LAUTERBACH
DEVELOPMENT TOOLS

Debugger Tutorial

MANUAL

Debugger Tutorial

Version 06-Jun-2024

History

05-Jun-2024 Revised manual

About the Tutorial

This tutorial guides you through the necessary steps to configure and start a debug session using a TRACE32 hardware-assisted debugger. It also demonstrates basic debug functionality using the TRACE32 PowerView interface, helping you become familiar with the fundamental features of TRACE32

This is an entry-level document intended for users with little or no prior experience with TRACE32 debug tools. For a detailed overview of all debug features, refer to **“Training Basic Debugging”** (training_debugger.pdf).

To follow this tutorial, a basic understanding of software debugging and the C-programming language is helpful, as it will allow you to follow the example code provided. Additionally, a basic knowledge of the target processor and the assembler/compiler used is necessary to get your debug system running.

This tutorial assumes that the TRACE32 debugger software is already installed. You can do so install the TRACE32 software from the DVD included with the tools. Alternatively, you can download the TRACE32 software installation package can also be downloaded from the Lauterbach website. The Installation process is outlined in **“Software Installation”** in TRACE32 Installation Guide, page 20 (installation.pdf).

Connect the TRACE32 Hardware

To proceed with this tutorial, a functioning target platform is needed, such as a development board.

Follow then these steps:

- If the debug probe is already connected to the target, disconnect it while the target power is off.
- Connect the TRACE32 hardware according to the instructions provided in **"Tool Configuration"** in TRACE32 Installation Guide, page 9 (installation.pdf).
- Power on the TRACE32 hardware
- Proceed with the next chapter of this tutorial

Start TRACE32 PowerView

The TRACE32 executables are named ***t32m<architecture>*[*.exe*]** and are located in the **bin\<os>** directory within TRACE32 installation directory
(e.g. **bin\windows64** or **bin/pc_linux64**).

These executables launch the TRACE32 graphical user interface, known as **TRACE32 PowerView**. Each executable is specific to a target processor architecture; for example, **t32marm** starts TRACE32 PowerView for Arm, while **t32mtc** starts it for TriCore. The executables available in the **bin\<os>** folder are determined by the target processor architectures selected during the installation process.

You can start TRACE32 PowerView by either double-clicking the ***t32m<architecture>*** executable or running it from the command line without any additional parameters.

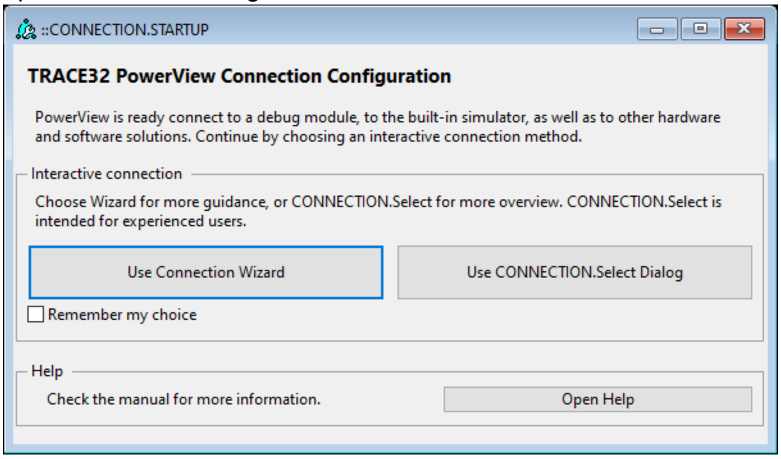
Example for Windows:

```
C:\T32\bin\windows64\t32marm.exe
```

Example for Linux:

```
/opt/t32/bin/pc_linux64/t32marm
```

TRACE32 PowerView will initiate an “interactive connection mode”, enabling users to select the desired operation mode through user interface:



::CONNECTION.STARTUP

TRACE32 PowerView Connection Configuration

PowerView is ready connect to a debug module, to the built-in simulator, as well as to other hardware and software solutions. Continue by choosing an interactive connection method.

Interactive connection

Choose Wizard for more guidance, or CONNECTION.Select for more overview. CONNECTION.Select is intended for experienced users.

Use Connection Wizard | Use CONNECTION.Select Dialog

☐ Remember my choice

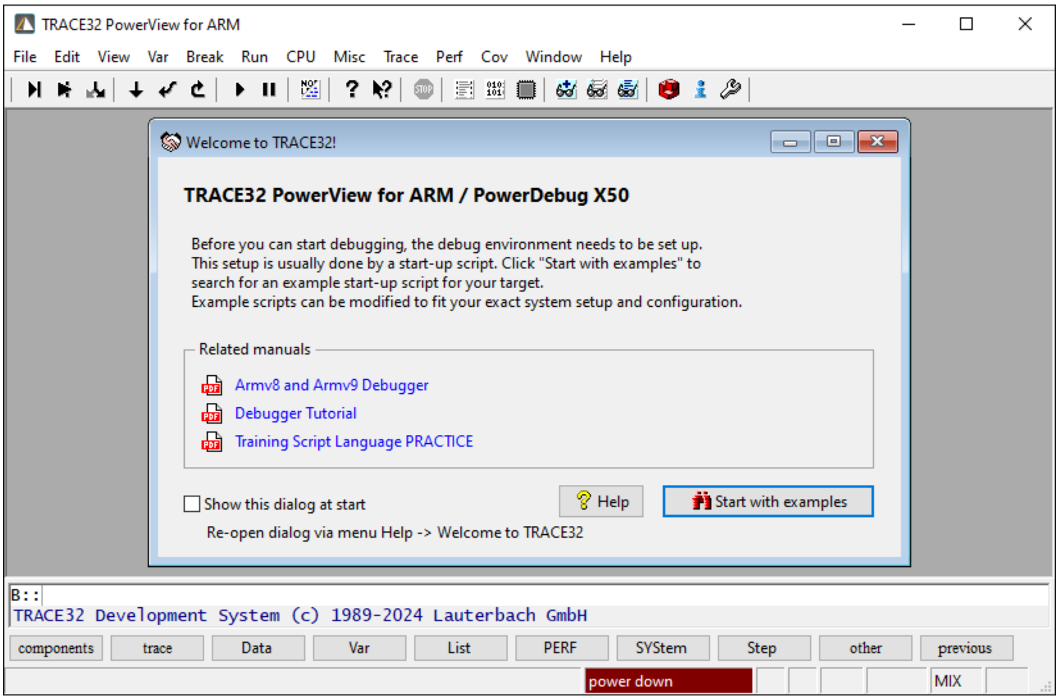
Help

Check the manual for more information. | Open Help

Please follow the “Connection Widzard” and enure that the TRACE32 debugger module being used is powered on and connected to the host PC via USB or Ethernet.

NOTE:	Before TRACE32 Release R.2024.09, a configuration file was by default used to define the TRACE32 operation mode and various other TRACE32 settings. However, this tutorial will not delve into the configuration file. For further details, please refer to the **“TRACE32 Installation Guide”** (installation.pdf).

If the connection to the TRACE32 debugger hardware is successfully established, the TRACE32 PowerView interface starts. By default, a **“Welcome to TRACE32!”** dialog appears. This dialog displays the target architecture and used TRACE32 debug module. It and includes links to key manuals.



TRACE32 PowerView for ARM
File Edit View Var Break Run CPU Misc Trace Perf Cov Window Help

Welcome to TRACE32!

TRACE32 PowerView for ARM / PowerDebug X50

Before you can start debugging, the debug environment needs to be set up.
This setup is usually done by a start-up script. Click "Start with examples" to search for an example start-up script for your target.
Example scripts can be modified to fit your exact system setup and configuration.

Related manuals

- Armv8 and Armv9 Debugger
- Debugger Tutorial
- Training Script Language PRACTICE

☐ Show this dialog at start | Help | Start with examples

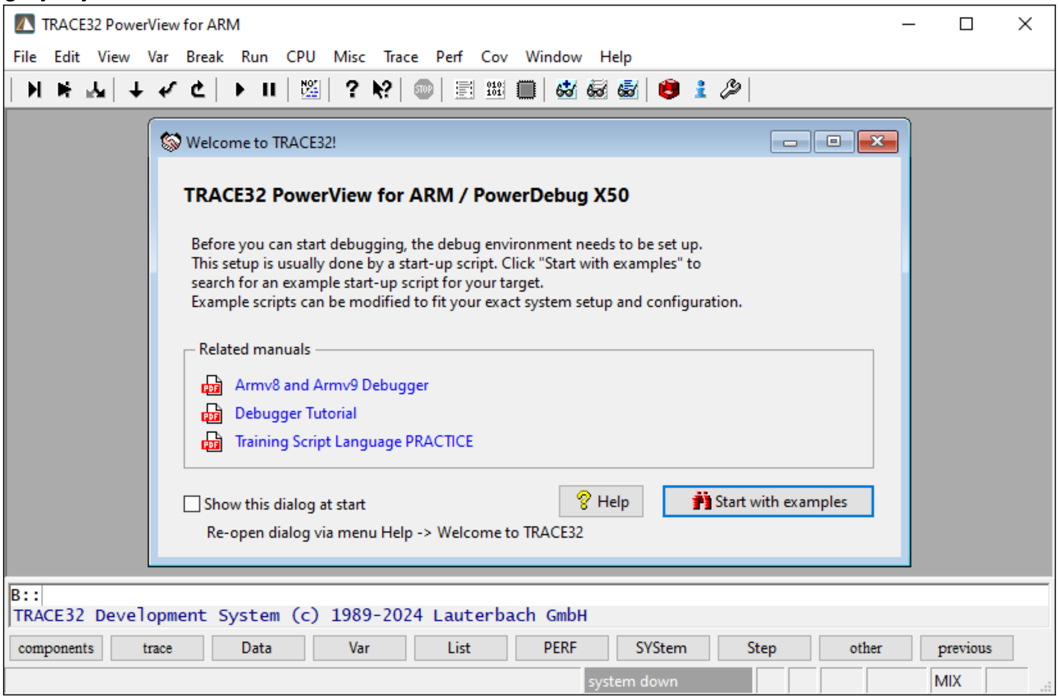
Re-open dialog via menu Help -> Welcome to TRACE32

```
B::
TRACE32 Development System (c) 1989-2024 Lauterbach GmbH
```

components trace Data Var List PERF SYStem Step other previous
power down MIX

The status line will display a red “power down” status, indicating that no reference voltage has been detected at the VCC pin of the debug probe. This is normal, since the debug probe is not yet connected to a powered target board.

Connect now the debug probe to your target platform then power on the target. The status line will indicate a grey “system down” status.



TRACE32 PowerView for ARM / PowerDebug X50

Before you can start debugging, the debug environment needs to be set up.
This setup is usually done by a start-up script. Click "Start with examples" to search for an example start-up script for your target.
Example scripts can be modified to fit your exact system setup and configuration.

- Armv8 and Armv9 Debugger
- Debugger Tutorial
- Training Script Language PRACTICE

Re-open dialog via menu Help -> Welcome to TRACE32

```
B::
TRACE32 Development System (c) 1989-2024 Lauterbach GmbH
```

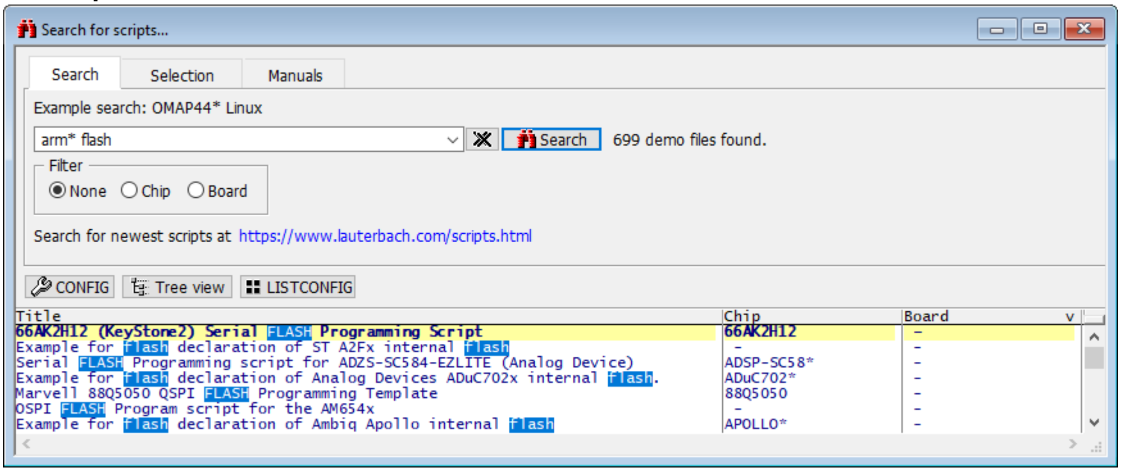
Set Up the Debug Environment

We can proceed now with the necessary steps to establish a debug connection with the target processor. The start-up procedure for the debug session depends significantly on the processor used.

NOTE:	For detailed information on CPU-specific settings, refer to the Processor Architecture Manual, which can be accessed by selecting **Help > Processor Architecture Manual** from the TRACE32 PowerView menu.

To simplify the start-up procedure for TRACE32 users, the demo directory within the TRACE32 installation folder includes a comprehensive collection of ready-to-run PRACTICE scripts. These scripts contain the necessary configurations to establish a debug connection with a wide variety of publicly available target chips and boards.

You can search for a suitable script for your target platform using the **"Search scripts..."** view, which can be accessed by clicking the **Start with examples** button in the **Welcome to TRACE32!** dialog. Additionally, you can open the **"Search scripts..."** view from the TRACE32 PowerView menu by selecting **"File" > "Seach for Script"**.



Example search: OMAP44* Linux

arm* flash

699 demo files found.

Filter [✓ None] Chip Board

Search for newest scripts at https://www.lauterbach.com/scripts.html

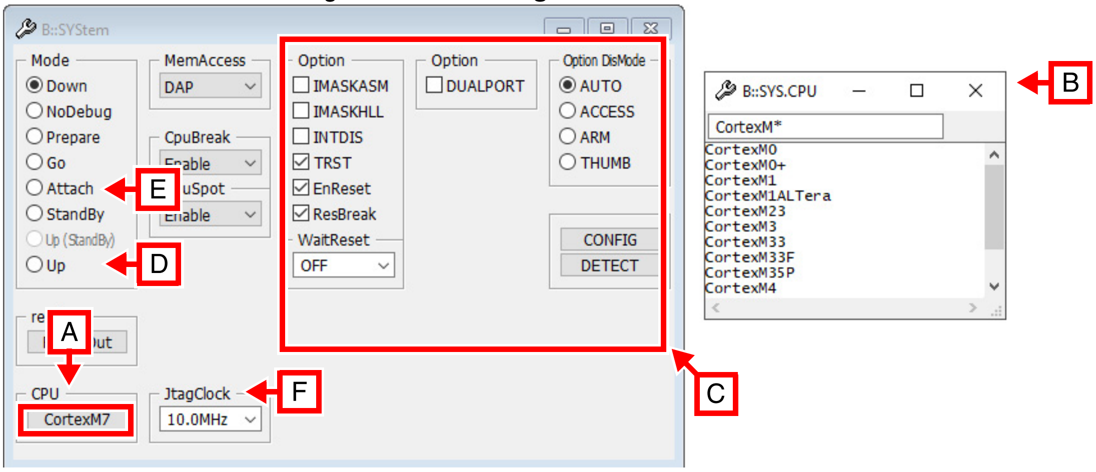
```
Title                                                          Chip          Board
66AK2H12 (KeyStone2) Serial FLASH Programming Script           66AK2H12      -
Example for flash declaration of ST A2Fx internal flash        -             -
Serial FLASH Programming script for ADZS-SC584-EZLITE (Analog Device)   ADSP-SC58*   -
Example for flash declaration of Analog Devices ADuC702x internal flash.  ADuC702*   -
Marvell 88Q5050 QSPI FLASH Programming Template                88Q5050       -
OSPI FLASH Program script for the AM654x                       -             -
Example for flash declaration of Ambiq Apollo internal flash   APOLLO*       -
```

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

A Typical Setup Procedure

This chapter outlines a typical setup procedure for the debugger. To demonstrate the necessary steps, we will start with a manual setup. Later, we will show you how to use PRACTICE scripts (*.cmm) to automate this process. For simplicity, we will use here a single-core system as an example.

The ***SYStem*** Window offers access to all CPU-specific settings. You can access this window by selecting the “**CPU**” and then choosing **“SYStem Settings...”**



CPU Selection

The initial step is to inform the TRACE32 debugger about the specific core/chip on your target **[A, B]**. A manual selection is not necessary if the target processor architecture supports automatic detection using the command **SYStem.DETECT CPU**. However, it's worth noting that this automatic detection is not always supported, such as in the case of Arm processors.

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

Examples:

```
SYStem.CPU CortexR5

SYStem.CPU CortexR5*                          ; Wild card symbols allowed
```

Adjust the JTAG Clock

The debugger uses a default JTAG clock of 10 MHz **[F]**. This frequency impacts the download speed. You may need to reduce the JTAG frequency if there are buffers, additional loads, high capacities on the JTAG lines, or if VTREF is very low.

SYStem.JtagClock *<frequency>*	Select the JTAG clock

Examples:

```
SYStem.JtagClock 1MHz               ; Reduce the JTAG clock to 1MHz on
                                    ; a slow target

SYStem.JtagClock 20MHz CTCK         ; Use compensation clock with
                                    ; an Arm debug cable
```

Establish Debug Communication

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

When ***Up*** is selected, the following steps are per default performed:

- The core is reset.
- Communication between the debugger and the core is initialized.
- The core is stopped at the reset vector, if supported by the core in use.

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

Another useful method to establish the communication between the debugger and the core is ***Attach*** **[E]**. ***Attach*** allows the debugger to connect to an already running core. The **Break.direct** command can then be used to stop the target processor's execution.

SYStem.Mode Attach	Establish the communication between the debugger and the target core (without reset)
Break.direct	Stop program execution

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

SYStem.Option Commands

Please note that some cores require additional settings using **SYStem.Option** commands, for example related to reset handling, before communication can be established. Most relevant options can be configured from the **SYStem** window. The available **SYStem.Option** commands depend on the target processor architecture. A description of all available options can be found in the **Processor Architecture Manual**.

In case you need assistance, please refer to the chapter **"Contacting the Lauterbach Support"**, page 40.

Additional Settings

In some cases, additional settings using Data.Set commands are necessary after establishing the debug connection. For example, you man need to disable a watchdog or initilize the target RAM.

Data.Set *<address>*|*<range>* [**%***<format>*] *<string>*	Modify memory-mapped configuration register/on-chip peripheral

Example: command sequence to disable a watchdog. The specific sequence will vary depending on the target processor used.

```
; disable the Watchdog
Data.Set AD:0x40011C00 %Long 0x1ACCE551
Data.Set AD:0x40011C00 %Long 0xE5331AAE
Data.Set AD:0x40011008 %Byte 0x0
```

For more information, please refer to the TRACE32 demo scripts and the documentation of your target processor.

Load Application

Setting up a debug environment involves loading the code to be debugged and the associated debug symbols. TRACE32 PowerView supports a wide range of compilers and compiler output formats. You can find a list of supported compilers on the Lauterbach website.

The most important commands for loading the code to be debugged and the associated debug symbols is **Data.LOAD**. Per default, the command loads the code/data from the specified file into the target memory and loads the symbol and debug information into TRACE32 PowerView. Using the option **/NoCODE** only the debug symbols are loaded.

Data.LOAD.*<sub_cmd> <file>* **/***<option>*	Load code and debug symbols
Data.LOAD.*<sub_cmd> <file>* **/NoCODE** **/***<option>*	Load only debug symbols

<sub_cmd> defines the file format, for example **Elf**. If you omit the format, TRACE32 PowerView tries to do an auto-detection.

The necessary steps for loading the application vary depending on whether the application is intended to run from RAM or flash memory.

Application Located in RAM

If the application is intended to run from RAM, you can directly use the **Data.LOAD** command. Just type **Data.LOAD.*** then select the file you want to download.

Examples:

```
Data.LOAD.Elf demo_sram.elf          ; Load code and debug symbols from
                                     ; Elf file

Data.LOAD.Elf *                      ; Load code and debug symbols from
                                     ; ELF file
                                     ; open file browser to select file
```

Application Located in Flash

TRACE32 supports programming on-chip and off-chip NOR flash memories, as well as serial flash memories, such as NAND, SPI and eMMC. However, For simplicity, this tutorial will focus on on-chip flash programming.

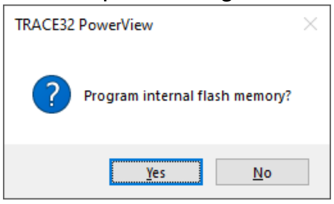
Ready-to-run scripts for most on-chip flash memories can be found in the TRACE32 installation under ~~/demo/*<architecture>*/flash/*<cpu>*.cmm

Examples:

```
~~/demo/arm/flash/stm32h7.cmm

~~/demo/tricore/flash/tc37x.cmm
```

To program your application to the on-chip flash memory of your processor/chip, follow these steps:

1. Call the flash programming script appropriate to your processor/chip.
2. The script will perform all necessary preparations then displays a pop-up asking the user to confirm proceeding with the flash programming.


TRACE32 PowerView

Program internal flash memory?

3. A file dialog is then opened, allowing you to browse the target application you want to program.

If the application is compiled with debug symbols they are automatically loaded into TRACE32 PowerView along with the flash programming.

The following framework can be used to call the flash programming script from your start-up script

```
CD.DO ~~/demo/tricore/flash/tc37x.cmm PREPAREONLY
FLASH.ReProgram ALL
Data.LOAD.Elf myapplication.elf
FLASH.ReProgram off
SYStem.Up
```

A video tutorial on programming the processor's internal flash memory using TRACE32 is available here:
support.lauterbach.com/kb/articles/flash-programming

Further Documents:

- For on-chip and off-chip NOR, as well as memory-mapped serial flash programming, refer to the **"Onchip/NOR FLASH Programming User's Guide"** (norflash.pdf).
- For non-memory-mapped flash programming, such as NAND, SPI and eMMC, refer to **"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).

Start-Up Scripts

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

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

Write a Start-Up Script

The debugger provides an PRACTICE script editor, that allows to write, to run and to debug a start-up script. The editor window provides syntax highlighting, configurable auto-indentation as well as multiple undo and redo.

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

```
PEDIT my_startup.cmm
```

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.cmm SYStem      ; Generate a script that allows you
                                      ; to reproduce the settings of the
                                      ; SYStem window at any time

PEDIT system_settings.cmm             ; Open the file system_settings.cmm
```

```
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

You can run a PRACTICE script from the TRACE32 PowerView interface by selecting the menu **“File” > “Run Script...”**. This action corresponds to using the TRACE32 command **DO** with the script name as parameter.

DO *<file>*	run PRACTICE script

Example:

```
DO my_startup.cmm
```

Alternatively, you can select the **“File” > “ChangeDir and Run Script...”**. The difference here is that TRACE32 PowerView will change the current working directory to the directory of the selected file before running the script.

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

Example:

```
ChDir.DO C:\my_scipts\my_startup.cmm
```

Automated Start-up Scripts

When a TRACE32 instance starts, the PRACTICE script **autostart.cmm** is executed, which then calls the following scripts:

- **system-settings.cmm** (from the TRACE32 system directory, usually C:\T32)
- **user-settings.cmm** (from the user settings directory: on Windows %APPDATA%\TRACE32 or ~/.trace32 otherwise)
- **work-settings.cmm** (from the current working directory)

With the command line option **-s** *<startup_script>* you can specify an additional PRACTICE script (*.cmm) which is automatically started afterwards.

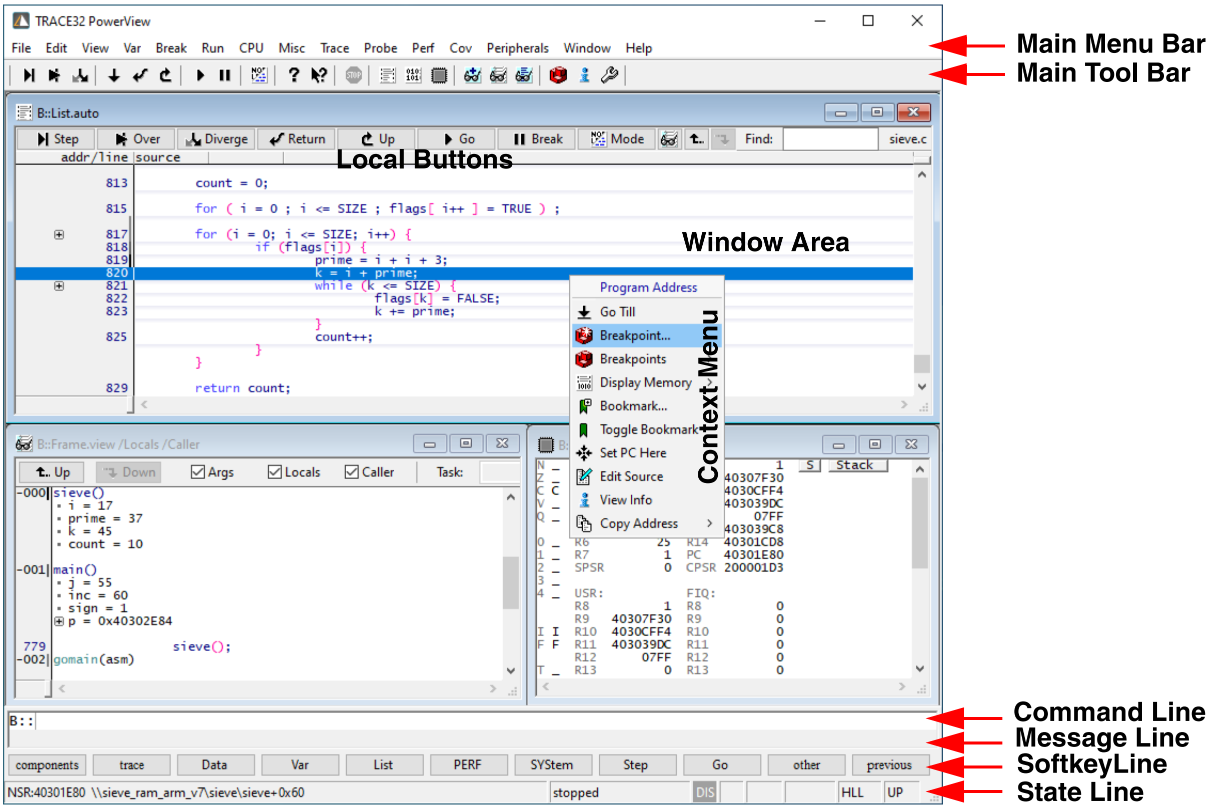
Example:

```
C:\T32\t32arm.exe -s C:\my_scripts\start.cmm
```

User Interface - TRACE32 PowerView

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

The following screenshot presents the main components of this interface.



```
addr/line source
  813       count = 0;
  815       for ( i = 0 ; i <= SIZE ; flags[ i++ ] = TRUE ) ;
  817       for (i = 0; i <= SIZE; i++) {
  818           if (flags[i]) {
  819               prime = i + i + 3;
  820               k = i + prime;
  821               while (k <= SIZE) {
  822                   flags[k] = FALSE;
  823                   k += prime;
                    }
  825               count++;
                }
            }
  829       return count;
```

```
-000|sieve()
     · i = 17
     · prime = 37
     · k = 45
     · count = 10

-001|main()
     · j = 55
     · inc = 60
     · sign = 1
     ⊞ p = 0x40302E84

 779                sieve();
-002|gomain(asm)
```

```
N _                        1
Z _                 40307F30
C C                 4030CFF4
V _                 403039DC
Q _                     07FF
                    403039C8
0 _  R6       25  R14  40301CD8
1 _  R7        1  PC   40301E80
2 _  SPSR      0  CPSR 200001D3
3 _
4 _  USR:         FIQ:
     R8        1  R8          0
     R9 40307F30  R9          0
I I  R10 4030CFF4 R10         0
F F  R11 403039DC R11         0
     R12    07FF  R12         0
T _  R13       0  R13         0
```

```
B::
```

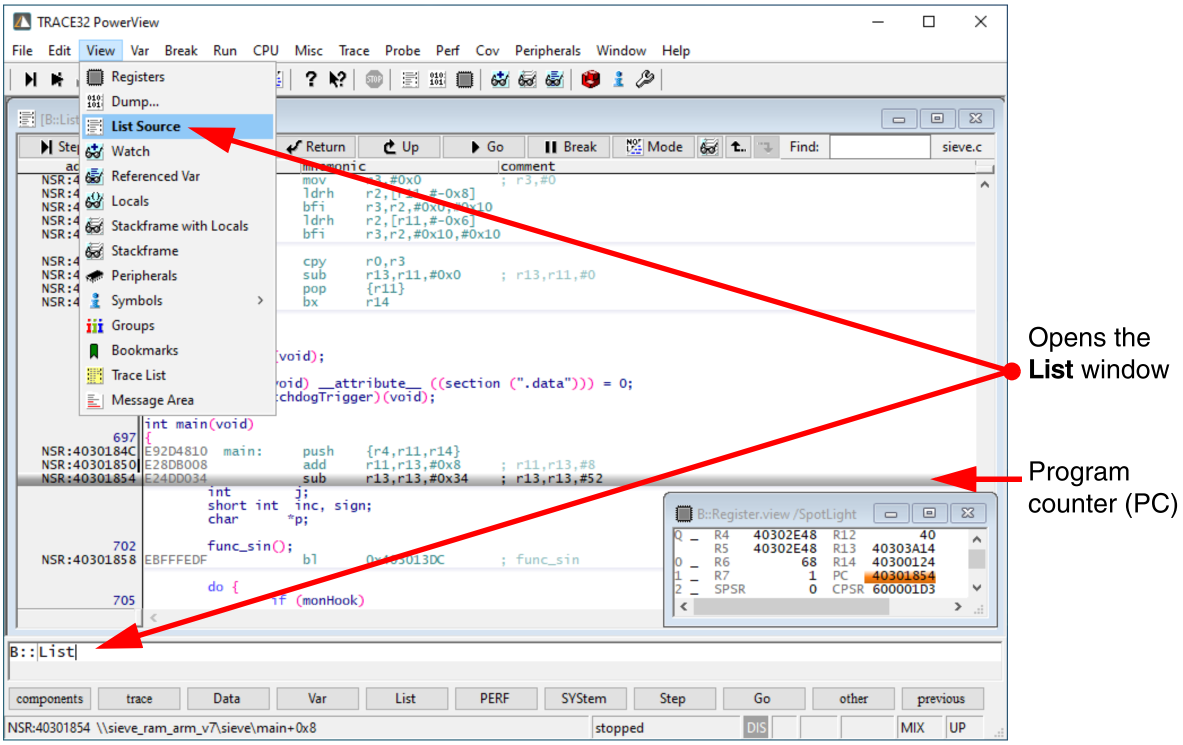
```
NSR:40301E80 \\sieve_ram_arm_v7\sieve\sieve+0x60    stopped    DIS    HLL    UP
```

We'll briefly explain the GUI using the **List** command and **List** window as an example. For a more comprehensive introduction, a video tutorial about the TRACE32 PowerView GUI is available here: **support.lauterbach.com/kb/articles/introduction-to-trace32-gui**

To open the **List** window, do one of the following:

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

The **List** window displays the code in both assembler mnemonic and HLL (High-Level Language). HLL refers to the programming language of your source code, e.g. C or C++.



```
      ad                               mnemonic                  comment
NSR:4                                  mov      r3,#0x0          ; r3,#0
NSR:4                                  ldrh     r2,[r11,#-0x8]
NSR:4                                  bfi      r3,r2,#0x0,#0x10
NSR:4                                  ldrh     r2,[r11,#-0x6]
NSR:4                                  bfi      r3,r2,#0x10,#0x10

NSR:4                                  cpy      r0,r3
NSR:4                                  sub      r13,r11,#0x0     ; r13,r11,#0
NSR:4                                  pop      {r11}
NSR:4                                  bx       r14

                              (void);

                              oid) __attribute__ ((section (".data"))) = 0;
                              chdogTrigger)(void);
                  int main(void)
             697  {
NSR:4030184C E92D4810  main:           push     {r4,r11,r14}
NSR:40301850 E28DB008                  add      r11,r13,#0x8     ; r11,r13,#8
NSR:40301854 E24DD034                  sub      r13,r13,#0x34    ; r13,r13,#52
                          int          j;
                          short int    inc, sign;
                          char         *p;

             702          func_sin();
NSR:40301858 EBFFFEDF                  bl       0x403013DC       ; func_sin

                          do {
             705                  if (monHook)
```

```
Q _   R4    40302E48   R12         40
      R5    40302E48   R13   40303A14
0 _   R6          68   R14   40300124
1 _   R7           1   PC    40301854
2 _   SPSR         0   CPSR  600001D3
```

```
B::List
```

```
NSR:40301854 \\sieve_ram_arm_v7\sieve\main+0x8    stopped    DIS    MIX    UP
```

In the **List** window, the gray bar indicates the position of the program counter (PC). In the screenshot above, it is located at 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, you can execute commands in TRACE32 PowerView using the following methods:

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
4. Using 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 entering frequently-used commands in the command line.

Examples:

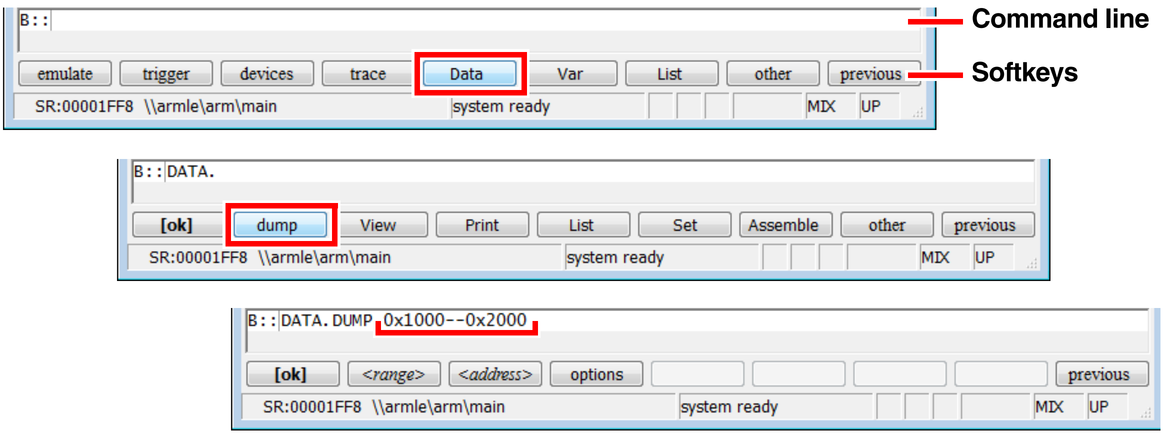
- Instead of the long form **`Register.view`** type just the short form **`r`** or **`R`**
- Instead of the long form **`List`** type just the short form **`l`** or **`L`**

The softkeys are located below the command line. The camel casing (i.e. upper and lower case letters) on any softkey indicates 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:

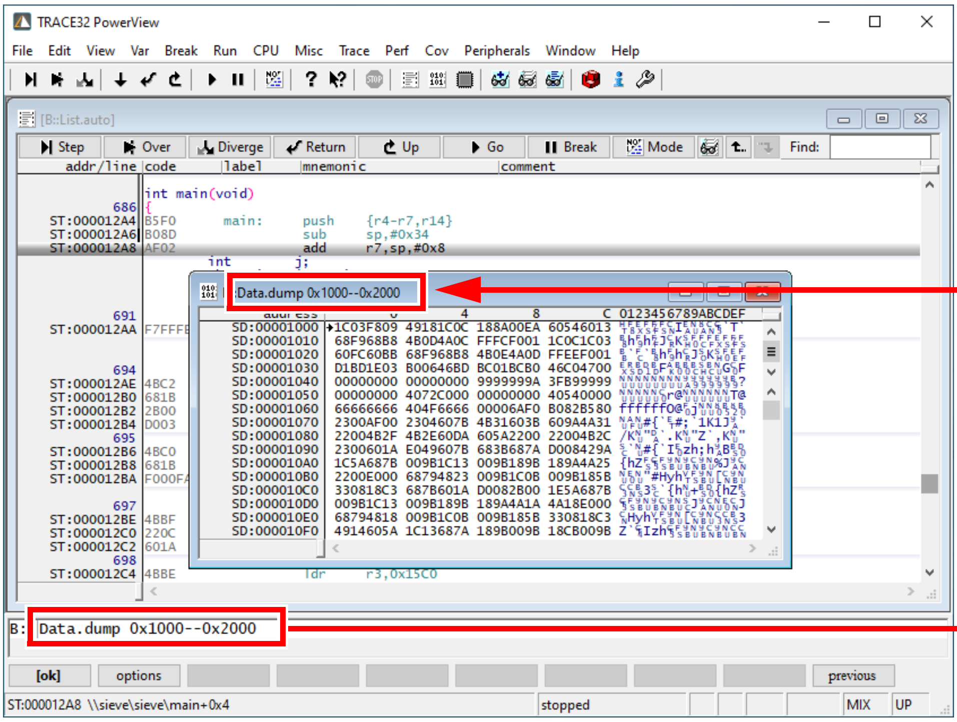
1. Click **Data**.
2. Click **dump**.
3. Type the *<range>* or *<address>* you want to dump. For example, **`0x1000--0x2000`**
4. Click **[ok]** to execute the command.

The **Data.dump** window will open.



```
B::
SR:00001FF8  \\armle\arm\main        system ready

B::DATA.
SR:00001FF8  \\armle\arm\main        system ready

B::DATA.DUMP 0x1000--0x2000
SR:00001FF8  \\armle\arm\main        system ready
```

The command used to open a window is displayed in the window caption, along with any parameters and options used.


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov Peripherals Window Help
[B::List.auto]
Step Over Diverge Return Up Go Break Mode Find:
[ok] options previous
ST:000012A8 \\sieve\sieve\main+0x4 stopped MIX UP

```
    addr/line code      label      mnemonic         comment
                int main(void)
          686   {
ST:000012A4     B5F0      main:     push    {r4-r7,r14}
ST:000012A6     B08D                sub     sp,#0x34
ST:000012A8     AF02                add     r7,sp,#0x8
                      int         j;
          691
ST:000012AA     F7FFFE
          694
ST:000012AE     4BC2
ST:000012B0     681B
ST:000012B2     2B00
ST:000012B4     D003
          695
ST:000012B6     4BC0
ST:000012B8     681B
ST:000012BA     F000FA
          697
ST:000012BE     4BBF
ST:000012C0     220C
ST:000012C2     601A
          698
ST:000012C4     4BBE                ldr     r3,0x15C0
```

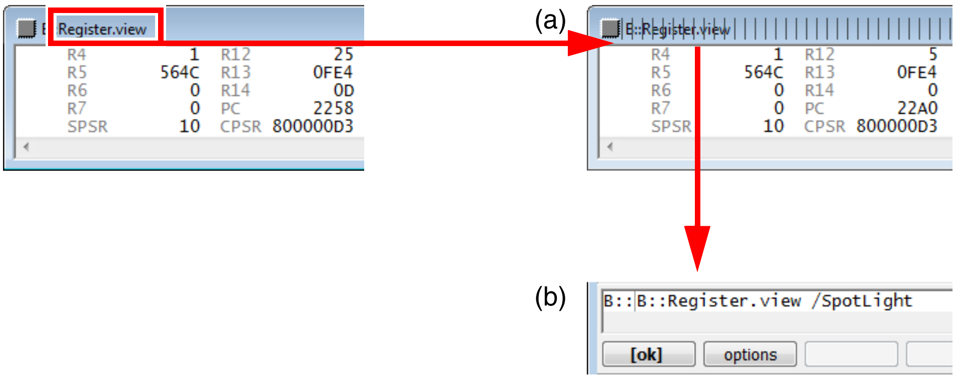
Data.dump 0x1000--0x2000

```
     address       0        4        8        C  0123456789ABCDEF
SD:00001000 ▸1C03F809 49181C0C 188A00EA 60546013
SD:00001010  68F968B8 4B0D4A0C FFFCF001 1C0C1C03
SD:00001020  60FC60BB 68F968B8 4B0E4A0D FFEEF001
SD:00001030  D1BD1E03 B00646BD BC01BCB0 46C04700
SD:00001040  00000000 00000000 9999999A 3FB99999
SD:00001050  00000000 4072C000 00000000 40540000
SD:00001060  66666666 404F6666 00006AF0 B082B580
SD:00001070  2300AF00 2304607B 4B31603B 609A4A31
SD:00001080  22004B2F 4B2E60DA 605A2200 22004B2C
SD:00001090  2300601A E049607B 683B687A D008429A
SD:000010A0  1C5A687B 009B1C13 009B189B 189A4A25
SD:000010B0  2200E000 68794823 009B1C0B 009B185B
SD:000010C0  330818C3 687B601A D0082B00 1E5A687B
SD:000010D0  009B1C13 009B189B 189A4A1A 4A18E000
SD:000010E0  68794818 009B1C0B 009B185B 330818C3
SD:000010F0  4914605A 1C13687A 189B009B 18CB009B
```

```
B: Data.dump 0x1000--0x2000
```

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



```
R4          1  R12        25
R5       564C  R13      0FE4
R6          0  R14        0D
R7          0  PC       2258
SPSR       10  CPSR 800000D3
```

```
R4          1  R12         5
R5       564C  R13      0FE4
R6          0  R14         0
R7          0  PC       22A0
SPSR       10  CPSR 800000D3
```

```
B::B::Register.view /SpotLight
```

4. Click **[ok]** to execute the modified command.

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

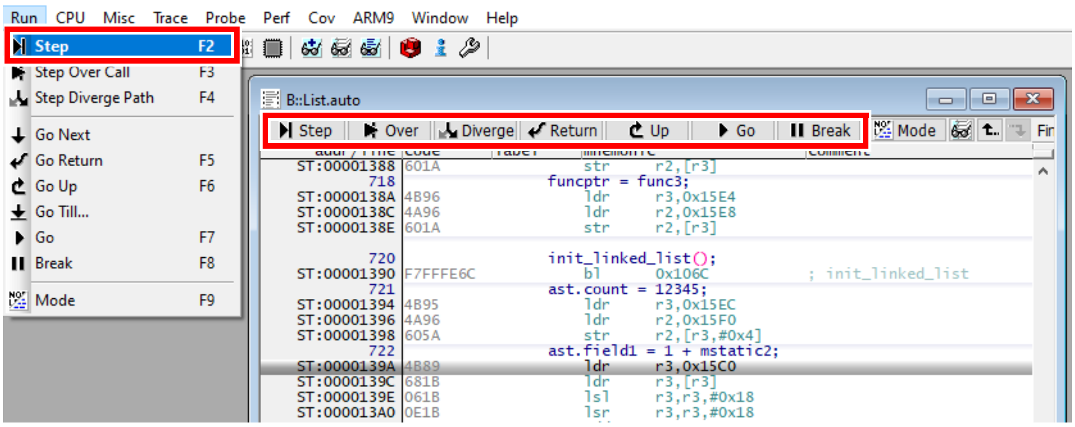
Debugging the Program

Basic Debug Commands

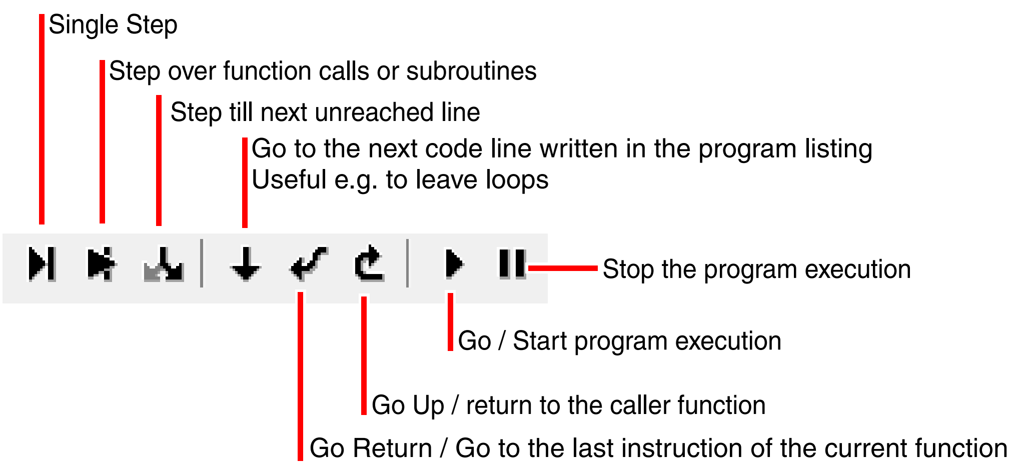
The basic debug commands are accessible from:

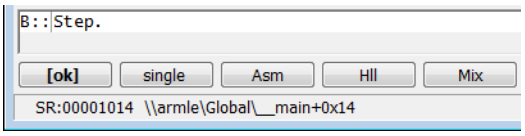
- the **Run** menu
- the toolbar of the **List** window
- the main toolbar
- the TRACE32 command line.

Single stepping ⏯ is one of these fundamental debug commands.

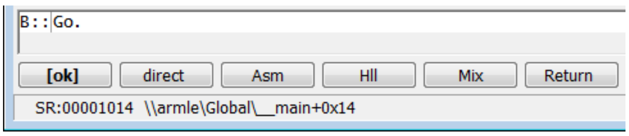


```
 ST:00001388 601A         str     r2,[r3]
         718          funcptr = func3;
 ST:0000138A 4B96         ldr     r3,0x15E4
 ST:0000138C 4A96         ldr     r2,0x15E8
 ST:0000138E 601A         str     r2,[r3]

         720          init_linked_list();
 ST:00001390 F7FFFE6C     bl      0x106C              ; init_linked_list
         721          ast.count = 12345;
 ST:00001394 4B95         ldr     r3,0x15EC
 ST:00001396 4A96         ldr     r2,0x15F0
 ST:00001398 605A         str     r2,[r3,#0x4]
         722          ast.field1 = 1 + mstatic2;
 ST:0000139A 4B89         ldr     r3,0x15C0
 ST:0000139C 681B         ldr     r3,[r3]
 ST:0000139E 061B         lsl     r3,r3,#0x18
 ST:000013A0 0E1B         lsr     r3,r3,#0x18
```


Single Step
Step over function calls or subroutines
Step till next unreached line
Go to the next code line written in the program listing
Useful e.g. to leave loops
Stop the program execution
Go / Start program execution
Go Up / return to the caller function
Go Return / Go to the last instruction of the current function



```
B::Step.
SR:00001014  \\armle\Global\__main+0x14
```



```
B::Go.
SR:00001014  \\armle\Global\__main+0x14
```

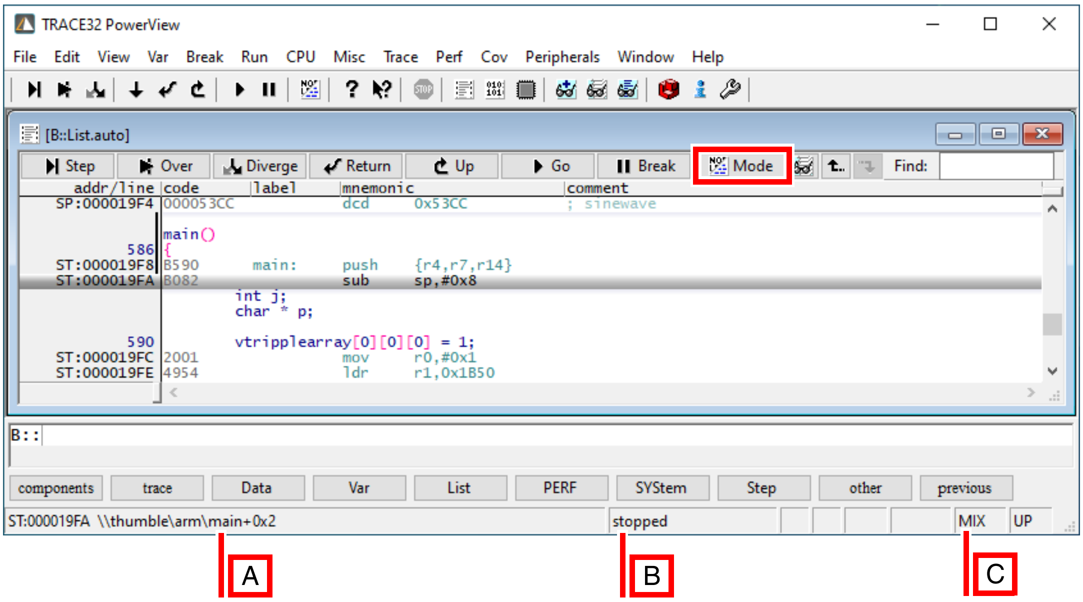
TRACE32 PowerView also offers more complex debug control commands. For example, you can step until an expression changes or becomes true.

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 PowerView main window:



```
    addr/line |code      |label    |mnemonic             |comment
  SP:000019F4 |000053CC             dcd      0x53CC       ; sinewave

              |main()
          586 |{
  ST:000019F8 |B590       main:     push     {r4,r7,r14}
  ST:000019FA |B082                 sub      sp,#0x8
                         int j;
                         char * p;

          590            vtripplearray[0][0][0] = 1;
  ST:000019FC |2001                 mov      r0,#0x1
  ST:000019FE |4954                 ldr      r1,0x1B50
```

```
ST:000019FA  \\thumble\arm\main+0x2        stopped        MIX   UP
```

The state line provides the following information

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

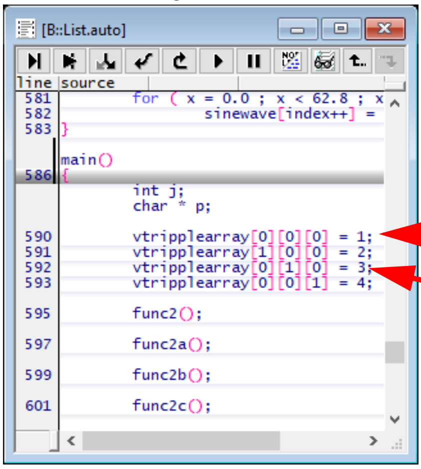
B The state of the debugger: **stopped** indicates that the program execution is stopped. At this point, you can inspect or modify memory, for example.

C The state line displays the currently selected debug mode, which can be:

- **HLL** (High Level Language)
- **ASM** (assembler)
- **MIX**ed mode showing both HLL and its corresponding assembler mnemonic.

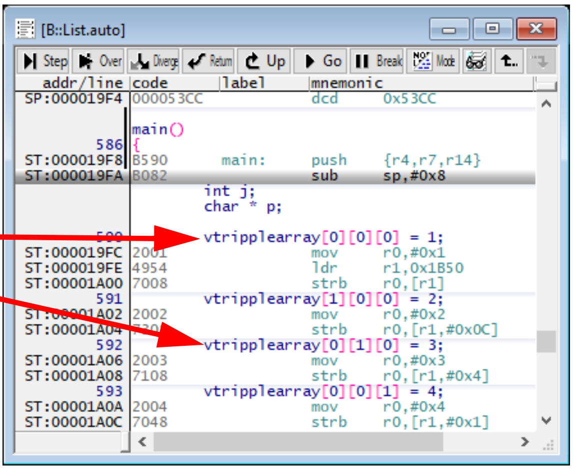
1. On the toolbar of the **List** window, click Mode **Mode** to toggle the debug mode to **HLL**.

Debug mode HLL



```
line source
581         for ( x = 0.0 ; x < 62.8 ; x
582                 sinewave[index++] =
583 }

    main()
586 {
            int j;
            char * p;

590         vtripplearray[0][0][0] = 1;
591         vtripplearray[1][0][0] = 2;
592         vtripplearray[0][1][0] = 3;
593         vtripplearray[0][0][1] = 4;

595         func2();

597         func2a();

599         func2b();

601         func2c();
```

Debug mode MIX



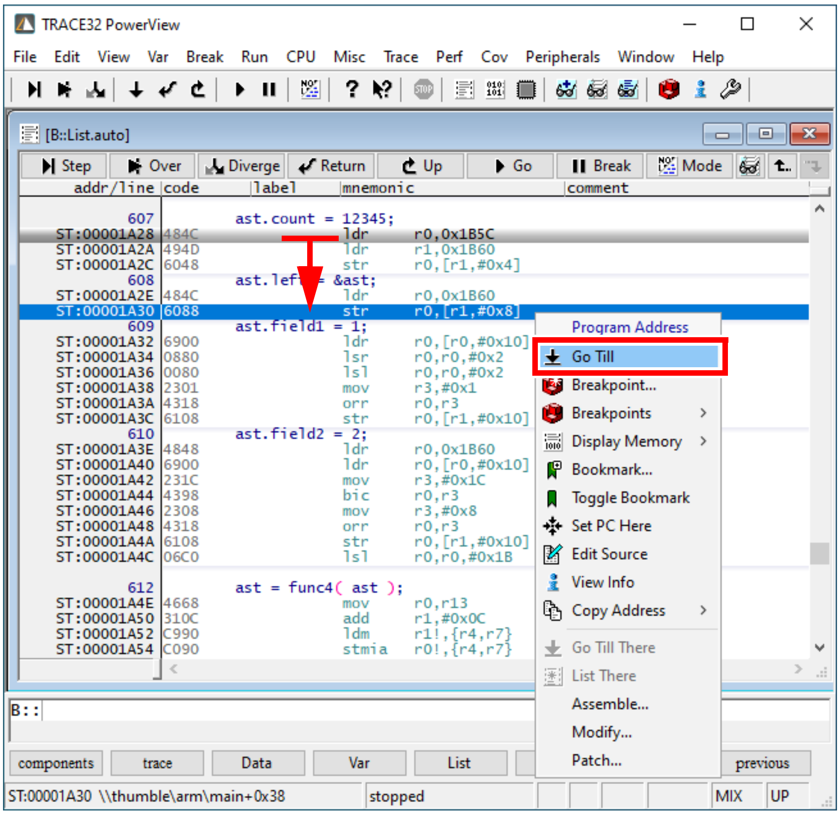
```
  addr/line code     label     mnemonic
SP:000019F4 000053CC           dcd      0x53CC

            main()
        586 {
ST:000019F8 B590     main:     push     {r4,r7,r14}
ST:000019FA B082               sub      sp,#0x8
                  int j;
                  char * p;

        590       vtripplearray[0][0][0] = 1;
ST:000019FC 2001               mov      r0,#0x1
ST:000019FE 4954               ldr      r1,0x1B50
ST:00001A00 7008               strb     r0,[r1]
        591       vtripplearray[1][0][0] = 2;
ST:00001A02 2002               mov      r0,#0x2
ST:00001A04 730[illegible]               strb     r0,[r1,#0x0C]
        592       vtripplearray[0][1][0] = 3;
ST:00001A06 2003               mov      r0,#0x3
ST:00001A08 7108               strb     r0,[r1,#0x4]
        593       vtripplearray[0][0][1] = 4;
ST:00001A0A 2004               mov      r0,#0x4
ST:00001A0C 7048               strb     r0,[r1,#0x1]
```

2. Click Step **Step**.
   In HLL mode, this action moves the program execution to the next source code line.

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

4. Click Step **Step**.
   This time, the step executes one assembler instruction.

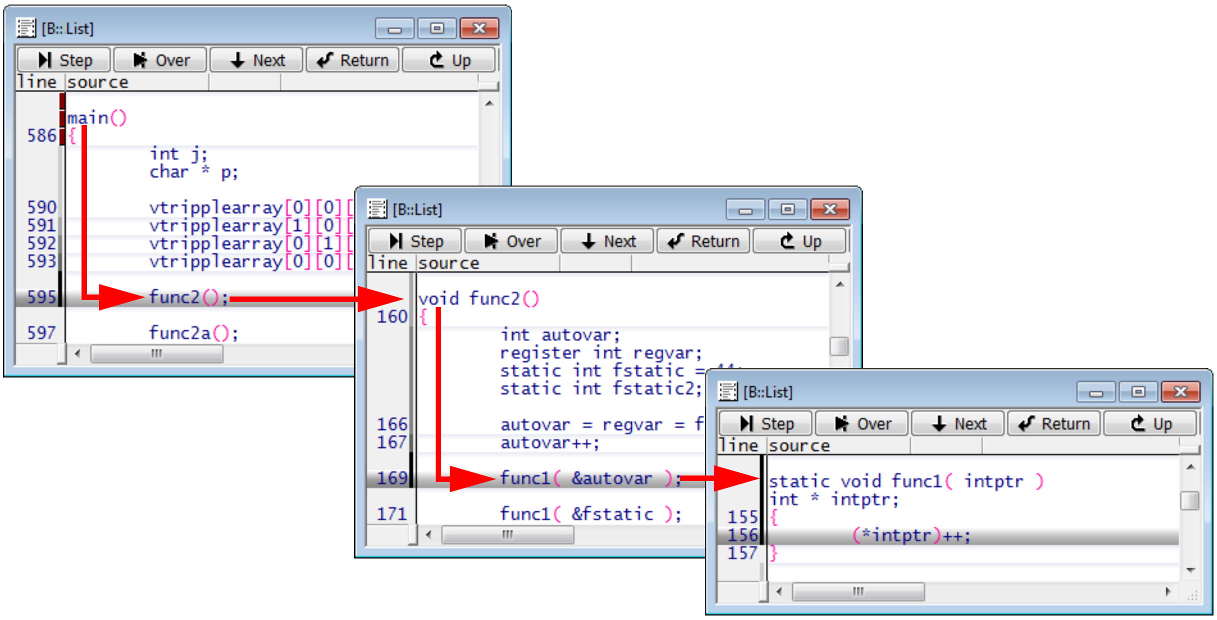
5. Right-click a code line, then select **Go Till**.
   Program execution starts, and stops when the program reaches the selected code line.



```
  addr/line code     label     mnemonic        comment

        607           ast.count = 12345;
ST:00001A28 484C               ldr      r0,0x1B5C
ST:00001A2A 494D               ldr      r1,0x1B60
ST:00001A2C 6048               str      r0,[r1,#0x4]
        608           ast.left = &ast;
ST:00001A2E 484C               ldr      r0,0x1B60
ST:00001A30 6088               str      r0,[r1,#0x8]
        609           ast.field1 = 1;
ST:00001A32 6900               ldr      r0,[r0,#0x10]
ST:00001A34 0880               lsr      r0,r0,#0x2
ST:00001A36 0080               lsl      r0,r0,#0x2
ST:00001A38 2301               mov      r3,#0x1
ST:00001A3A 4318               orr      r0,r3
ST:00001A3C 6108               str      r0,[r1,#0x10]
        610           ast.field2 = 2;
ST:00001A3E 4848               ldr      r0,0x1B60
ST:00001A40 6900               ldr      r0,[r0,#0x10]
ST:00001A42 231C               mov      r3,#0x1C
ST:00001A44 4398               bic      r0,r3
ST:00001A46 2308               mov      r3,#0x8
ST:00001A48 4318               orr      r0,r3
ST:00001A4A 6108               str      r0,[r1,#0x10]
ST:00001A4C 06C0               lsl      r0,r0,#0x1B

        612           ast = func4( ast );
ST:00001A4E 4668               mov      r0,r13
ST:00001A50 310C               add      r1,#0x0C
ST:00001A52 C990               ldm      r1!,{r4,r7}
ST:00001A54 C090               stmia    r0!,{r4,r7}

B::

ST:00001A30 \\thumble\arm\main+0x38   stopped   MIX   UP
```

Displaying the Stack Frame

For the next example, we will assume that we have the following call hierarchy: **`main()`** calls **`func2()`** and **`func2()`** calls **`func1()`**:



```
main()
{
        int j;
        char * p;

        vtripplearray[0][0][
        vtripplearray[1][0][
        vtripplearray[0][1][
        vtripplearray[0][0][

        func2();

        func2a();
```

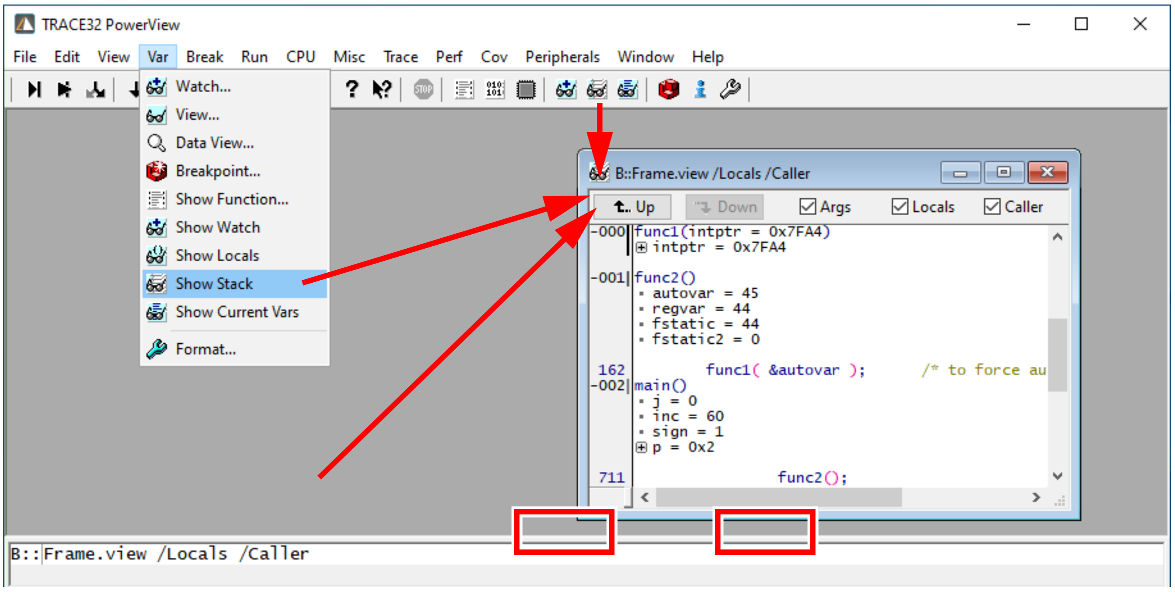
```
void func2()
{
        int autovar;
        register int regvar;
        static int fstatic =
        static int fstatic2;

        autovar = regvar = f
        autovar++;

        func1( &autovar );

        func1( &fstatic );
```

```
static void func1( intptr )
int * intptr;
{
        (*intptr)++;
}
```

Select **Show Stack** in the **Var** menu. This will open the **Frame.view** window, displaying the call hierarchy.

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

This screenshot corresponds to the calling hierarchy described above.



```
-000 func1(intptr = 0x7FA4)
      ⊞ intptr = 0x7FA4

-001 func2()
      ▪ autovar = 45
      ▪ regvar = 44
      ▪ fstatic = 44
      ▪ fstatic2 = 0

 162        func1( &autovar );       /* to force au
-002 main()
      ▪ j = 0
      ▪ inc = 60
      ▪ sign = 1
      ⊞ p = 0x2

 711            func2();
```

```
B::Frame.view /Locals /Caller
```

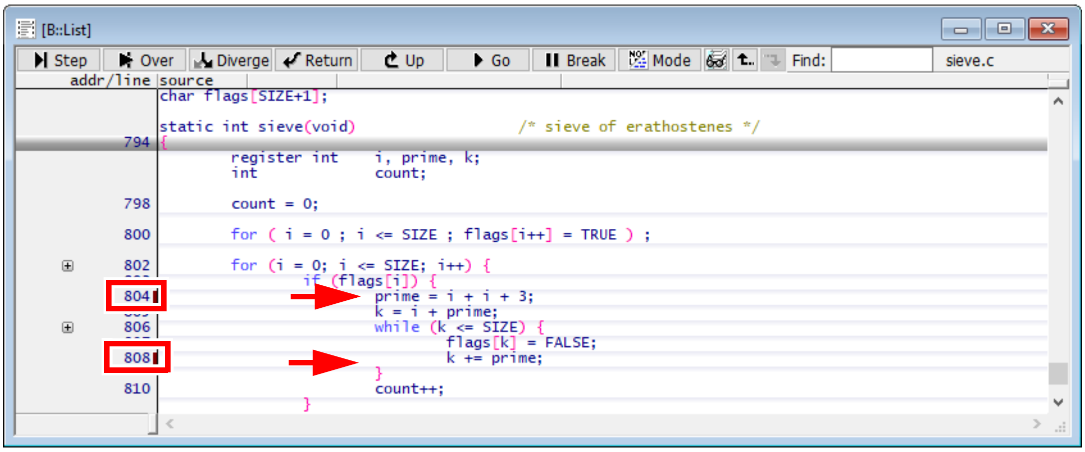
Breakpoints

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

Setting Breakpoints

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

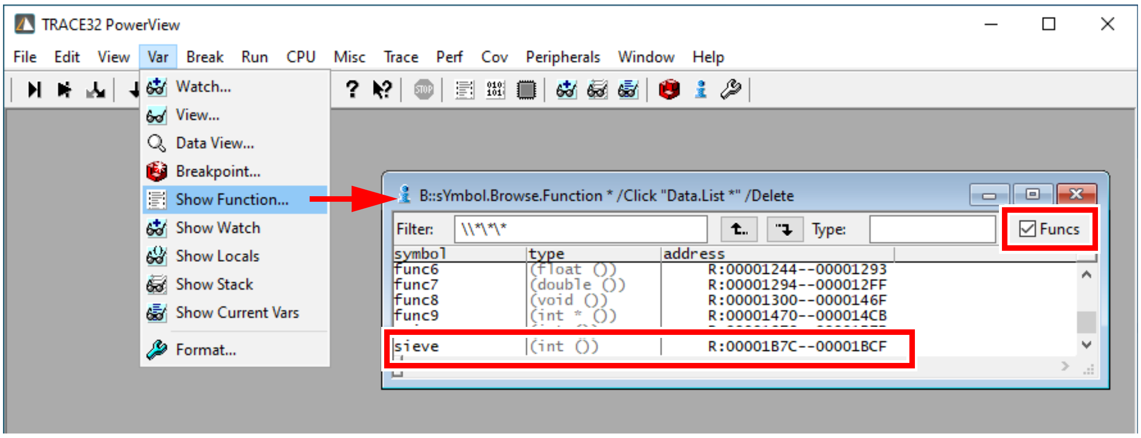
To set a program breakpoint, double-click a code line where you want to set the breakpoint. Ensure to click the white space in the code line, and not the code literal. All code lines with a program breakpoint are marked with a red vertical bar.



```
char flags[SIZE+1];

static int sieve(void)                  /* sieve of erathostenes */
{
        register int    i, prime, k;
        int             count;

        count = 0;

        for ( i = 0 ; i <= SIZE ; flags[i++] = TRUE ) ;

        for (i = 0; i <= SIZE; i++) {
                if (flags[i]) {
                        prime = i + i + 3;
                        k = i + prime;
                        while (k <= SIZE) {
                                flags[k] = FALSE;
                                k += prime;
                        }
                        count++;
                }
```

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

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



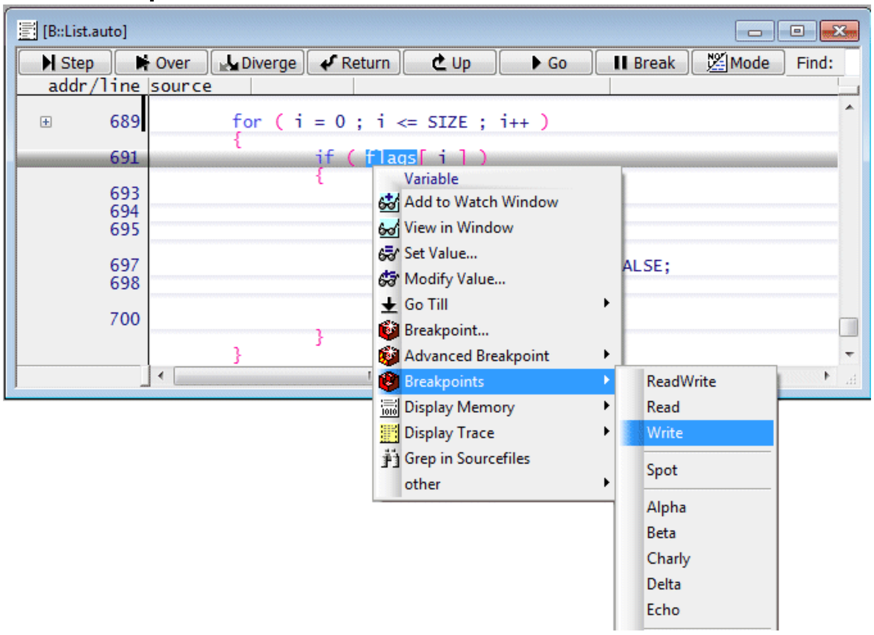
```
Filter: \\*\*\*
symbol   type           address
func6    (float ())     R:00001244--00001293
func7    (double ())    R:00001294--000012FF
func8    (void ())      R:00001300--0000146F
func9    (int * ())     R:00001470--000014CB
sieve    (int ())       R:00001B7C--00001BCF
```

2. Select the function you are interested in, for example `sieve`.
   The **List** window will open, 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.

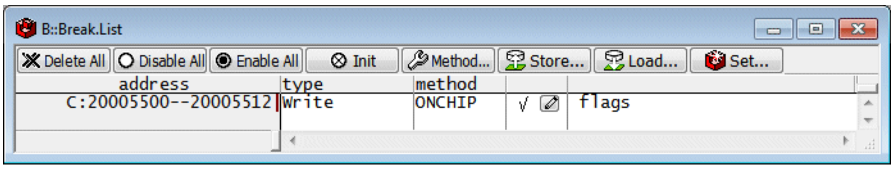
Setting Read/Write Breakpoints

You can set a breakpoint that halts the program execution at a read or write access to a memory location, such as global variable.

To set a breakpoint on the array **`flags`**, for instance, right-click on the array name in the **List** window then select **Breakpoints > Write**.



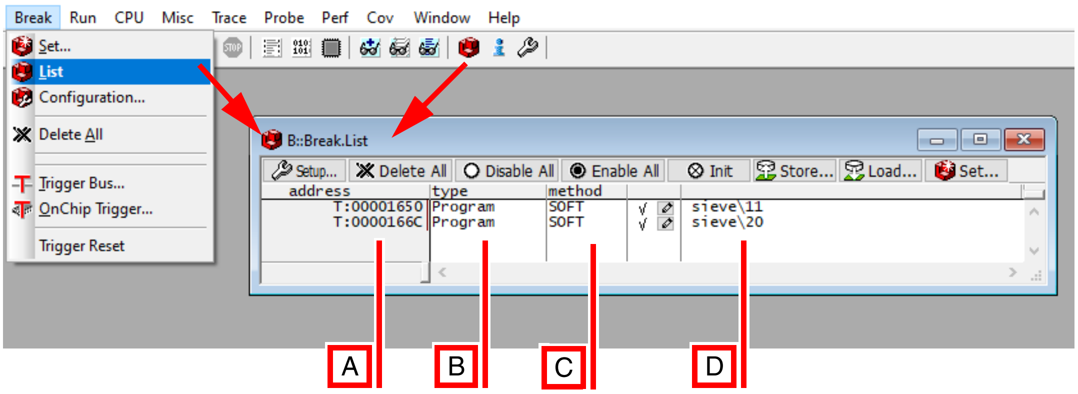
```
      for ( i = 0 ; i <= SIZE ; i++ )
      {
              if ( flags[ i ] )
              {
                                              ALSE;
              }
      }
```



```
     address           type          method
  C:20005500--20005512 Write        ONCHIP      √       flags
```

Listing all Breakpoints

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



```
address       type      method
T:00001650 Program   SOFT      √  sieve\11
T:0000166C Program   SOFT      √  sieve\20
```

A Address of the breakpoint.

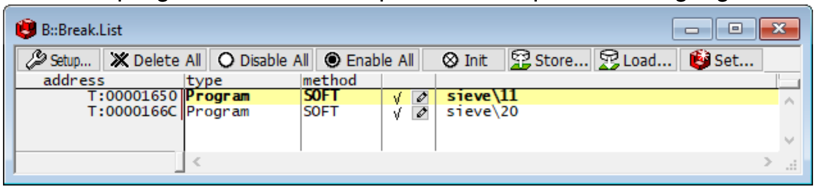
B Breakpoint type, for example Program, Read, Write

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.



```
address       type      method
T:00001650 Program   SOFT      √  sieve\11
T:0000166C Program   SOFT      √  sieve\20
```

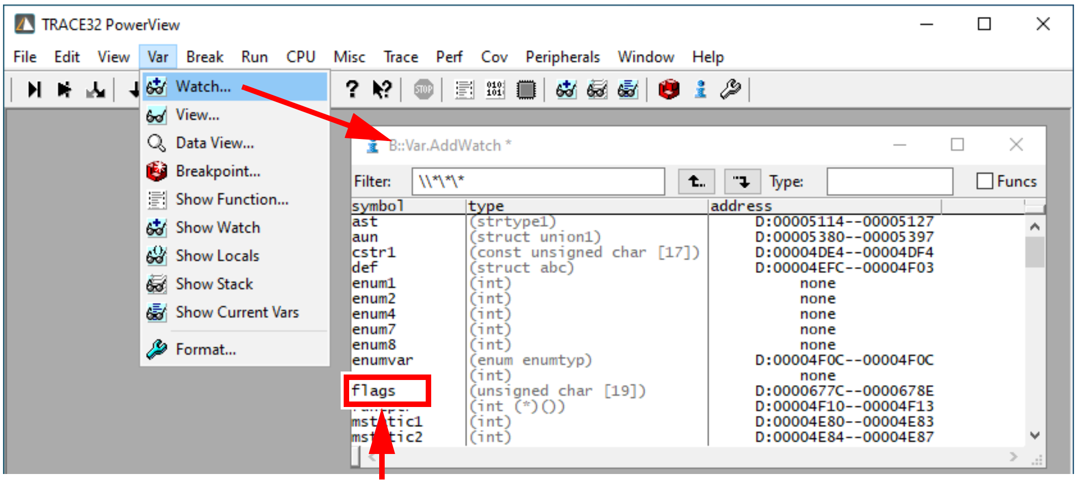
Variables

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

Displaying Variables

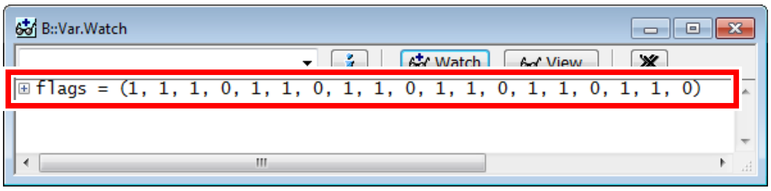
Let's display the variables `flags`, `def`, and `ast`.

1. Choose **Var** > **Watch... from the menu.**
   The **Var.AddWatch** window will open, displaying the variables known to the symbol database.



```
symbol      type                          address
ast         (strtype1)                    D:00005114--00005127
aun         (struct union1)               D:00005380--00005397
cstr1       (const unsigned char [17])    D:00004DE4--00004DF4
def         (struct abc)                  D:00004EFC--00004F03
enum1       (int)                         none
enum2       (int)                         none
enum4       (int)                         none
enum7       (int)                         none
enum8       (int)                         none
enumvar     (enum enumtyp)                D:00004F0C--00004F0C
            (int)                         none
flags       (unsigned char [19])          D:0000677C--0000678E
            (int (*)())                   D:00004F10--00004F13
mstatic1    (int)                         D:00004E80--00004E83
mstatic2    (int)                         D:00004E84--00004E87
```

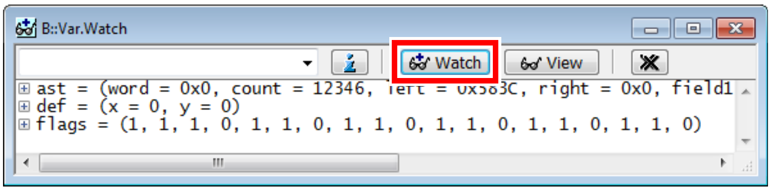
2. Double-click the variable `flags`.
   The **Var.Watch** window will open, displaying the selected variable.



```
⊞ flags = (1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0)
```

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.



```
⊞ ast = (word = 0x0, count = 12346, left = 0x583C, right = 0x0, field1
⊞ def = (x = 0, y = 0)
⊞ flags = (1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0)
```

   - 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.

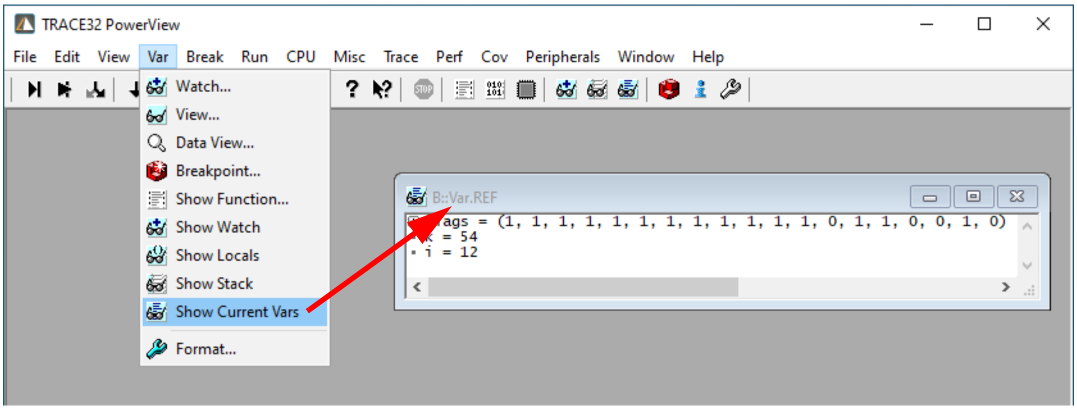
- If you want to display a more complex structure or an array in a separate window, select the menu **Var** > **View**.

Displaying Variables of the Current Program Context

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

```
Register.Set PC sieve
```

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



```
ags = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0)
k = 54
▪ i = 12
```

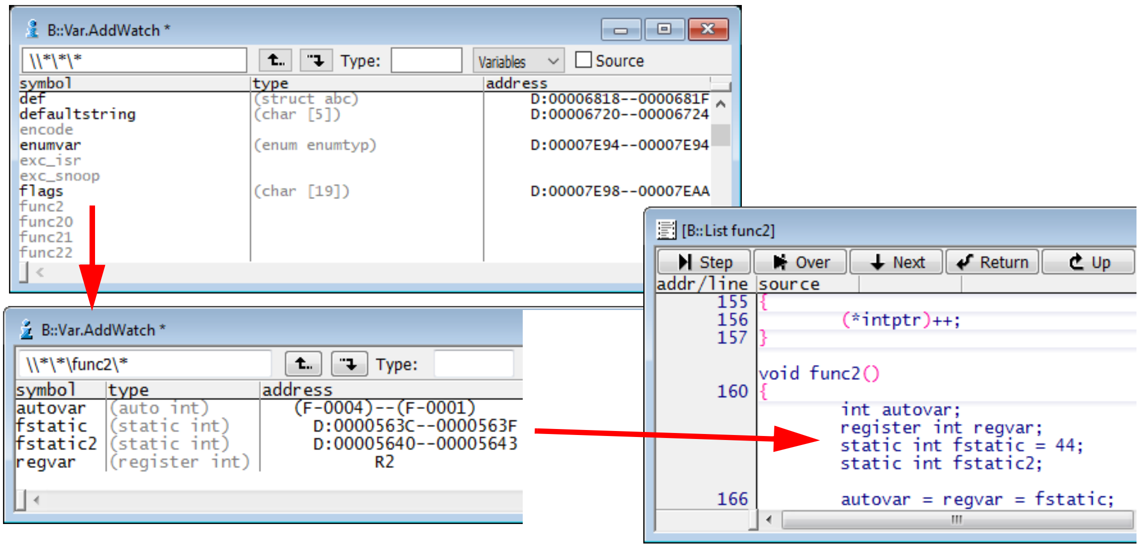
3. Click **Step** on the TRACE32 PowerView toolbar to execute a few single steps.
   The **Var.REF** window is updated automatically.

Using the Symbol Browser

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

1. Select **Var** from the menu, then choose **Watch...**
   The **Var.AddWatch** window will open, allowing you to browse through the contents of the symbol database. Global variables are displayed in black and functions in gray. Double-clicking a function will display its local variables are displayed.

2. In the **Var.AddWatch** window, double-click **`func2`**.



```
\\*\*\*
symbol          type               address
def             (struct abc)       D:00006818--0000681F
defaultstring   (char [5])         D:00006720--00006724
encode
enumvar         (enum enumtyp)     D:00007E94--00007E94
exc_isr
exc_snoop
flags           (char [19])        D:00007E98--00007EAA
func2
func20
func21
func22
```

```
\\*\*\func2\*
symbol    type            address
autovar   (auto int)      (F-0004)--(F-0001)
fstatic   (static int)    D:0000563C--0000563F
fstatic2  (static int)    D:00005640--00005643
regvar    (register int)  R2
```

```
addr/line source
          {
                  (*intptr)++;
          }

          void func2()
          {
                  int autovar;
                  register int regvar;
                  static int fstatic = 44;
                  static int fstatic2;

                  autovar = regvar = fstatic;
```

Formatting Variables

To format the display of variables with global settings:

1. Choose **Var** from the menu, then select **Format**.

2. In the **SETUP.Var** window, configure your settings. TRACE32 applies your settings to all **Var.view** windows that you open *afterwards*.

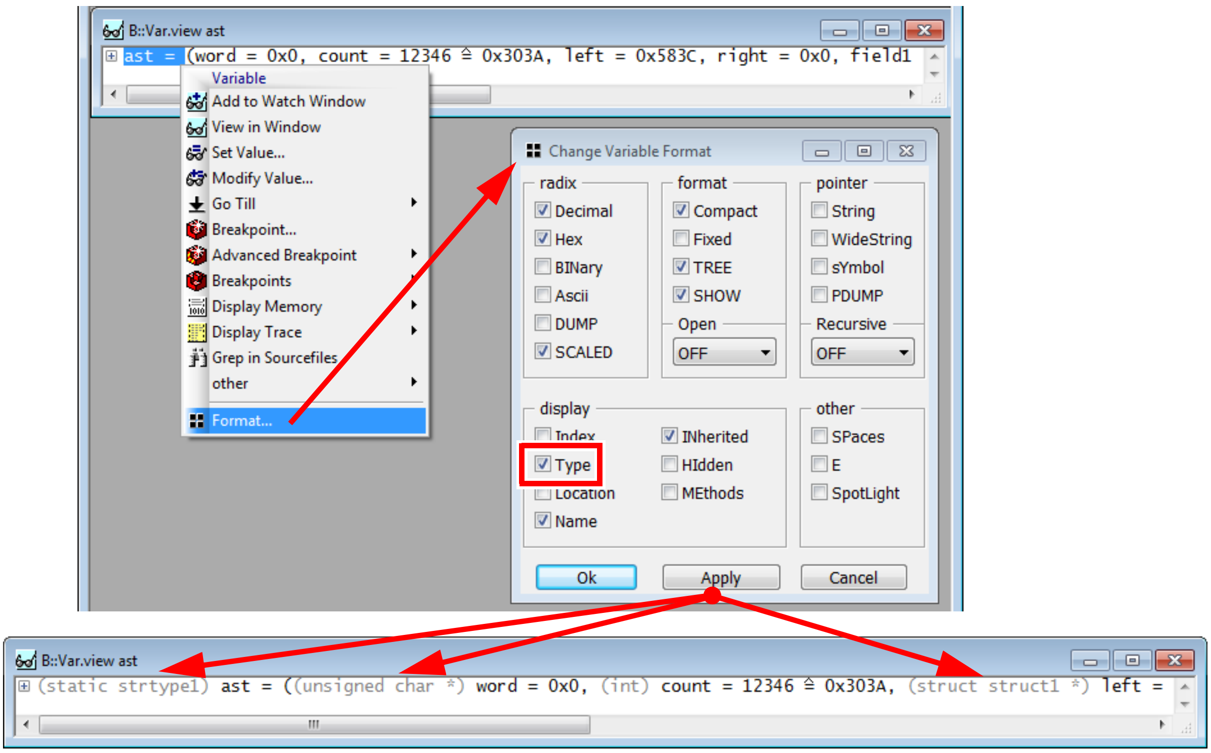
To format the display of an individual variable:

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.



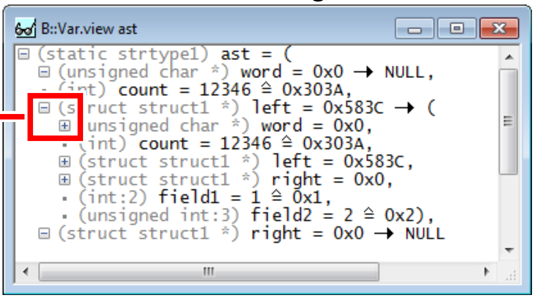
```
⊞ ast = (word = 0x0, count = 12346 ≙ 0x303A, left = 0x583C, right = 0x0, field1
```

radix: [✓ Decimal] [✓ Hex] BINary Ascii DUMP [✓ SCALED]
format: [✓ Compact] Fixed [✓ TREE] [✓ SHOW]
Open: OFF
pointer: String WideString sYmbol PDUMP
Recursive: OFF
display: Index [✓ Type] Location [✓ Name] [✓ INherited] HIdden MEthods
other: SPaces E SpotLight

```
⊞ (static strtype1) ast = ((unsigned char *) word = 0x0, (int) count = 12346 ≙ 0x303A, (struct struct1 *) left =
```

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

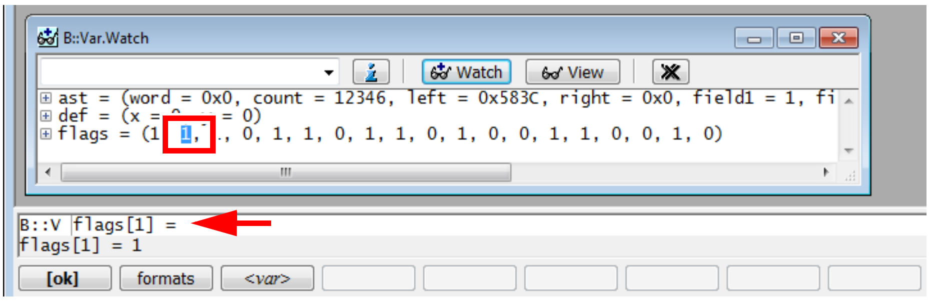
Click **+** and **-** to expand and collapse the tree.



```
⊟ (static strtype1) ast = (
  ⊟ (unsigned char *) word = 0x0 → NULL,
  · (int) count = 12346 ≙ 0x303A,
  ⊟ (struct struct1 *) left = 0x583C → (
    ⊞ (unsigned char *) word = 0x0,
    · (int) count = 12346 ≙ 0x303A,
    ⊞ (struct struct1 *) left = 0x583C,
    ⊞ (struct struct1 *) right = 0x0,
    · (int:2) field1 = 1 ≙ 0x1,
    · (unsigned int:3) field2 = 2 ≙ 0x2),
  ⊟ (struct struct1 *) right = 0x0 → NULL
```

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**



```
⊞ ast = (word = 0x0, count = 12346, left = 0x583C, right = 0x0, field1 = 1, fi
⊞ def = (x = ... = 0)
⊞ flags = (1 1, ., 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0)
```

```
B::V flags[1] =
flags[1] = 1
```

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

Memory

Displaying Memory

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

   - Choose **View** from the menu then select **Dump**,

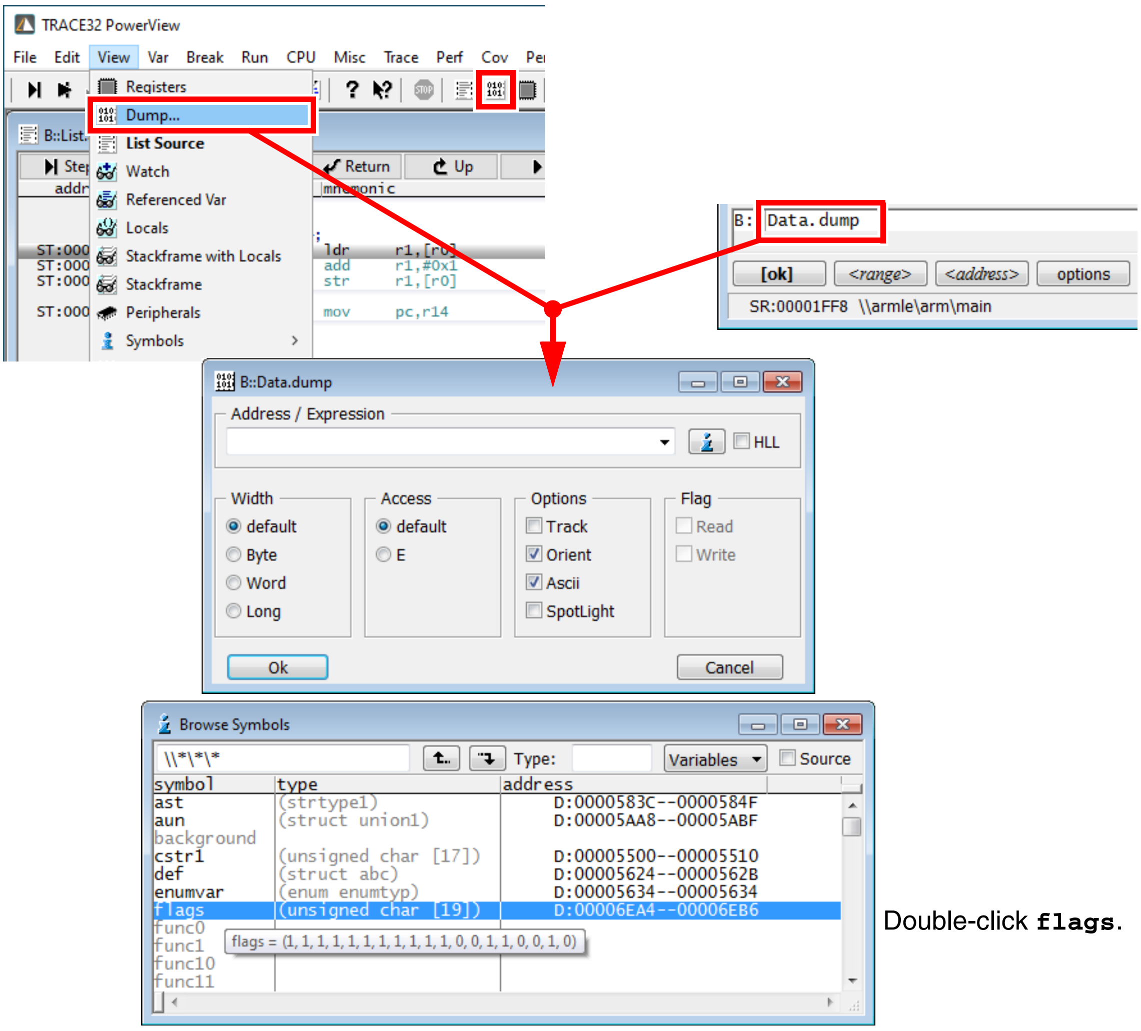
   - Click **Memory Dump** on the toolbar,

   - Type: **`Data.dump`** in the TRACE32 command line. 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**.



```
ST:000    ldr    r1,[r0]
ST:000    add    r1,#0x1
ST:000    str    r1,[r0]
ST:000    mov    pc,r14
```

```
B: Data.dump
SR:00001FF8  \\armle\arm\main
```

B::Data.dump — Address / Expression: (empty), HLL [ ]
Width: [● default] ○ Byte ○ Word ○ Long
Access: [● default] ○ E
Options: Track [ ], Orient [✓], Ascii [✓], SpotLight [ ]
Flag: Read [ ], Write [ ]

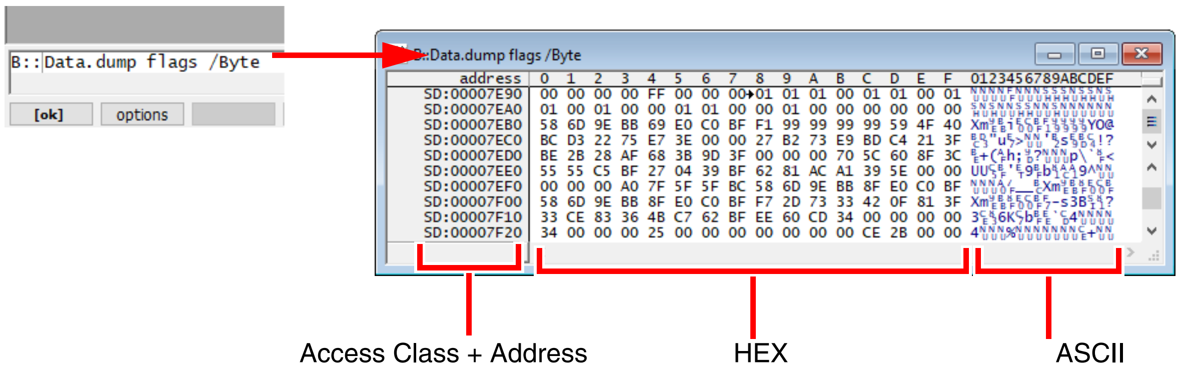
Browse Symbols — `\\*\*\*`, Type: Variables, Source [ ]

```
symbol      type                     address
ast         (strtype1)               D:0000583C--0000584F
aun         (struct union1)          D:00005AA8--00005ABF
background
cstr1       (unsigned char [17])     D:00005500--00005510
def         (struct abc)             D:00005624--0000562B
enumvar     (enum enumtyp)           D:00005634--00005634
flags       (unsigned char [19])     D:00006EA4--00006EB6
func0
func1
func10
func11
```

```
flags = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0)
```

Double-click **`flags`**.

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



```
B::Data.dump flags /Byte
```

```
B::Data.dump flags /Byte
     address   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
 SD:00007E90  00 00 00 00 FF 00 00 00 01 01 01 00 01 01 00 01
 SD:00007EA0  01 00 01 00 00 01 01 00 00 01 00 00 00 00 00 00
 SD:00007EB0  58 6D 9E BB 69 E0 C0 BF F1 99 99 99 99 59 4F 40
 SD:00007EC0  BC D3 22 75 E7 3E 00 00 27 B2 73 E9 BD C4 21 3F
 SD:00007ED0  BE 2B 28 AF 68 3B 9D 3F 00 00 00 70 5C 60 8F 3C
 SD:00007EE0  55 55 C5 BF 27 04 39 BF 62 81 AC A1 39 5E 00 00
 SD:00007EF0  00 00 00 A0 7F 5F 5F BC 58 6D 9E BB 8F E0 C0 BF
 SD:00007F00  58 6D 9E BB 8F E0 C0 BF F7 2D 73 33 42 0F 81 3F
 SD:00007F10  33 CE 83 36 4B C7 62 BF EE 60 CD 34 00 00 00 00
 SD:00007F20  34 00 00 00 25 00 00 00 00 00 00 00 CE 2B 00 00
```

There are different ways to define an address range:

- *<start_address>--<end_address>* (SD is an access class)

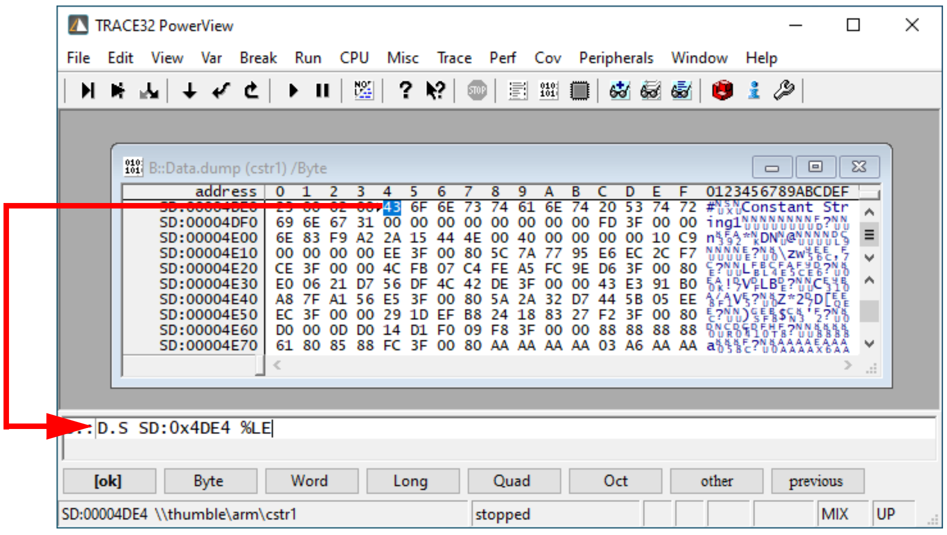
  ```
  Data.dump SD:0x5530--SD:0x554F
  ```

- *<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`**


TRACE32 PowerView
File Edit View Var Break Run CPU Misc Trace Perf Cov Peripherals Window Help
[ok]
Byte
Word
Long
Quad
Oct
other
previous
MIX
UP

```
B::Data.dump (cstr1) /Byte
     address   0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
 SD:00004DE0  23 00 02 00 43 6F 6E 73 74 61 6E 74 20 53 74 72   Constant Str
 SD:00004DF0  69 6E 67 31 00 00 00 00 00 00 00 00 FD 3F 00 00  ing1
 SD:00004E00  6E 83 F9 A2 2A 15 44 4E 00 40 00 00 00 00 10 C9
 SD:00004E10  00 00 00 00 EE 3F 00 80 5C 7A 77 95 E6 EC 2C F7
 SD:00004E20  CE 3F 00 00 4C FB 07 C4 FE A5 FC 9E D6 3F 00 80
 SD:00004E30  E0 06 21 D7 56 DF 4C 42 DE 3F 00 00 43 E3 91 B0
 SD:00004E40  A8 7F A1 56 E5 3F 00 80 5A 2A 32 D7 44 5B 05 EE
 SD:00004E50  EC 3F 00 00 29 1D EF B8 24 18 83 27 F2 3F 00 80
 SD:00004E60  D0 00 0D D0 14 D1 F0 09 F8 3F 00 00 88 88 88 88
 SD:00004E70  61 80 85 88 FC 3F 00 80 AA AA AA AA 03 A6 AA AA
```

```
B::D.S SD:0x4DE4 %LE
```

```
SD:00004DE4  \\thumble\arm\cstr1    stopped
```

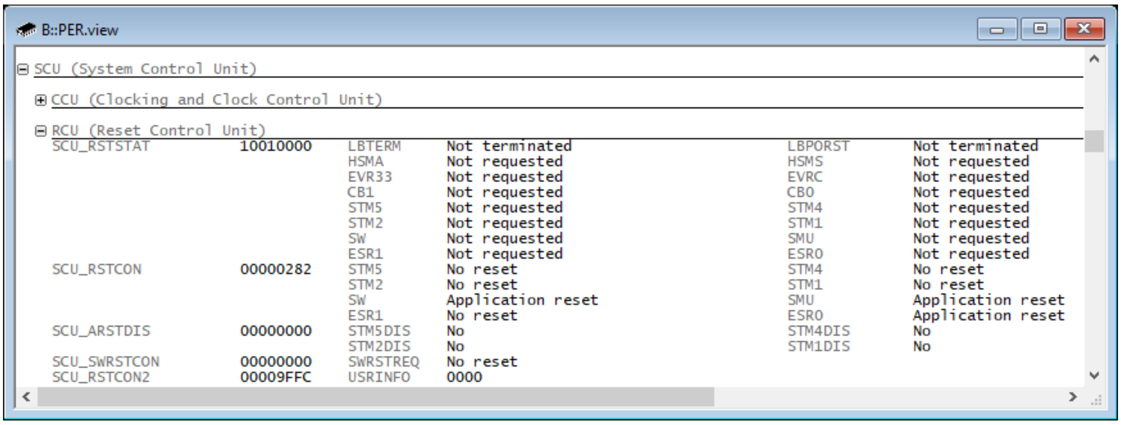
2. Enter the new value directly after **%LE**, and then confirm with **[ok]**.
   (**%LE** stands for Little Endian).

Peripheral View

TRACE32 supports a freely configurable window for displaying and manipulating configuration registers and on-chip peripheral registers at a logical level. Predefined peripheral files are available for most standard processors/chips.

You can open the peripheral register view in the TRACE32 by selection the **CPU** menu, then **Peripherals**, or by using the command **PER.view** in the TRACE32 command line.

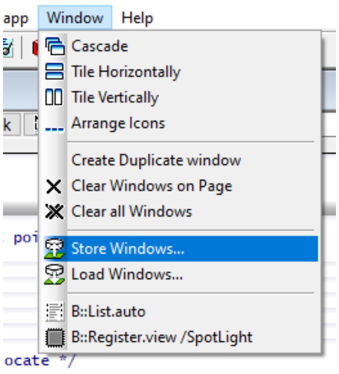
PER.view	Display peripheral registers

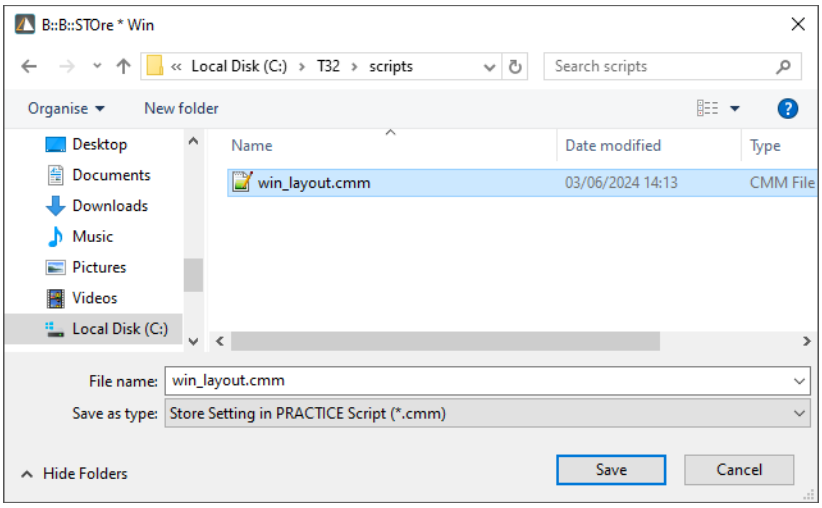


```
B::PER.view
⊟ SCU (System Control Unit)
  ⊞ CCU (Clocking and Clock Control Unit)
  ⊟ RCU (Reset Control Unit)
    SCU_RSTSTAT      10010000   LBTERM    Not terminated      LBPORST   Not terminated
                                HSMA      Not requested       HSMS      Not requested
                                EVR33     Not requested       EVRC      Not requested
                                CB1       Not requested       CB0       Not requested
                                STM5      Not requested       STM4      Not requested
                                STM2      Not requested       STM1      Not requested
                                SW        Not requested       SMU       Not requested
                                ESR1      Not requested       ESR0      Not requested
    SCU_RSTCON       00000282   STM5      No reset            STM4      No reset
                                STM2      No reset            STM1      No reset
                                SW        Application reset   SMU       Application reset
                                ESR1      No reset            ESR0      Application reset
    SCU_ARSTDIS      00000000   STM5DIS   No                  STM4DIS   No
                                STM2DIS   No                  STM1DIS   No
    SCU_SWRSTCON     00000000   SWRSTREQ  No reset
    SCU_RSTCON2      00009FFC   USRINFO   0000
```

Store Window Configuration

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





Name	Date modified	Type
win_layout.cmm	03/06/2024 14:13	CMM File

File name: win_layout.cmm

Save as type: Store Setting in PRACTICE Script (*.cmm)

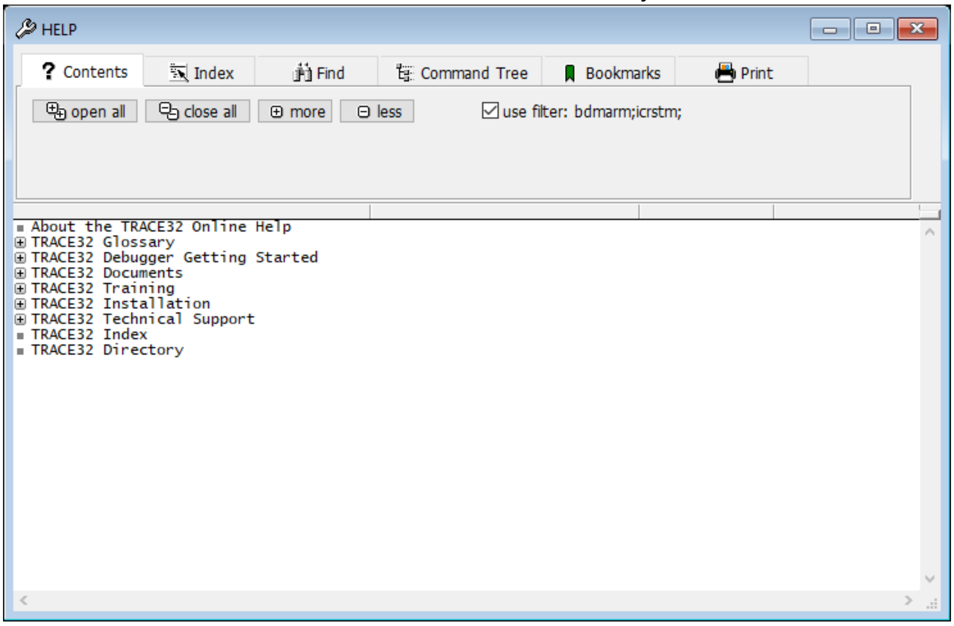
The saved window layout can be loaded again for the next debug session with the ***Load Windows…*** from the ***Window*** menu.

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.

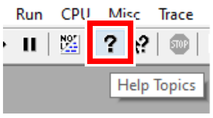


use filter: bdmarm;icrstm;

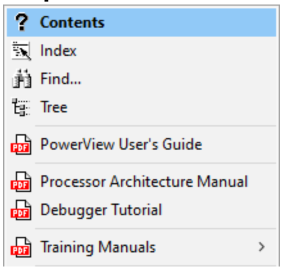
```
■ About the TRACE32 Online Help
⊞ TRACE32 Glossary
⊞ TRACE32 Debugger Getting Started
⊞ TRACE32 Documents
⊞ TRACE32 Training
⊞ TRACE32 Installation
⊞ TRACE32 Technical Support
■ TRACE32 Index
■ TRACE32 Directory
```

There are different ways to open the TRACE32 online help:

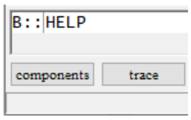
- **Help Topics** button on the toolbar



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



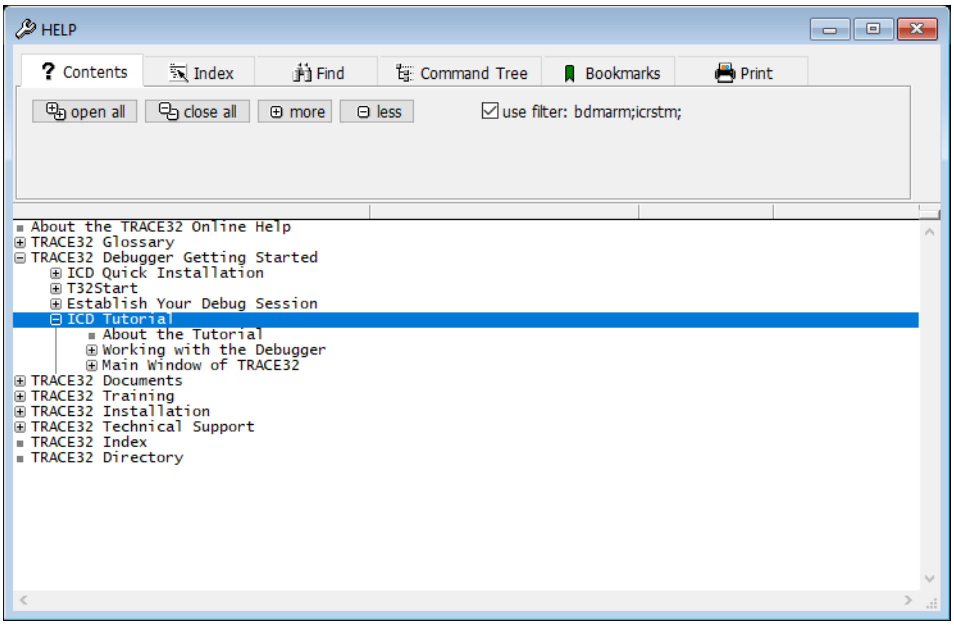
- **HELP** command in the command line



```
B::HELP
```

- **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.



use filter: bdmarm;icrstm;

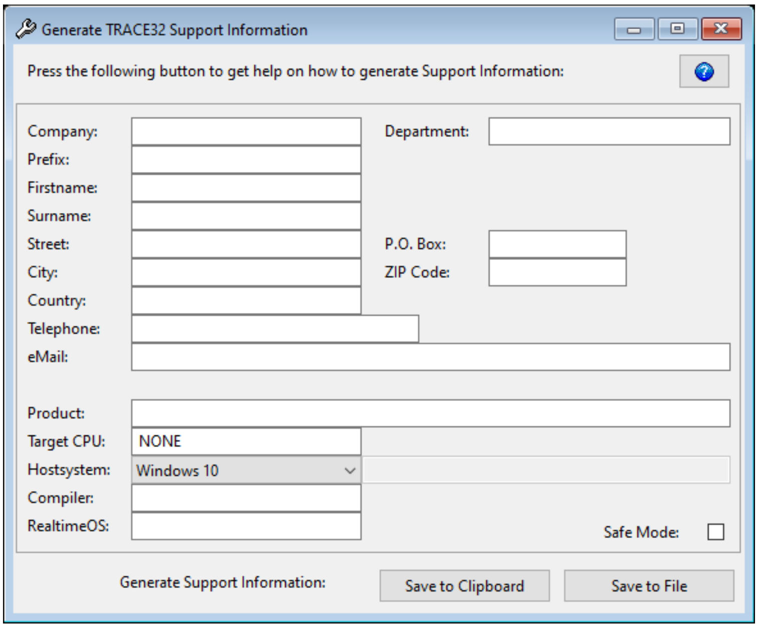
```
■ About the TRACE32 Online Help
⊞ TRACE32 Glossary
⊟ TRACE32 Debugger Getting Started
    ⊞ ICD Quick Installation
    ⊞ T32Start
    ⊞ Establish Your Debug Session
    ⊟ ICD Tutorial
        ■ About the Tutorial
        ⊞ Working with the Debugger
        ⊞ Main Window of TRACE32
⊞ TRACE32 Documents
⊞ TRACE32 Training
⊞ TRACE32 Installation
⊞ TRACE32 Technical Support
■ TRACE32 Index
■ TRACE32 Directory
```

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

Contacting the Lauterbach Support

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 …**



Generate TRACE32 Support Information

Press the following button to get help on how to generate Support Information:

Company:		Department:	
Prefix:			
Firstname:			
Surname:			
Street:		P.O. Box:	
City:		ZIP Code:	
Country:			
Telephone:			
eMail:			
Product:			
Target CPU:	NONE		
Hostsystem:	Windows 10		
Compiler:			
RealtimeOS:		Safe Mode:	☐

Generate Support Information:

2. Fill in the form then click **Save to File**,

3. Send the system information as an attachment to support@lauterbach.com together with the problem description.