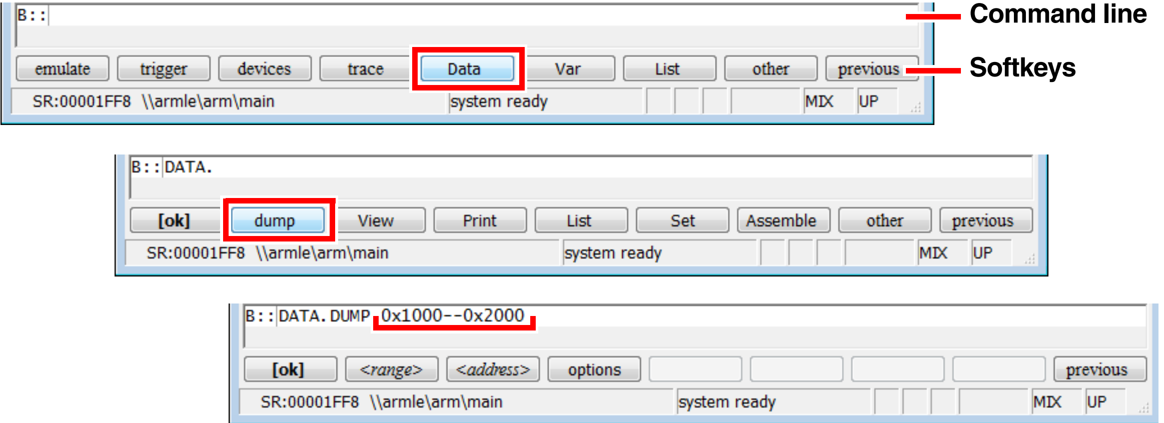
UPPER CASE letters indicate the short forms of commands and must not be omitted. All lower case letters can be omitted. This makes short forms an efficient time saver when you are entering frequently-used commands in the command line. Examples:

- 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.

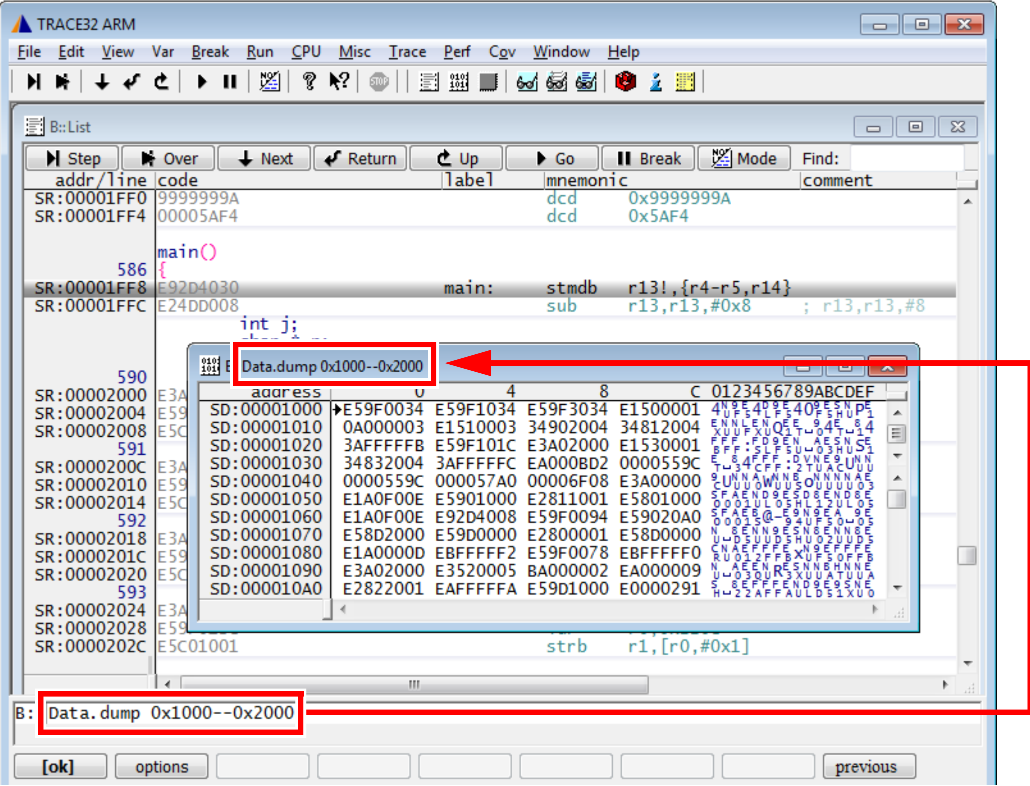
Example - To assemble the Data.dump command using the softkeys:

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.

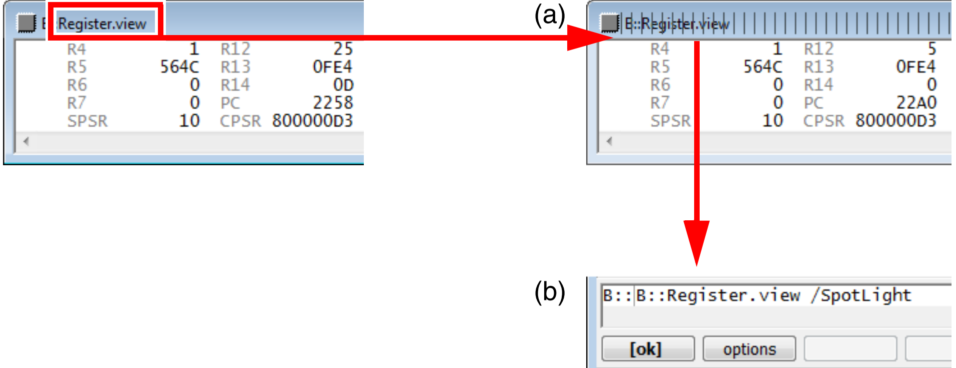


You can **re-**insert a command from a window caption (a) into the command line (b) in order to modify the command. Let's do this with the **Register** window.

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.

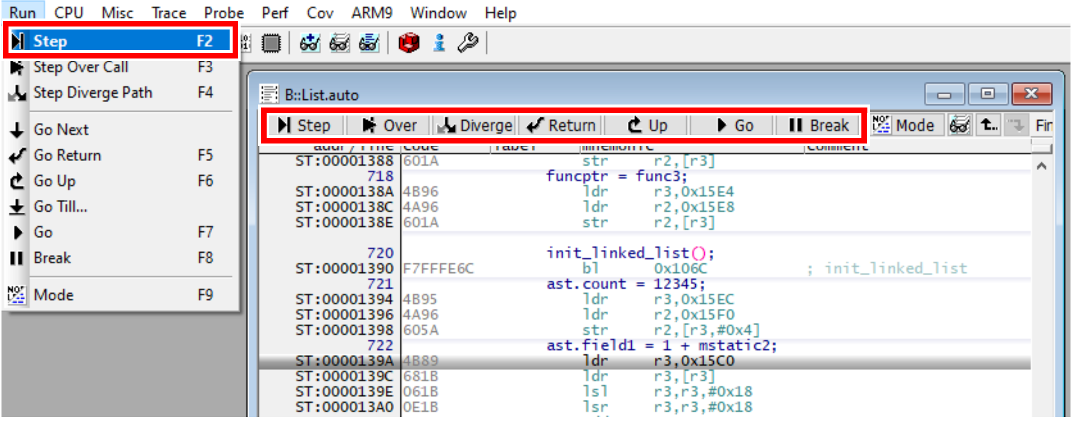
5. Click **Single Step** on the TRACE32 toolbar. Changed registers are highlighted immediately.

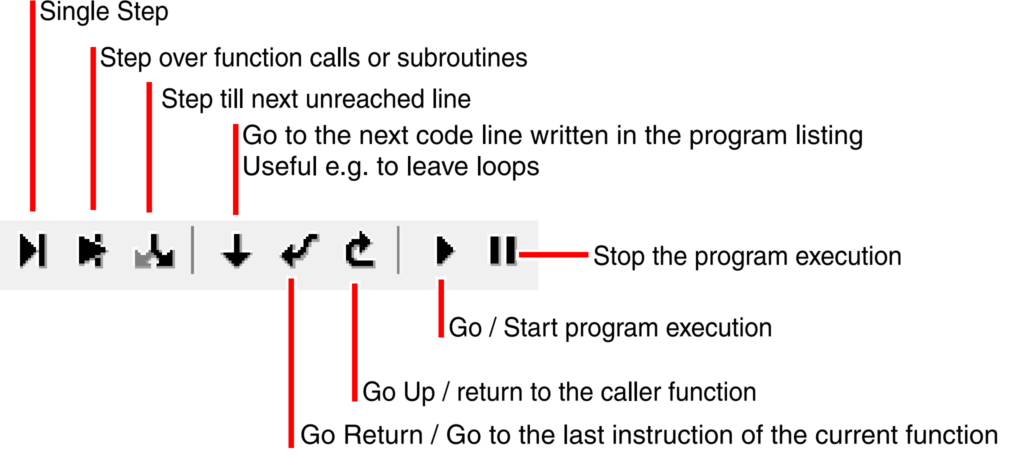
Debugging the Program

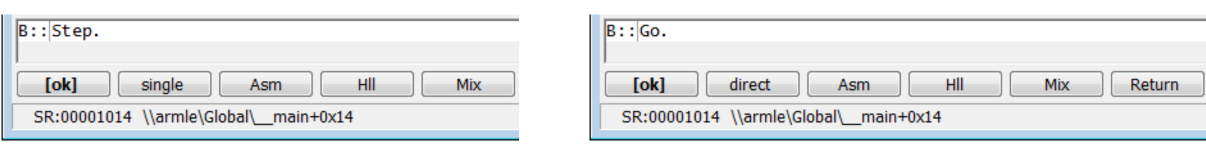
Basic Debug Commands

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.



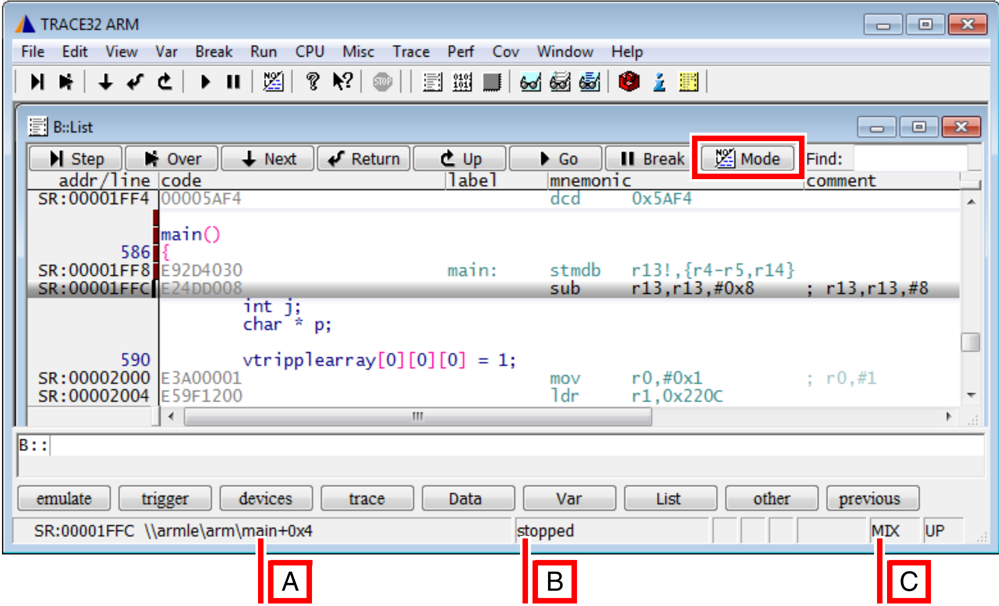




TRACE32 provides also more complex debug control commands. You can step until an expression changes or becomes true.

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.

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



The state line tells you:

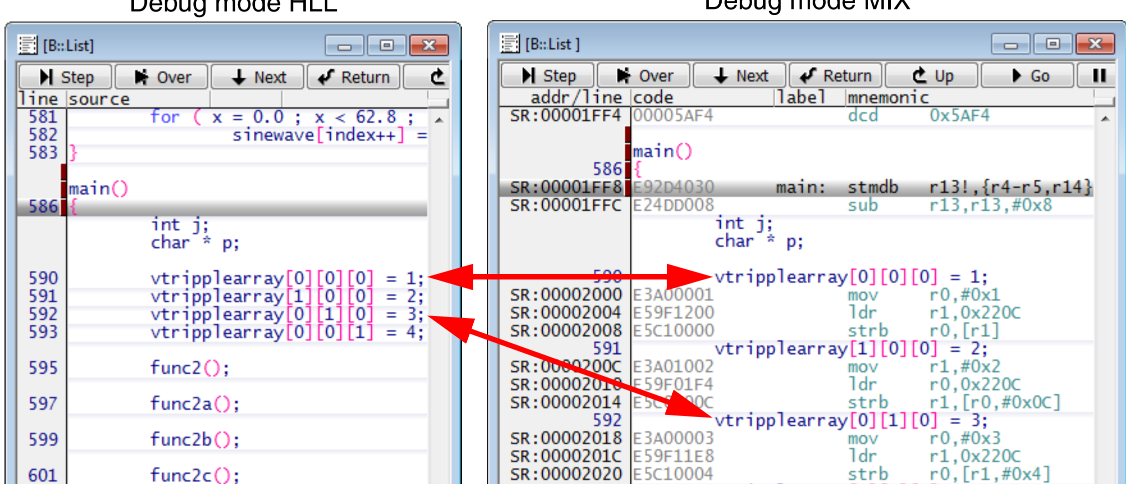
A The (symbolic) address of the current cursor position.
The program counter (PC) is highlighted in gray.

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 **MIX**ed mode with HLL and its corresponding assembler mnemonic.

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

Debug mode HLL Debug mode MIX

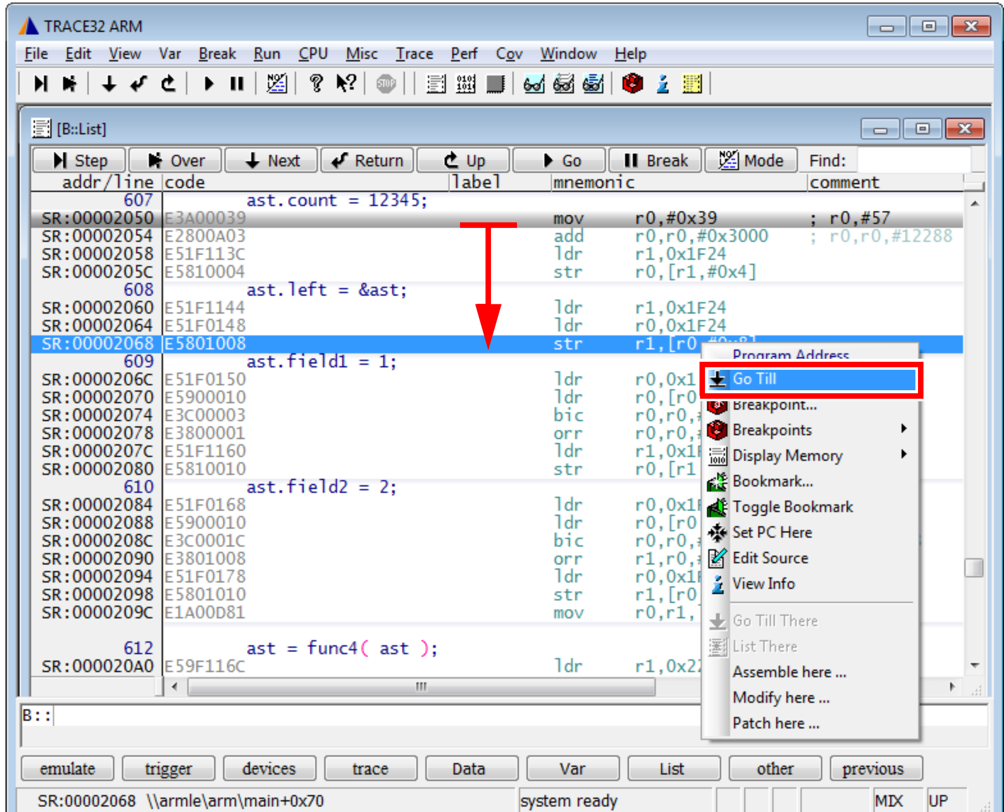


7. Click Step **Step**.
The step you are taking is debug mode HLL goes to the next source code line.

8. Click Mode **Mode** again to toggle the debug mode to **MIX**.

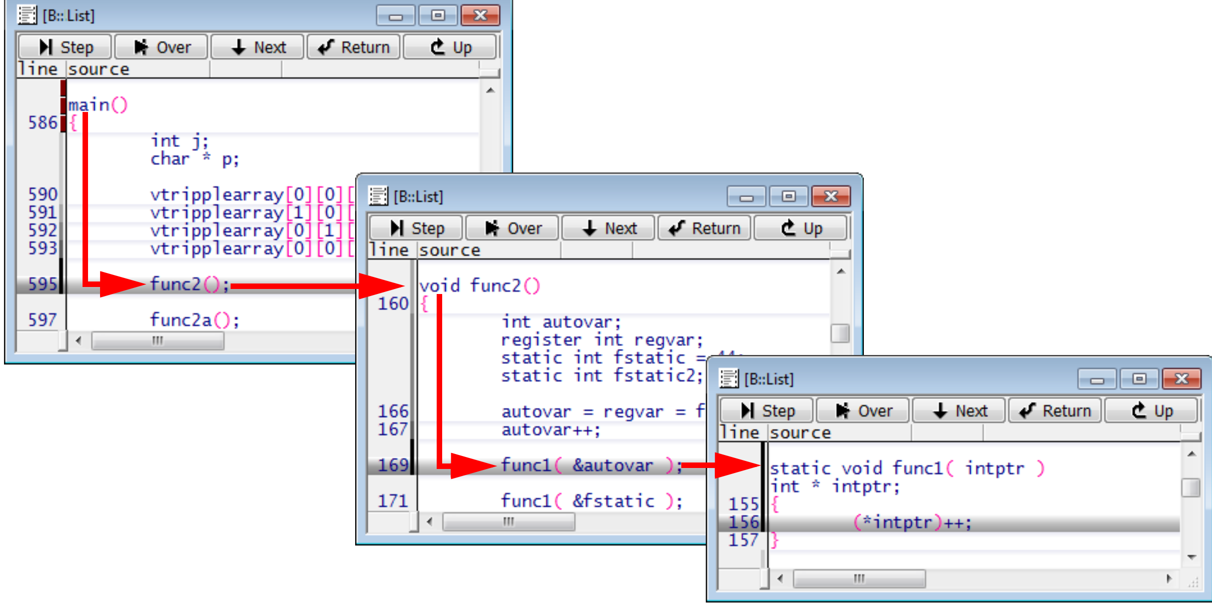
9. Click Step **Step**.
This time, the step executes one assembler line.

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

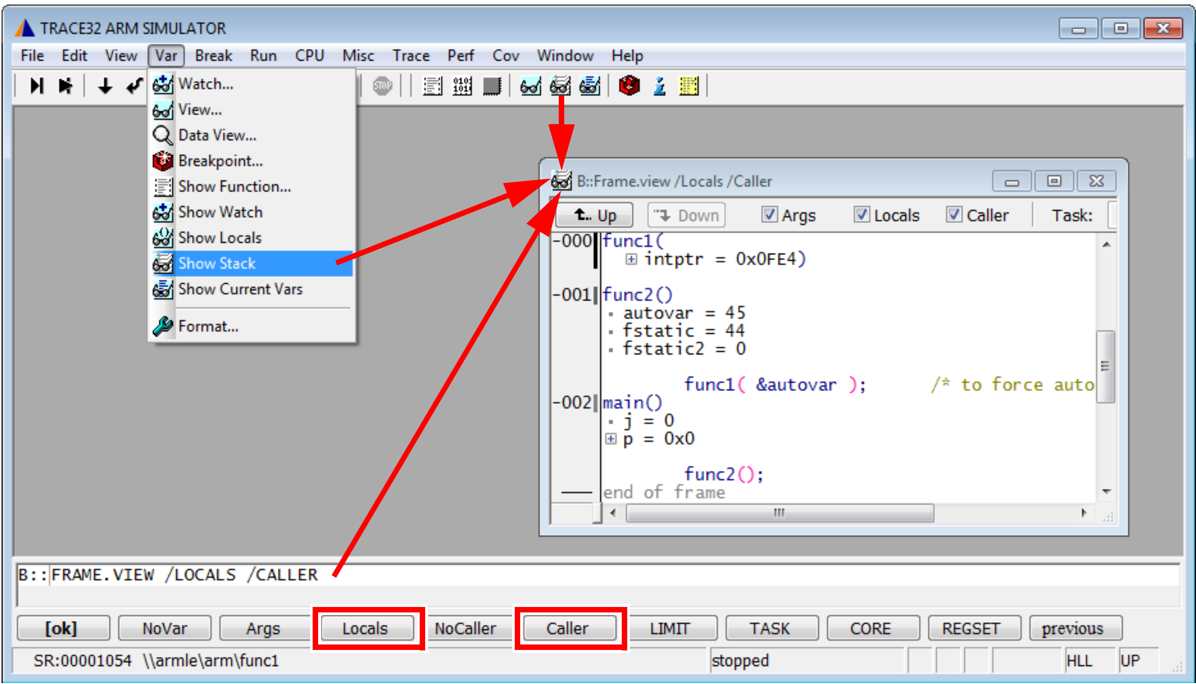
For the following example, let's assume we have the following call hierarchy: **`main()`** calls **`func2()`** and **`func2()`** calls **`func1()`**:



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

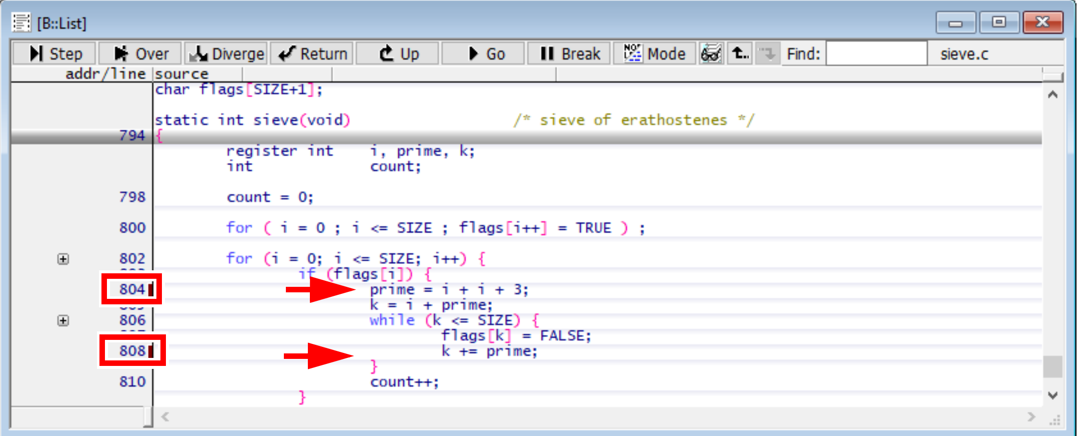
Setting Breakpoints

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.

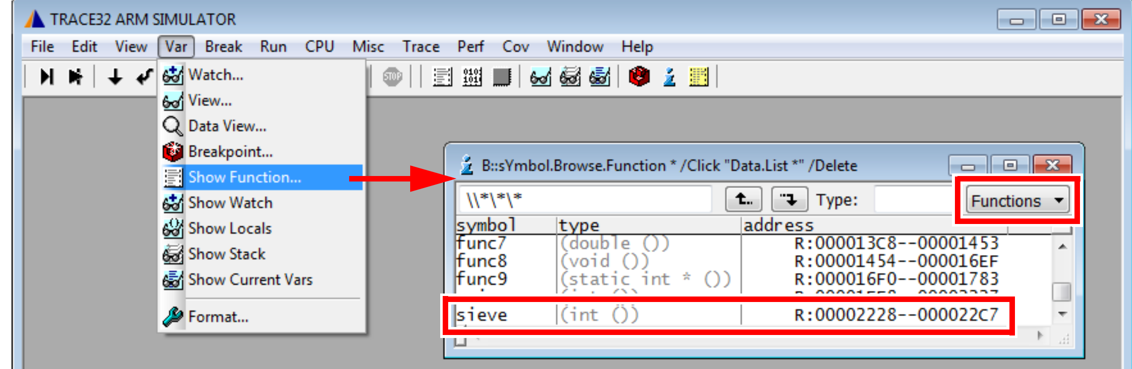
2. Make sure to click the white space in the code line, and not the code literal.

   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

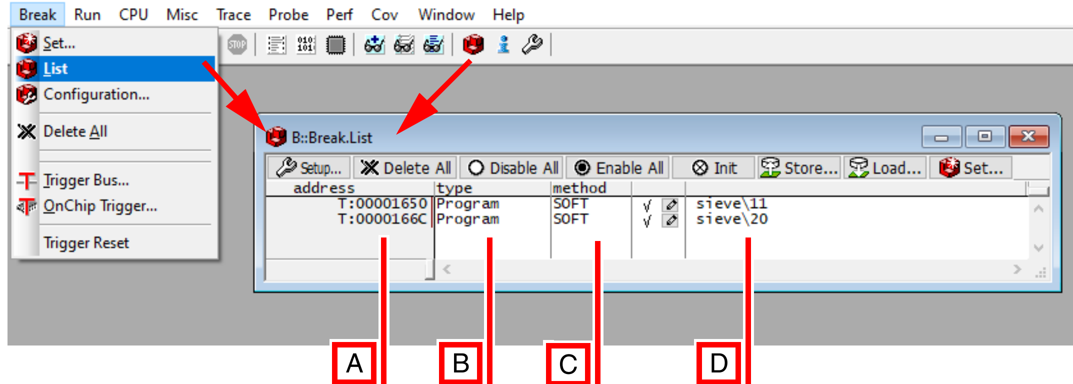
1. Choose **Var** menu > **Show Function**.
   The **sYmbol.Browse.Function** window opens.



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.

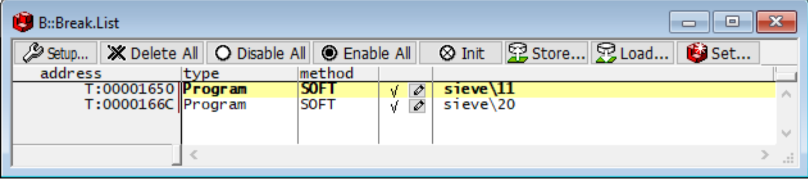
B Breakpoint type.

C Breakpoint method: SOFTware, ONCHIP or DISABLED.

D Symbolic address of the breakpoint. Example:
- `sieve\11` means source code line 11 in function `sieve`.

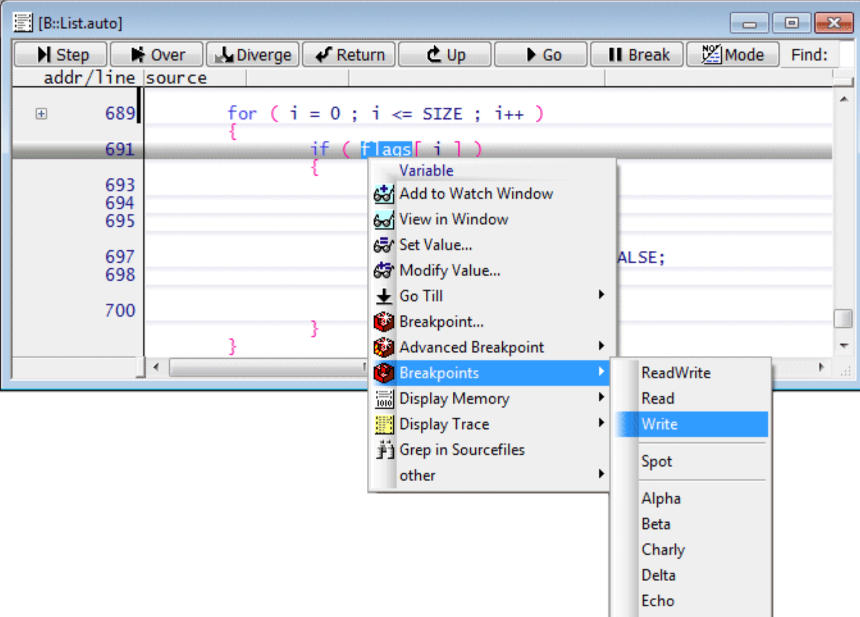
2. On the toolbar, click ▶ **Go** to start the program execution.

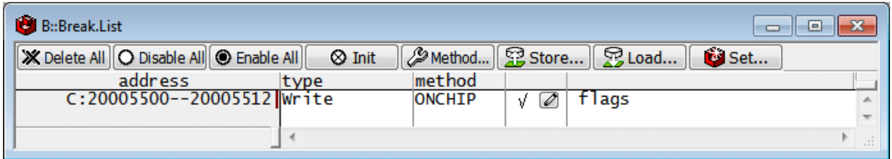
3. When the program execution stops at a breakpoint, it is highlighted in the **Break.List** window.



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**.





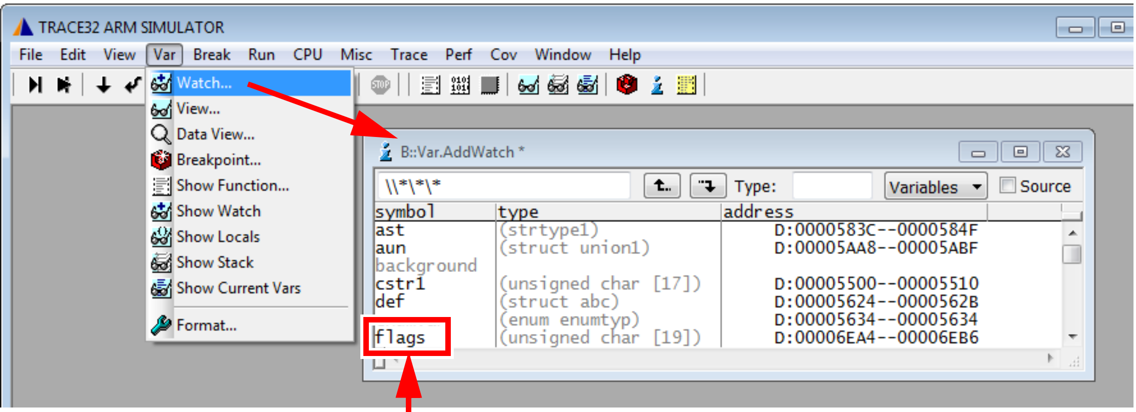
Variables

Video tutorials about variable display in TRACE32 are available here:
support.lauterbach.com/kb/articles/variable-logging-and-monitoring-in-trace32

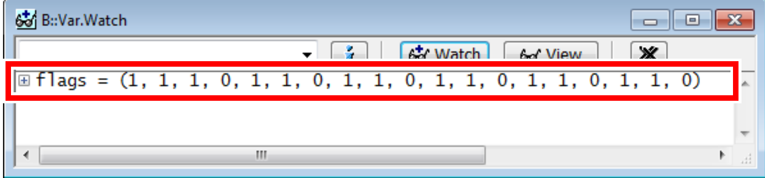
Displaying Variables

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.

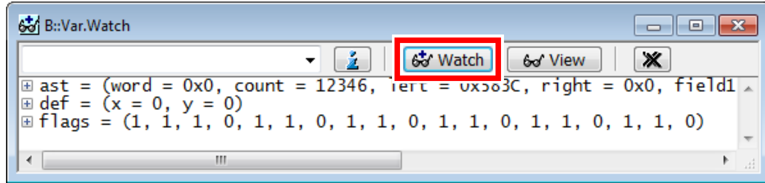


2. Double-click the variable `flags`.
   The **Var.Watch** window opens, displaying the selected variable.



3. **Alternative steps:**

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



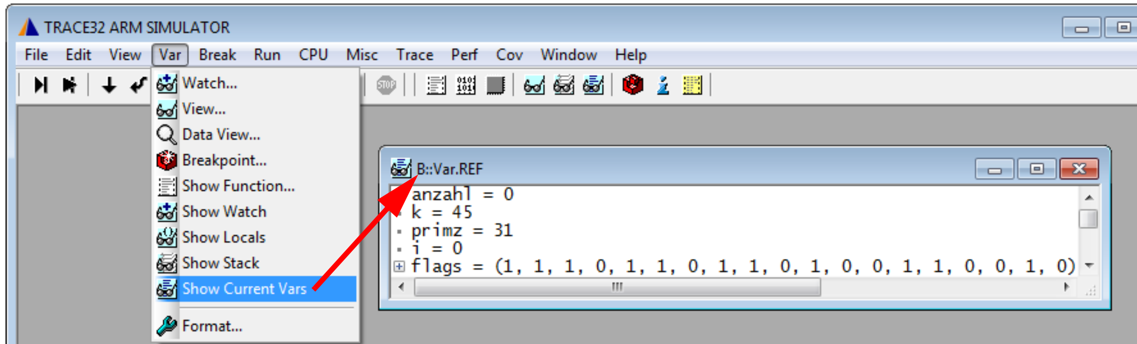
   - From a **List** window, drag and drop any variable you want into the **Var.Watch** window.
   - In a **List** window, right-click any variable, and then select **Add to Watch window** from the context menu.
   - If you want to display a more complex structure or an array in a separate window, choose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



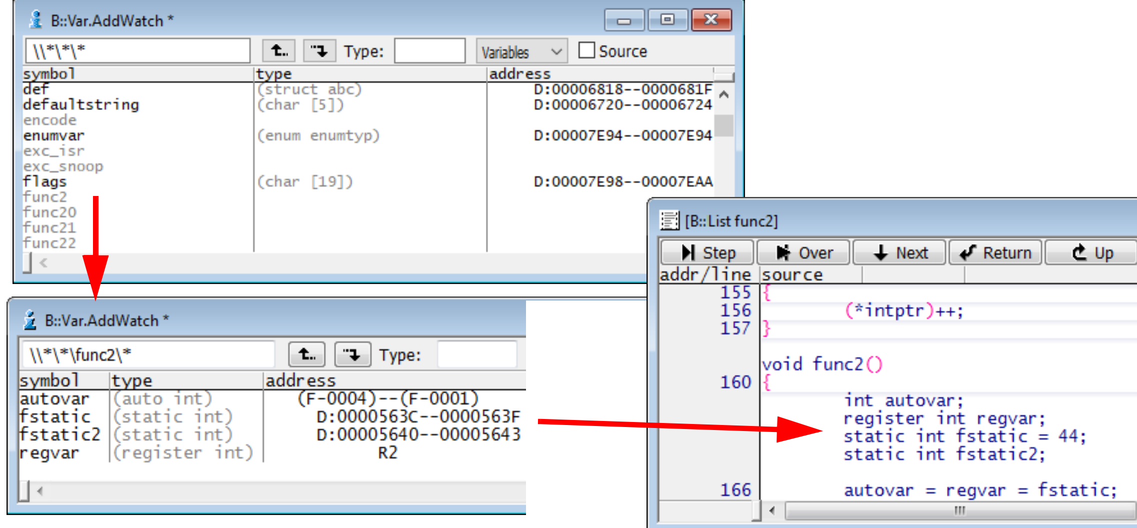
3. Click **Step** on the TRACE32 toolbar to execute a few single steps.
   The **Var.REF** window is updated automatically.

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*.

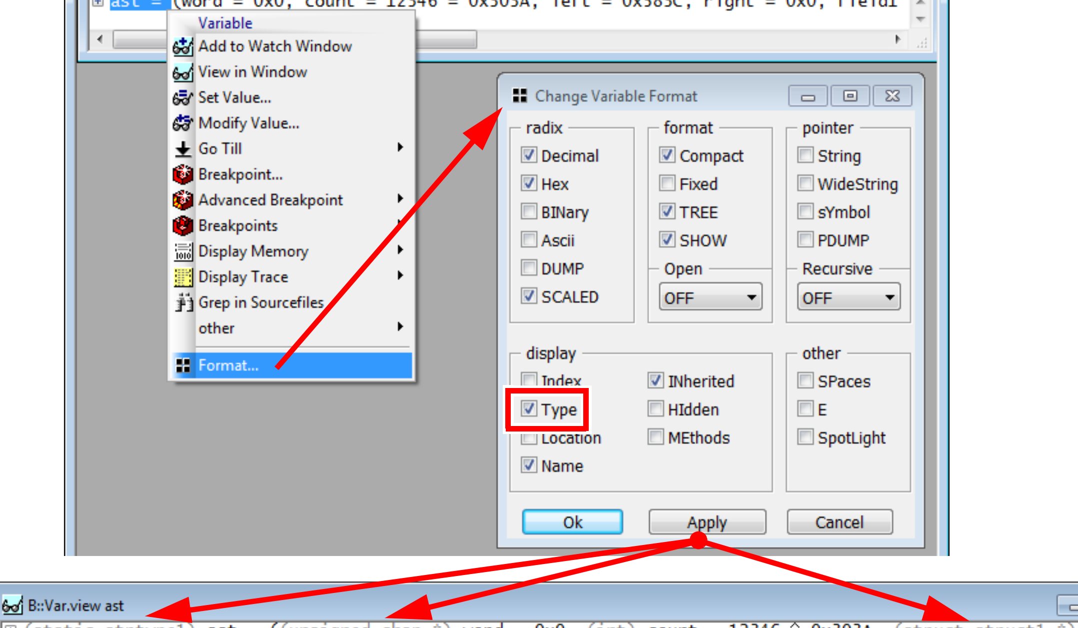
To format the display of an individual variable:

1. At the command line, type: `Var.view ast` (The variable `ast` is included in this demo.)

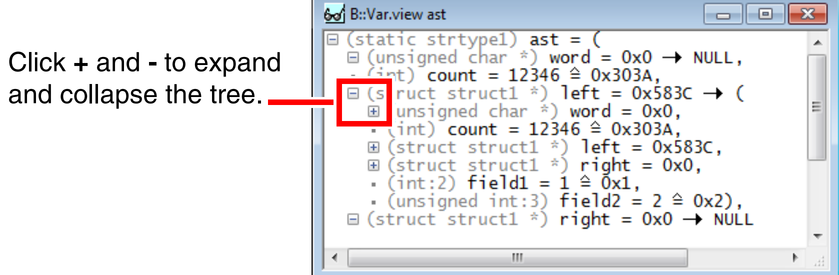
2. In the **Var.view** window, right-click `ast`, and then click **Format**.
   The **Change Variable Format** dialog opens.

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.



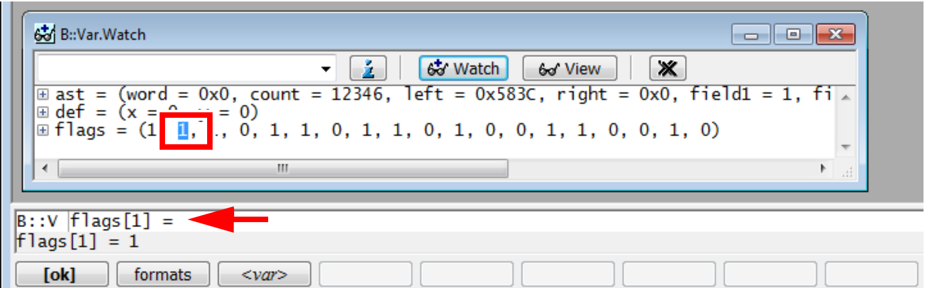
5. For more complex variable select **TREE** in the **Change Variable Format** dialog box.



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 **v**



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

Displaying Memory

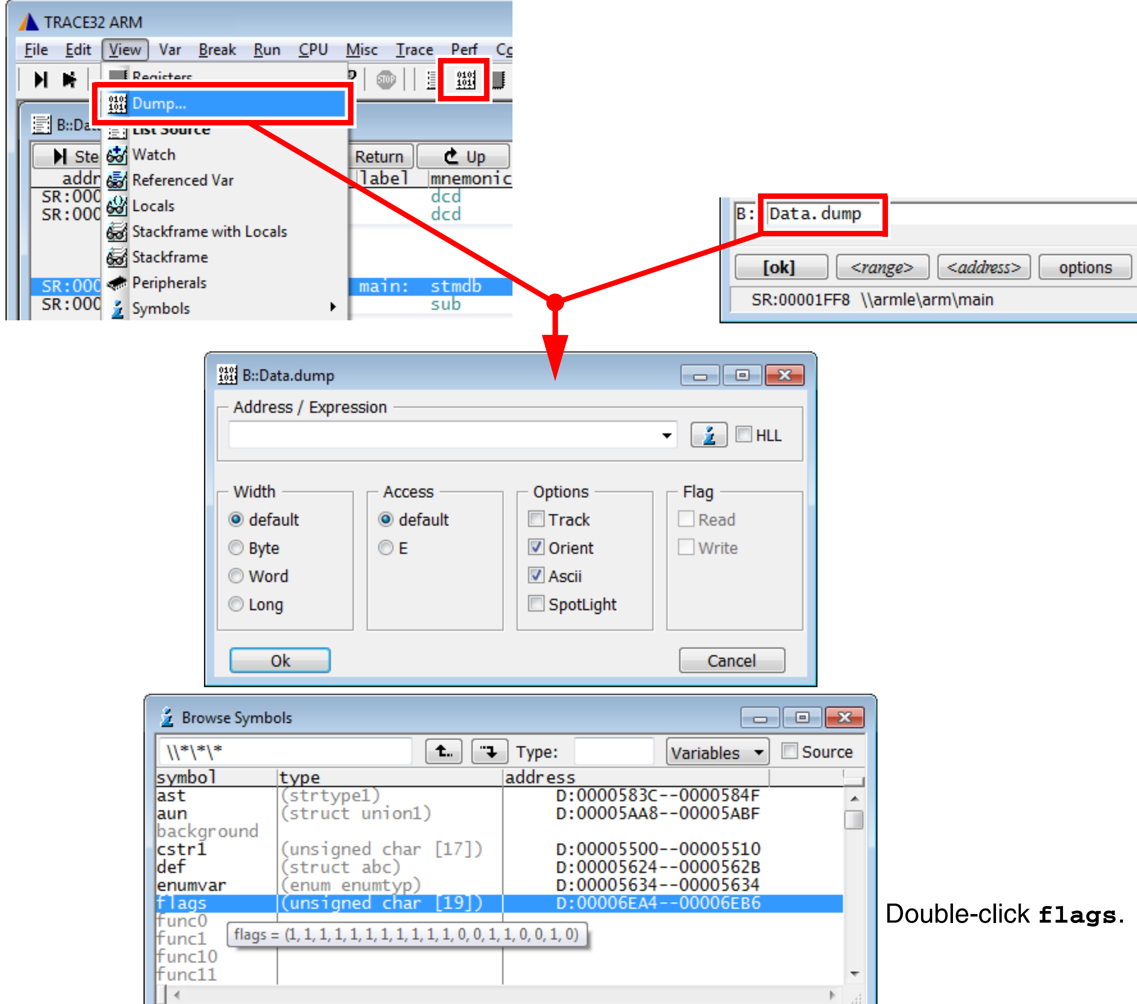
1. To display a memory dump in a **Data.dump** window, do one of the following:

   - 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`

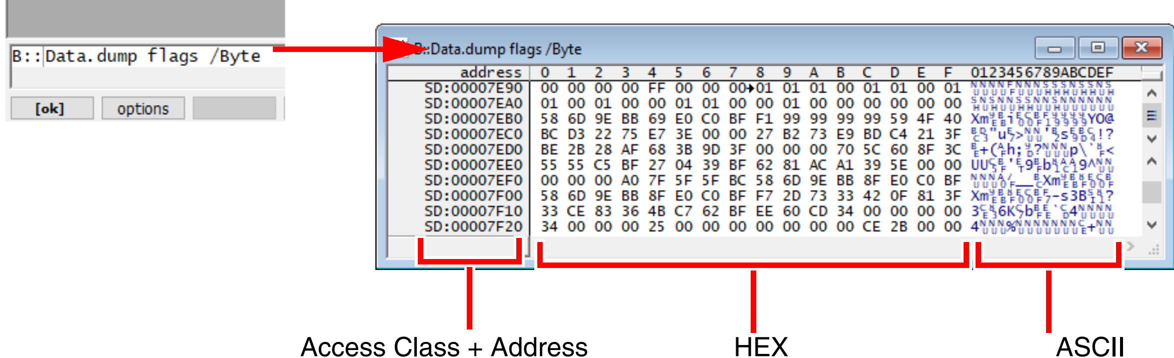
   - Alternatively click to browse through the symbol database.

3. In the **Browse Symbols** window, double-click the symbol `flags` to select it, and then click **OK**.



Double-click `flags`.

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)

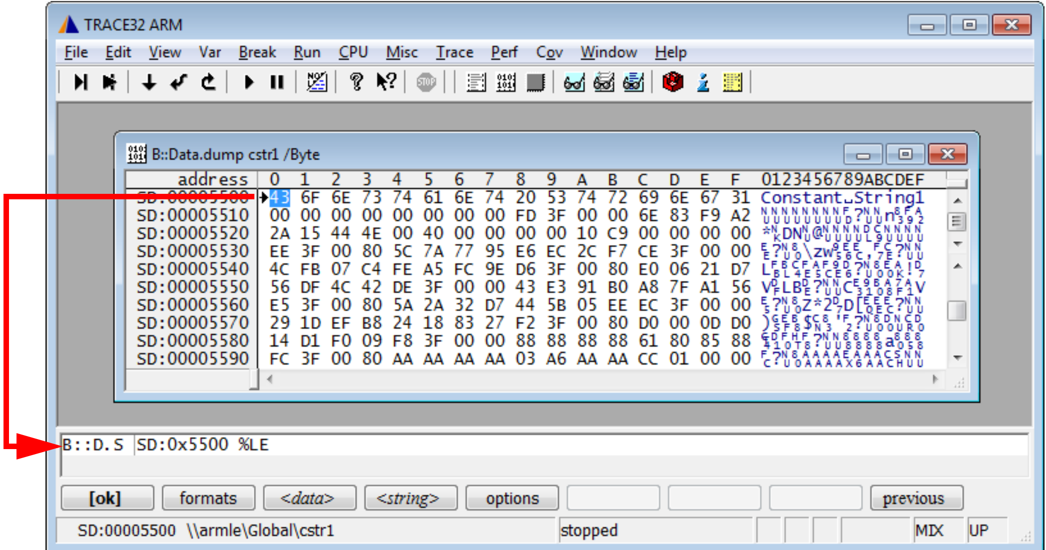
```
Data.dump SD:0x5530--SD:0x554F
```

- *<start_address>++<offset>*

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

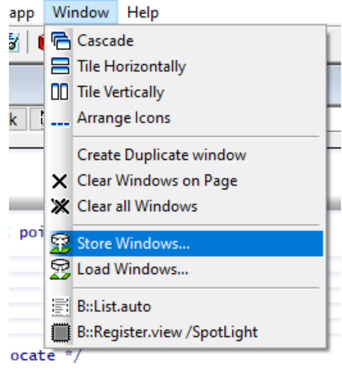
Modifying Memory

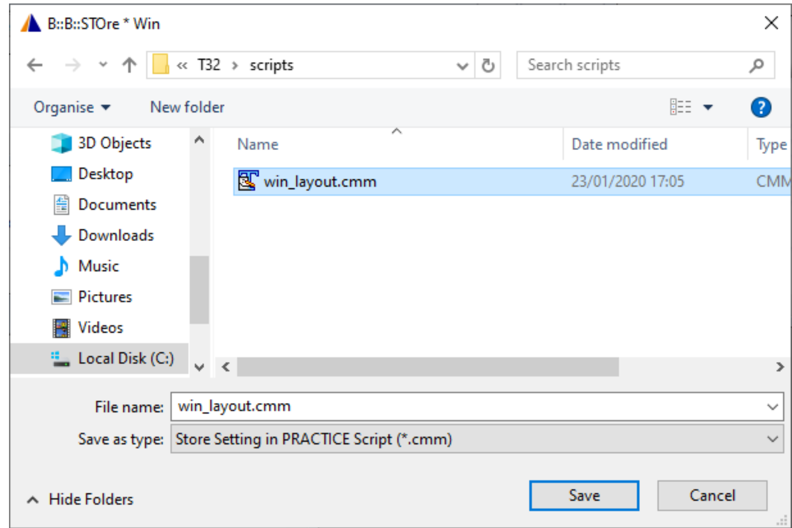
1. In a **Data.dump** window, double-click the value you want to modify.
   A **Data.Set** command for the selected address is displayed in the command line. The short form of the command is **`D.S`** or **`d.s`**



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

To save the window configuration for you next debug session use ***Store Windows…*** from the ***Window*** menu. ***Store Windows…*** generates a PRACTICE file, that includes all commands to reactivate your complete window configuration automatically.





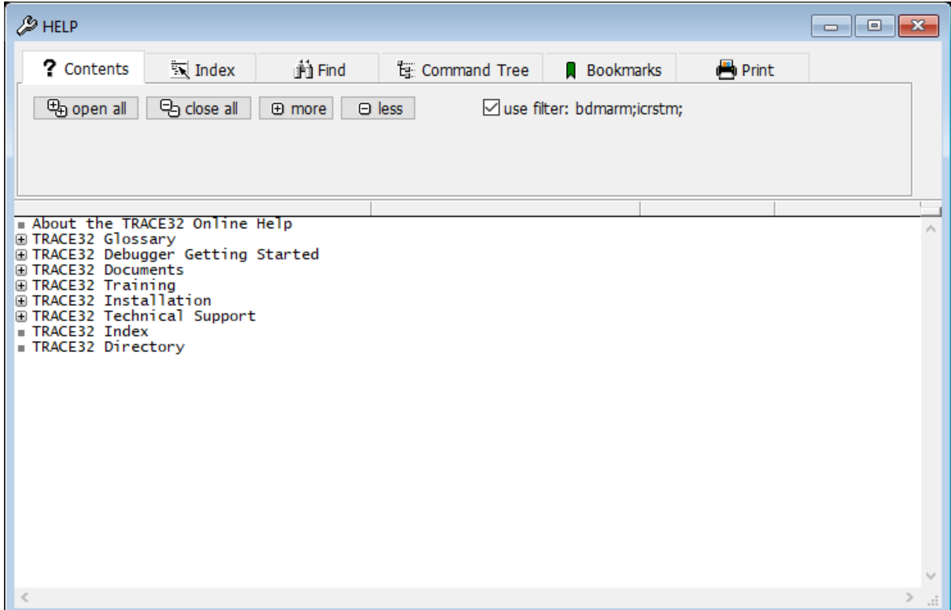
The saved window layout can be loaded again for the next debug session with the ***Load Windows…*** in the ***Window*** menu.

You can also add a call into your start-up file:

```
DO win_layout.cmm
```

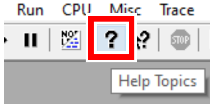
Getting Online Help

The online help system consists of several documents. They are accessible as PDF-files directly from the TRACE32 software and can be found in the **pdf** directory.

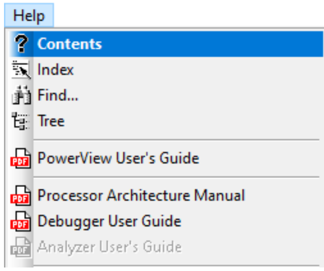


There are different ways to open the TRACE32 online help:

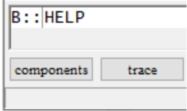
- **Help Topics** button on the toolbar



- **Help** menu > **Contents**

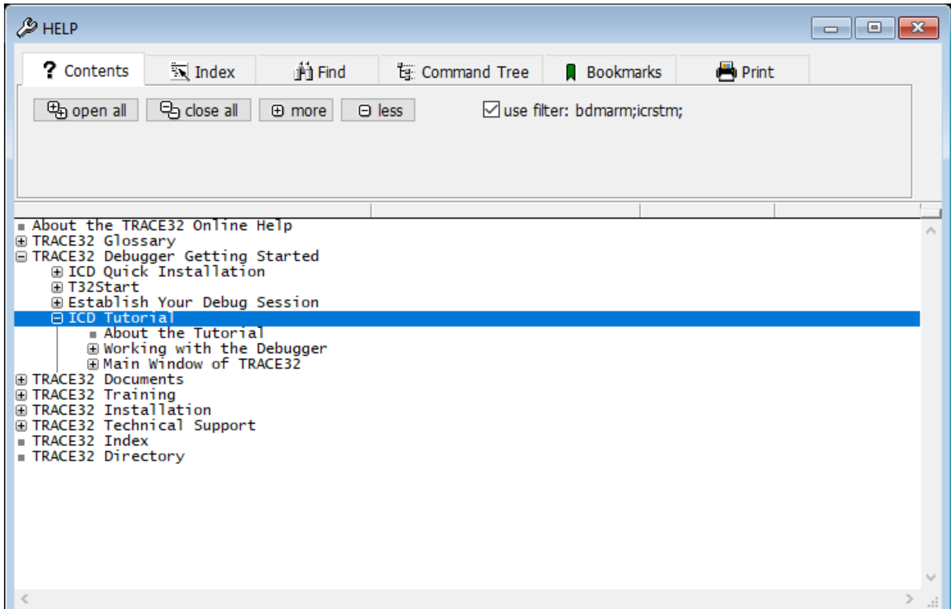


- **HELP** command in the command line



- **Help** button in the **Welcome to TRACE32!** dialog.

The help system is organized in a multilevel structured way. The screen below shows how to find this tutorial.



It is also possible to help for a single command. Enter the command into the command line, add a space and push F1.