LAUTERBACH
DEVELOPMENT TOOLS

TRACE32 as GDB Back-End

Release 02.2026

MANUAL

TRACE32 as GDB Back-End

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .....

GDB Support .....

TRACE32 as GDB Back-End 1

History 3

Introduction 3

Related Documents 3

Supported Architectures 4

Operation Theory and Restrictions 5

TRACE32 Setup 6

GDB Front-Ends Setup 7

The GNU Project Debugger GDB 7

Eclipse 9

Microsoft Visual Studio 12

Visual Studio Code 14

JetBrains CLion 17

Remote Serial Protocol 18

Protocol Extensions 18

Symmetrical Multiprocessing Support 18

TRACE32 as GDB Back-End

Version 03-Apr-2026

History

09-Jul-2025 New chapter: JetBrains CLion as Front end to TRACE32.

Introduction

TRACE32 PowerView implements a GDB server/stub functionality. This provides an interface to any application using the GDB Remote Serial Protocol (RSP) to control TRACE32 PowerView via TCP.

For the end users, this document presents:

- The supported target architectures.
- The operation theory of the GDB back-end and the solution restrictions.
- How to configure TRACE32 as a GDB back-end.
- TRACE32 integration to some example front-ends via the GDB interface.

For the users interested in the integration of TRACE32 with their custom GDB front-end, this document presents the limitations of the RSP and the protocol extensions defined to overcome them.

For information about using TRACE32 to debug e.g a virtual target over its GDB stub or to debug an application over gdbserver, please refer to **"TRACE32 as GDB Front-End"** (frontend_gdb.pdf).

Related Documents

- For information about how to install TRACE32, see **"TRACE32 Installation Guide"** (installation.pdf).
- For information about TRACE32 configuration, please refer to **"Training Basic SMP Debugging"** (training_debugger_smp.pdf).

Supported Architectures

The TRACE32 GDB stub is supported for the following architectures:

- 68K/ColdFire
- 8051/XC800/M51
- Arm (32 and 64 bit)
- GTM
- Intel x86/x86 64
- MicroBlaze
- MIPS32/MIPS64
- PowerArchitecture (32 and 64 bit)
- RISC-V
- SuperH
- TriCore
- V850/RH850
- Xtensa

Other architectures could be supported on demand. Please submit your request at **support.lauterbach.com/new-ticket**.

Operation Theory and Restrictions

TRACE32 PowerView receives commands from the remote GDB front-end, executes them and sends the response back to the front-end.

The Communication to TRACE32 PowerView is implemented as a socket interface. This means, that the GDB front-end and TRACE32 PowerView can reside on two different hosts, using network connections for the communication. TRACE32 PowerView routes the GDB requests to the TRACE32 hardware. The answer to the request goes exactly the opposite way, returning information to the GDB front-end.

The GDB requests are operated just beside normal TRACE32 operation. You can use both, TRACE32 PowerView and the GDB front-end interchangeable, although it is not recommended. The GDB front-end won't be informed about changes that are done in TRACE32 PowerView.

Due to some limitations of the Remote Serial Protocol, extended TRACE32 functionality like trace views, MMU views and OS-aware debugging is not supported.

TRACE32 Setup

To configure TRACE32 PowerView as GDB back-end, use the command **SETUP.API.GDB.Enable**. You can specify a port number using the **/PORT** option. If no port number is specified, the default port 30000 will be used.

Example 1:

```
SETUP.API.GDB.Enable                        ; enable GDB server in TRACE32
                                            ; using default port 30000
```

Example 2:

```
SETUP.API.GDB.Enable /PORT 12345.           ; enable GDB server in TRACE32
                                            ; using port 12345
```

GDB Front-Ends Setup

TRACE32 PowerView could be controlled by any front-end that supports the GDB Remote Serial Protocol. This section presents some examples of third party tools that could be configured to control TRACE32 PowerView via its GDB stub.

The GNU Project Debugger GDB

In the following example **gdb-multiarch** is used. The other alternative, is to use the appropriate **gdb** for the selected architecture:

```
$gdb-multiarch GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
```

1. Make sure that you set the correct architecture. In this example, the architecture **arm** is used to control TRACE32 PowerView for Arm:

   ```
   (gdb) set architecture arm
   The target architecture is assumed to be arm
   ```

2. Connect to TRACE32 PowerView using **TCP**. Here the **localhost** address **127.0.0.1** is used. If TRACE32 PowerView is started on a different machine than the one running **gdb**, the remote machine IP address needs to be used. At this level, there are two different way to start the GDB session. This depends on the status of the communication between TRACE32 debugger and the target:

   - If the debug communication between TRACE32 and the target is already established. TRACE32 PowerView should show for example **"system ready"** in the status bar. Then, the command **target remote** needs to be used.

     ```
     (gdb) target remote 127.0.0.1:30000
     Remote debugging using 127.0.0.1:30000
     0x000011fc in ?? ()
     ```

   - If the debug communication between TRACE32 and the target is not yet established. TRACE32 PowerView should show for example **"system down"** in the status bar. Then, the command **target extended-remote** needs to be used. In the following example, the monitor commands are used to initiate the communication between TRACE32 and the target. For more details please see **Monitor Commands**.

     ```
     (gdb) target extended-remote 127.0.0.1:30000
     Remote debugging using 127.0.0.1:30000
     (gdb) monitor B::SYStem.CPU CortexA9
     (gdb) monitor B::SYStem.Up
     ```

3. Now you can continue debugging using the GDB command line. E.g:

   - Set the symbol file relative to the application loaded into the target memory

```
(gdb) symbol-file sieve.elf
Reading symbols from sieve.elf...done.
```

   - Set a breakpoint

```
(gdb) break sieve
Breakpoint 1 at 0x15e6: file ./src/sieve.c, line 798.
```

   - Continue the program execution

```
(gdb) continue
Continuing.
Breakpoint 1, sieve () at ./src/sieve.c:798
798    count = 0;
```

Monitor Commands

The TRACE32 GDB stub supports the following monitor commands:

help	The command “monitor help” returns the string “TRACE32”
B::<*cmd*>	Execute a TRACE32 PRACTICE command. Example: “monitor B::Var.set mcount=1”
eval <*func*>	Get the return value of a PRACTICE function. Example: “monitor eval Register(PC)”
Practice-State	The command “monitor Practice-State” returns the run-state of PRACTICE. 0: not running 1: running 2: dialog window open
set step-mode [hll | asm]	Execute an assembly or high-level single step when getting a GDB RSP step packet. Default is asm.
sync [on | off]	Enable / disable the software component that allows a TRACE32 instance to connect to other instances by executing the TRACE32 command **SYnch.ON** or **SYnch.OFF**.

default-reset [core | system]	**core**: execute the command **SYStem.Option.EnReset OFF**. **system**: execute the command **SYStem.Option.EnReset ON**.
reset debug	Execute a **SYStem.Mode Up**.

Examples:

```
(gdb) monitor help
TRACE32
(gdb) monitor eval Var.Value(mcount)
00000792
(gdb) monitor B::Data.Set 0x100 %Long 0
(gdb) monitor eval Var.Value(mcount)
00000000
(gdb) monitor B::QUIT
Remote connection closed
```

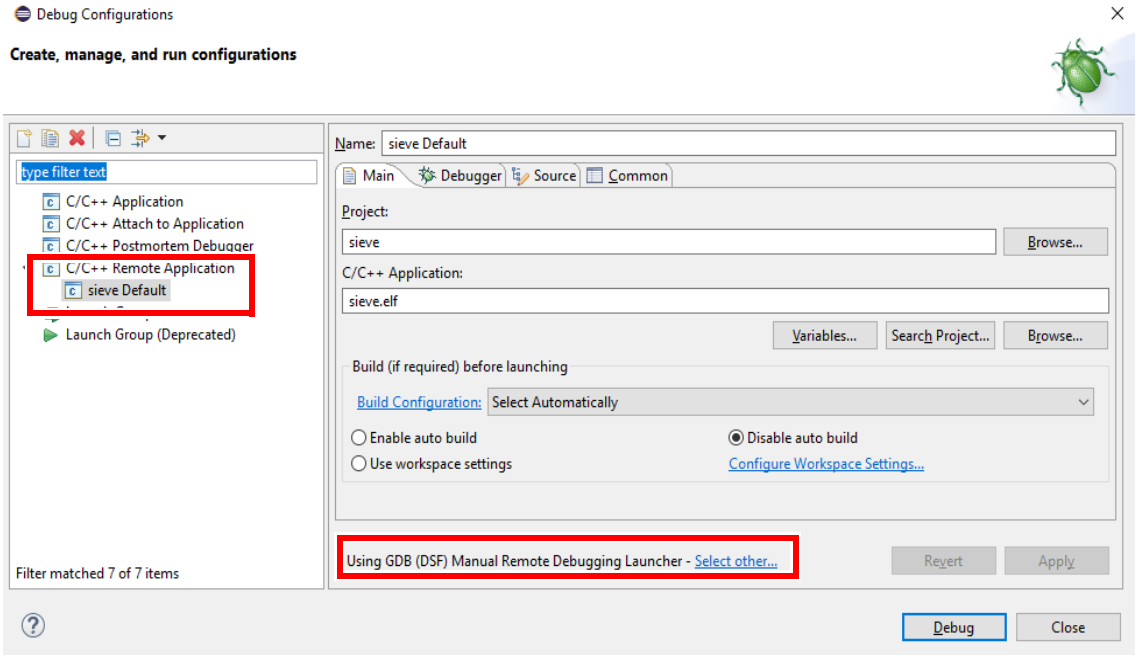
Eclipse

It is possible to control TRACE32 PowerView via its GDB stub from Eclipse. However, it is recommended to use the Target Communication Framework (TCF) instead, since it offers more features. For more details, please refer to **"TRACE32 as TCF Agent"** (app_tcf_setup.pdf).

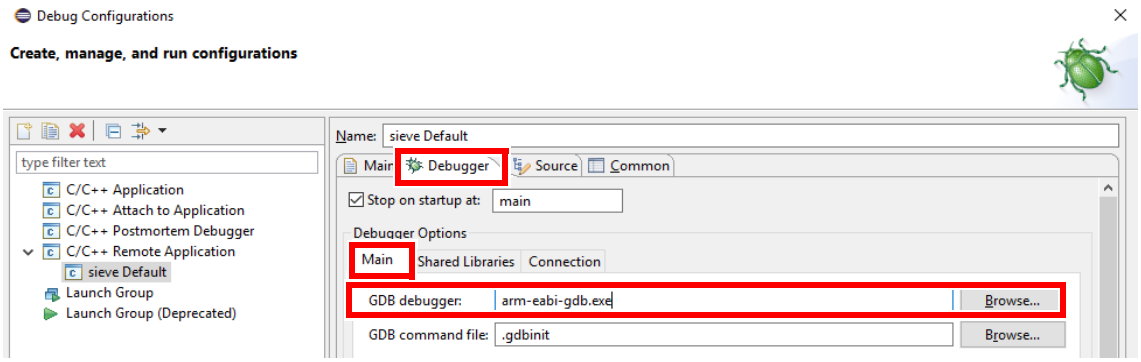
The following screenshots are generated using Eclipse IDE for C/C++ Developers, Oxygen.2 Release (4.7.2).

You need to configure Eclipse as follows:

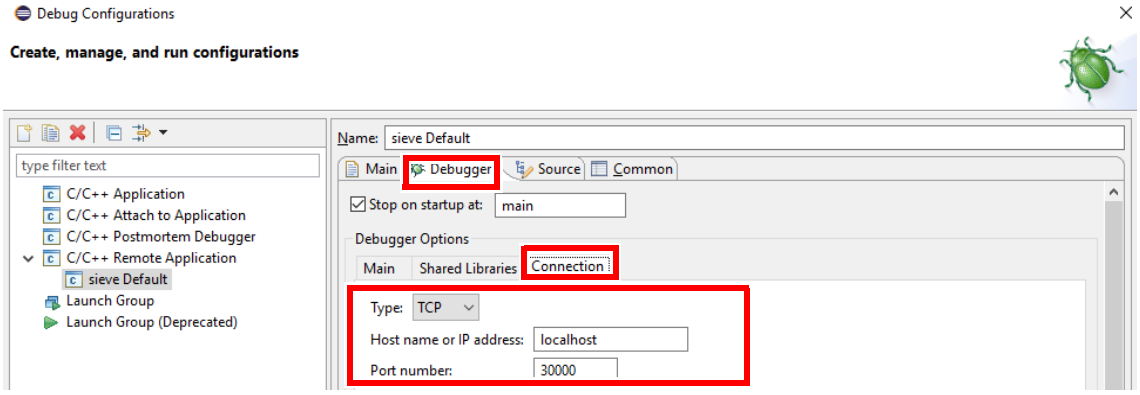
1. Add a debug configuration for remote application to your Eclipse work space. In the **Debug Configurations** window, you need to select the configuration **"GDB (DSF) manual Remote Debugging Launcher".**



2. In the **Debugger** tab, select your **GDB debugger** according to your target architecture.



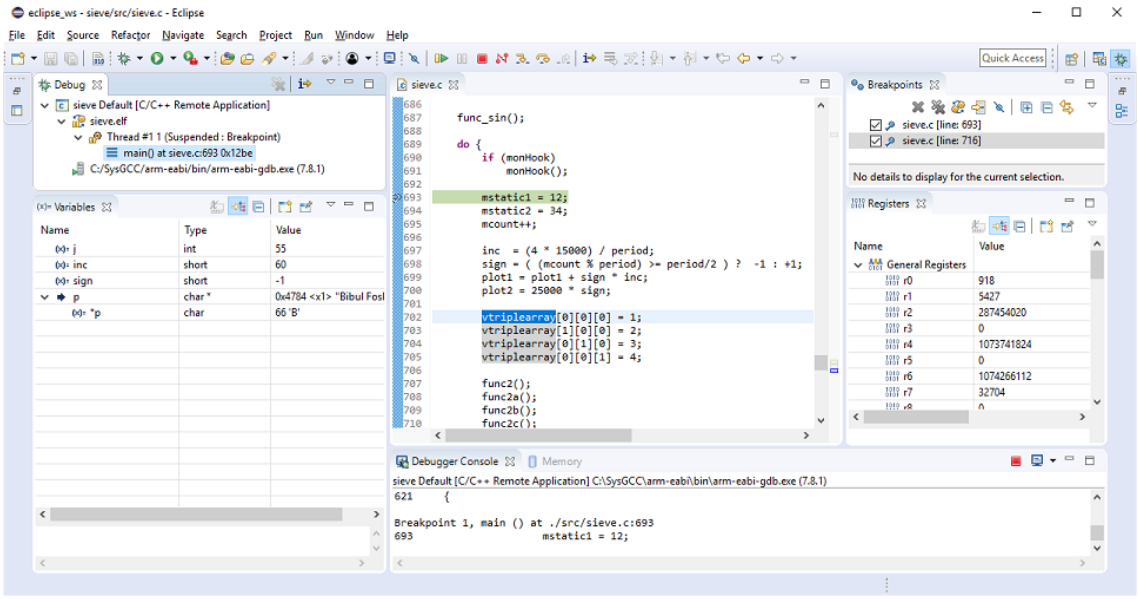
3. Set up the connection configuration according to your TRACE32 PowerView setup.



- Set the **Host name or IP address** of the machine running the TRACE32 PowerView instance.

- Set the **Port number** according to the GDB port configuration of the PowerView instance.

4. Before you try to start debugging from Eclipse, you need to make sure that the debug communication between TRACE32 and the target is already established. TRACE32 PowerView should show for example **“system ready”** in the status bar.

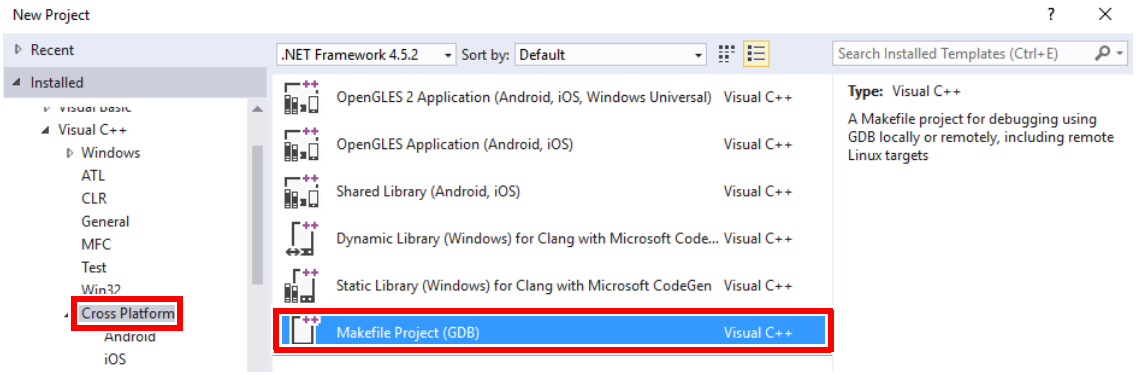


Starting from Microsoft Visual Studio 2015, it is possible to install the GDB Debugger extension for Visual C++. This introduces a new project type, that provides additional project properties and allows to connect the Visual Studio debugger to a GDB stub.

The following screenshots are based on Microsoft Visual Studio Professional 2015.

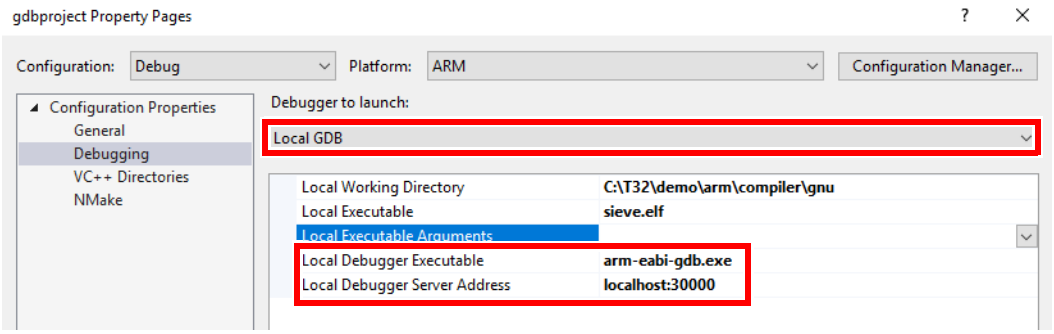
You need to configure Visual Studio as follows:

1. Create a new project by selecting the template **"Makefile Project (GDB)"**.

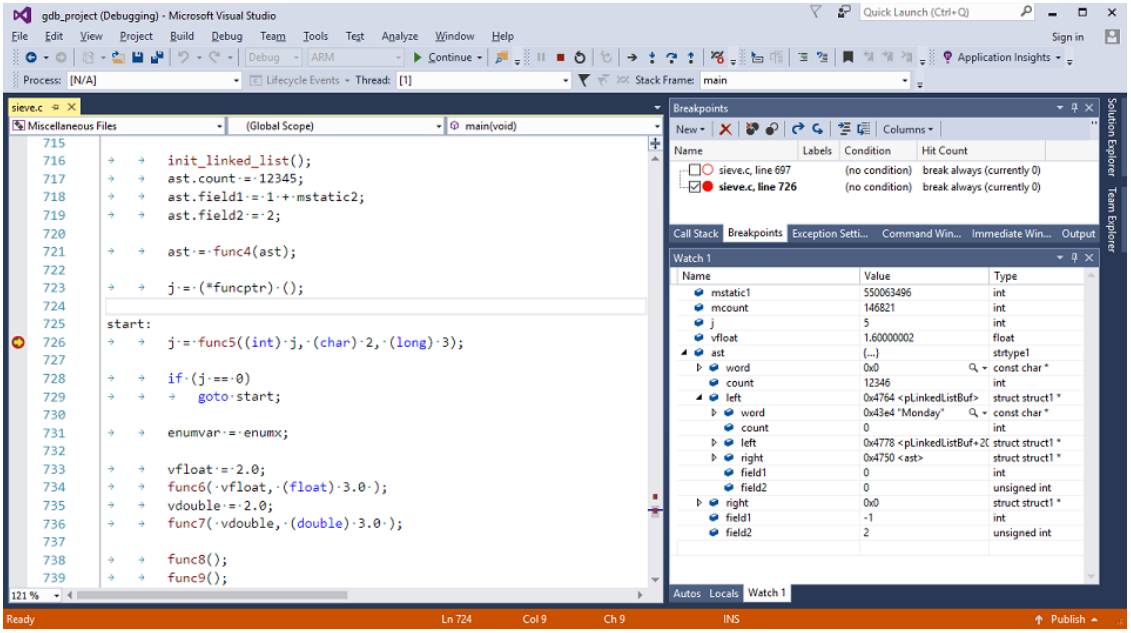


2. Set up the debugging configuration on the project properties page:

   - Chose the item **Local GDB**.
   - Set up the correct GDB debugger executable according to your target architecture. In this example, a configuration for Arm architecture is used.
   - Configure the **Local Debugger Server Address** with the IP address or host name of the machine, running the TRACE32 PowerView instance, and the GDB port number of the TRACE32 PowerView instance.



3. Make sure that the debug communication between TRACE32 and the target is already established. The TRACE32 PowerView should show for example “system ready” in the status bar.



You need to configure Visual Studio Code as follows to establish a connection with TRACE32 PowerView:

1. Select the menu **Run** > **Add Configuration** then add an empty configuration **{}**.
2. Set the following parameters:

"name"	Select the name you want to give to your configuration
"type"	"gdb"
"request"	"launch"
"target"	ELF file
"gdbpath"	GDB executable
"cwd"	Current working directory
"valuesFormatting"	"parseText"
"autorun"	"target remote localhost:30000", "symbol-file < ELF file >",

Example:

```
{
    "configurations": [
        {
            "name": "TRACE32 Debug",
            "type": "gdb",
            "request": "launch",
            // set the path to the project's ELF file:
            "target":"C:/T32/demo/arm/compiler/gnu/sieve.elf",
            // set the path to the gdb executable on the host:
            "gdbpath": "C:/ARMGCC/bin/arm-none-eabi-gdb.exe",
            "cwd": "C:/T32/demo/arm/compiler/gnu/",
            "valuesFormatting": "parseText",
            "autorun": [
                // connect to TRACE32 PowerView listening to port
                // number 3000
                "target remote localhost:30000",
                // load the debug symbols in VSCode
               "symbol-file C:/T32/demo/arm/compiler/gnu/sieve.elf",
            ]
        }
    ]
}
```

3. Start TRACE32 PowerView with a startup script in order to establish connection with the target. Example:

```
C:\T32\bin\windows64\t32marm.exe -s \
                              C:\T32\demo\arm\compiler\gnu\demo_sram.cmm
```

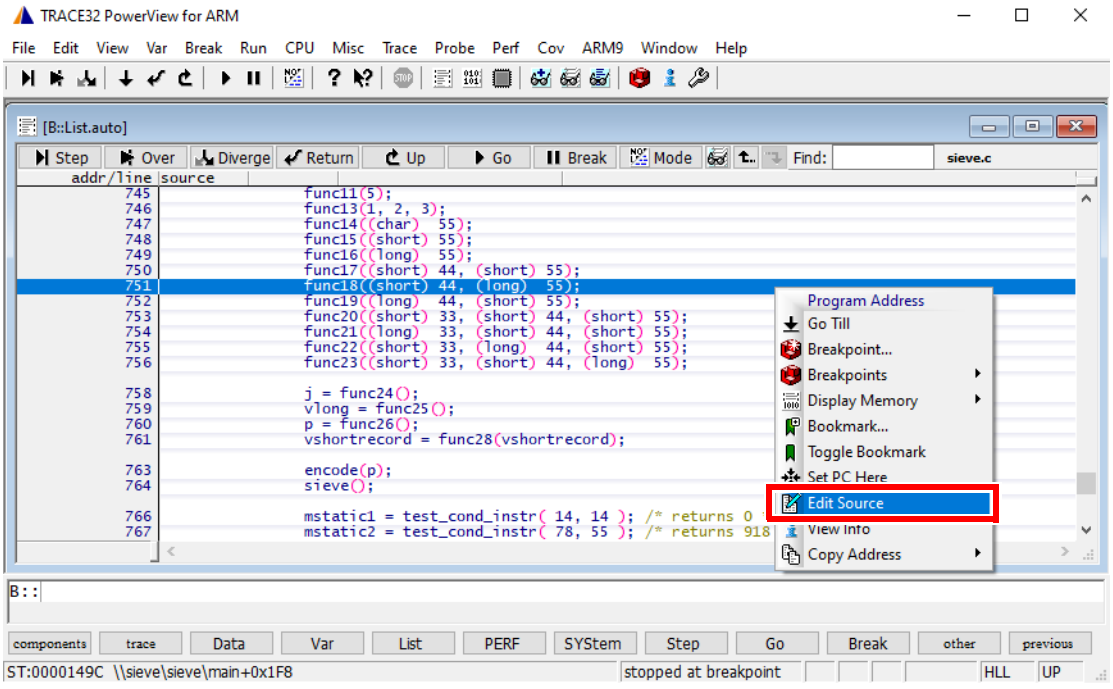
4. To start debugging, select the menu **Run > Start Debugging** or press **F5**.

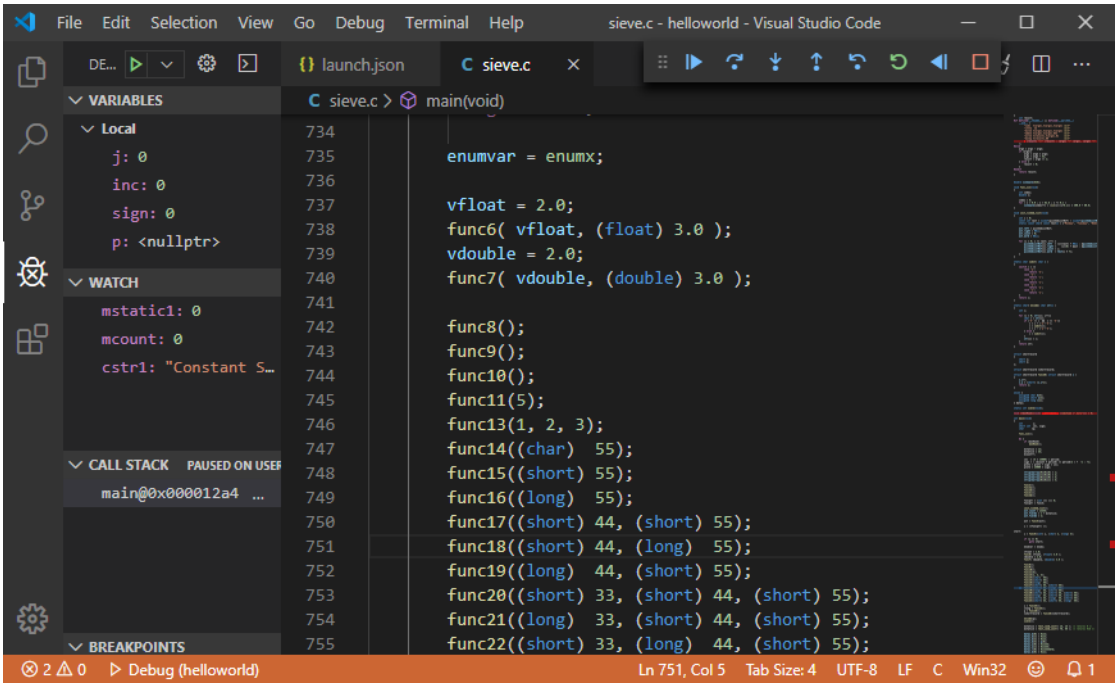
NOTE:	VSCode may display a red message “Exception has occurred” at the program counter position and the message “Not implemented stop reason (assuming exception): undefined” in the debug console. This message can be ignored.

You can additionally replace the TRACE32 built-in editor call with a call to the Visual Studio Code editor using the command **SETUP.EDITEXT**:

```
SETUP.EDITEXT ON "<path_to_vscode>\Code.exe -g ""*:#"""
```

If you do the a right mouse click in the TRACE32 **List** window then select Edit Source, the file will be displayed in Visual Studio Code on the selected line number.



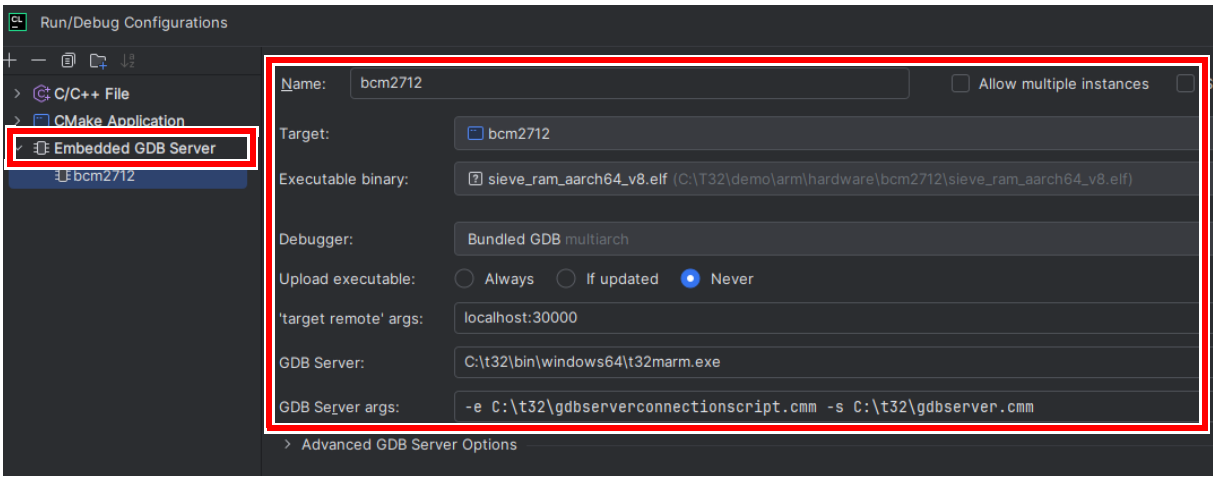


JetBrains CLion

This chapter presents the setup of JetBrains CLion as a front-end for controlling TRACE32 through its built-in GDB stub interface. In this configuration, CLion manages program execution while TRACE32 operates as the debugging back-end.

Use the following configuration steps to establish this connection.

1. Begin by creating a new **Embedded GDB Server** configuration in CLion.



2. Set up the debug configuration in the project properties:

   - Define a name for the debug configuration.
   - Specify the target.
   - Optionally, select the binary file to be loaded in **Executable binary**.
   - Choose the **GDB Debugger** instance to run on the CLion side.
     Example: Bundled GDB
   - Configure the **Target remote** using the GDB port number that matches the one defined in TRACE32 PowerView. Refer to **"TRACE32 Setup"**, page 6 for more information.
   - Specify the correct **GDB server** executable provided by TRACE32, based on your target architecture. In this example, an executable for Arm architecture is used.
   - Configure the **GDB Server Arguments** by providing the command line to start TRACE32 PowerView, including -e option to specify the connection script and -s option for the start up script. For details on available options, refer to **"TRACE32 Installation Guide"** (installation.pdf).
     The startup script defines and initializes the target configuration, it should select the CPU and use **SYStem.Mode Up** command to attach, then activate the GDB API.

```
DO init.cmm
...
SETUP.API.GDB.Enable
```

3. Click Debug to start the debug session. CLion will launch TRACE32 with the GDB stub enabled and connect to it, allowing full use of GDB-based features like run, breakpoints, and step.

Remote Serial Protocol

TRACE32 GDB stub supports almost all the packets defined by the Remote Serial Protocol except some packets that are generally not needed to control TRACE32 PowerView via the GDB stub.If you need the exhaustive list of the supported packets please submit a request at **support.lauterbach.com/new-ticket**.

Protocol Extensions

The Remote Serial Protocol does not provide a way to distinguish between different memory types. When the RSP protocol is used to communicate with a GDB stub other than the GNU gdbserver, the memory address is not always sufficient to identify a unique physical memory location. Depending on the access mode, the same memory address could refer to different physical memory locations (e.g. secure/non secure memory for Arm architecture). To overcome these limitations, Lauterbach has defined a protocol extension:

- A packet to read *<length>* addressable memory of type defined by *<access_class>* starting at address *<address>*.

```
qtrace32.memory:<access_class>,<address>,<length>
```

- A packet to write *<length>* addressable memory of type defined by *<access_class>* starting at address *<address>*. The data is given by *<values>*; each byte is transmitted as a two-digit hexadecimal number.

```
Qtrace32.memory:<access_class>,<address>,<length>,<values>
```

- If the TRACE32 software version implements this protocol extension, it should include the string **"qtrace32.memory+;Qtrace32.memory+"** in the reply to the **"qSupported"** packet.

The available access classes are dependent on the processor architecture in use. Therefore refer to the Access Class/Memory Class section of your Processor Architecture Manual for more details.

Symmetrical Multiprocessing Support

In order to support Symmetrical Multi-Processing (SMP) debugging over the GDB interface, TRACE32 considers each core from an SMP system as a thread. Thus, all the RSP packets relative to the multi-thread handling are used for multi-core handling.