LAUTERBACH
DEVELOPMENT TOOLS

Run Mode Debugging Manual Linux

Release 02.2025

MANUAL

Run Mode Debugging Manual Linux

Run Mode Debugging Manual Linux

Version 13-Feb-2025

Debugging Modes for Embedded Linux

TRACE32 provides 3 modes for debugging embedded Linux:

- Run Mode Debugging
- Stop Mode Debugging
- Integrated Run & Stop Mode Debugging.

Run Mode Debugging with TRACE32 as GDB Front-end

The TRACE32 GDB Front-end is a pure software debugger i.e. no TRACE32 hardware is required. The TRACE32 software is licensed in this case with a floating license via RLM (Reprise License Manager). a `gdbserver` or `gdbstub` has to be running on the target.

When debugging a Linux process using a `gdbserver`, the TRACE32 GDB Front-end works in **Run Mode debugging**: at a breakpoint only the selected process is stopped, while the kernel and all other processes continue to run. When debugging a virtual target (e.g. QEMU), the TRACE32 GDB Font-end operates however in **Stop Mode**.

Please refer for more information about the TRACE32 GDB Front-end to the document **"TRACE32 as GDB Front-End"** (frontend_gdb.pdf). The document also includes a list of processor architectures supported by the TRACE32 GDB Front-end.

Stop Mode Debugging

When debugging in Stop Mode, the whole system is halted at a breakpoint and not a single process. This is e.g. the case when debugging via JTAG.

The main advantages of Stop Mode debugging are:

- Debugging can start at the reset vector.
- Debugging of the kernel and beyond process boundaries is possible.

Stop Mode debugging is described in the document **"OS Awareness Manual Linux"** (rtos_linux_stop.pdf).

Integrated Run & Stop Mode Debugging via JTAG

Integrated Run & Stop Mode debugging requires a TRACE32 JTAG debugger hardware.

If debugging is performed via the JTAG interface, TRACE32 can be configured:

- To allow Stop Mode debugging via JTAG.
- To allow Run Mode debugging via the `t32server` or `gdbserver` running as debug agent on the target.

Switching between both modes is also possible.

TRACE32 communicates with:

- The `gdbserver` via Ethernet for all supported architectures or
- The `t32server` via DCC (Debug Communication Channel) for the **ARM** architecture.

Integrated Run & Stop Mode debugging is supported for the following architectures:

- ARM over Ethernet or DCC
- Intel x86 via Ethernet
- MIPS32 via Ethernet
- PowerPC via Ethernet
- SH4 via Ethernet

For Integrated Run & Stop Mode debugging, Stop Mode debugging via the JTAG interface is extended by:

- `t32server` or `gdbserver` as debug agent on the target.
- A communication interface between TRACE32 and the debug agent (Ethernet or DCC).

Ethernet as Communication Interface to the gdbserver

TRACE32 communicates in this case with the `gdbserver` via Ethernet using the GDB Remote Serial Protocol. The version 7.1 or newer of the `gdbserver` is recommended. This means that, additionally to JTAG, an Ethernet connection is needed between the target and the host PC where TRACE32 PowerView is executed.

The JTAG communication with the target should be established before switching to Run Mode. The following steps are then needed:

1. Start the `gdbserver` on the target. To allow multi-process debugging, the `gdbserver` has to be started in **Multi-process mode**, also called **target extended-remote mode**. The `--multi` command line option has to be used:

   ```
   gdbserver --multi :2345
   ```

2. Inform TRACE32 about the IP address of the target and the port number used by the `gdbserver` using the command **SYStem.PORT** e.g.

   ```
   SYStem.PORT 10.1.2.99:2345
   ```

3. Switch to Run Mode using the command **Go.MONitor**.

After the communication is configured, debugging can be performed completely via the TRACE32 PowerView user interface.

DCC as Communication Interface to the t32server

The JTAG interface of the ARM architecture includes a Debug Communication Channel (DCC). Information exchange via DCC is possible between TRACE32 and a target application.

The `t32server` is a Linux application provided by Lauterbach that can be used as an extension to the `gdbserver`. Compared to the `gdbserver` the `t32server` allows debugging over DCC for the ARM architecture. The `t32server` starts a `gdbserver` for debugging. The `gdbserver` has to be in the `/bin` directory of the Linux file system. The `t32server` communicates with the `gdbserver` via localhost (TCP/IP).

The source code of the `t32server` is available in the TRACE32 installation under `~~/demo/arm/etc/t32server`.

In order to provide Integrated Run & Stop Mode debugging the `t32server` has to be started as a Linux process on the target via the terminal window e.g.:

```
./t32server                              ; Communication via DCC
```

On an SMP system, the debugger only communicates with the DCC registers of the first core. Thus, the `t32server` should always run on this core. For details about DCC refer to your **ARM Technical Reference Manual**.

After TRACE32 was started and configured for Stop Mode debugging switching to Run Mode is performed as follows:

```
SYStem.MemAccess GdbMON

Go.MONitor
```

SYStem.MemAccess GdbMON	Configure DCC as communcation interface to t32server
Go.MONitor	Switch to Run Mode Debugging

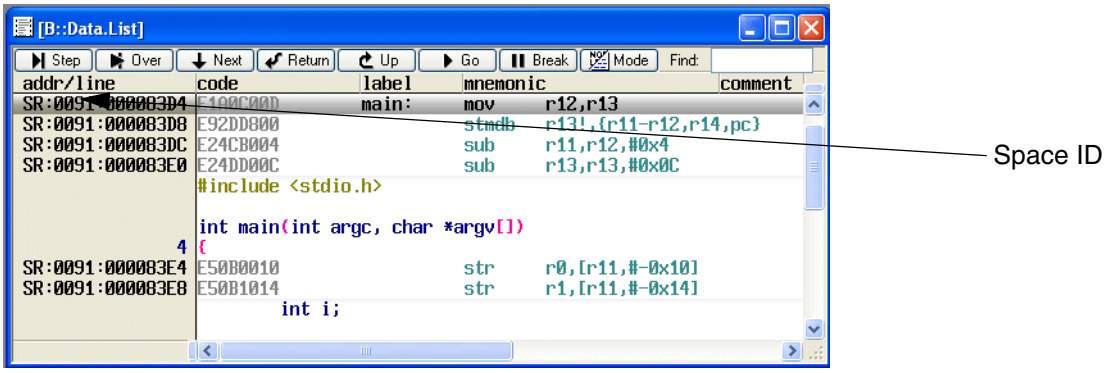
After the communication is configured, debugging can be performed completely via the TRACE32 PowerView user interface.

The Space ID for Run Mode Debugging

Processes of Linux may reside virtually on the same addresses. To distinguish those addresses, the debugger uses an additional identifier called **space ID** that specifies to which virtual memory space an address refers. In Run Mode debugging the space ID is equal to the process ID.

The command **SYStem.Option.MMUSPACES ON** enables the additional space ID in TRACE32.

A source code listing for the process `sieve` is displayed as follows:



For details on the space ID for Stop Mode Debugging, refer to **“Training Linux Debugging”** (training_rtos_linux.pdf).

1. Start the `t32server` or the `gdbserver` in multi-process mode.
2. Switch to Run Mode debugging as previously described.
3. Check if the process is already running.

TASK.List.tasks	List all running processes

```
TASK.List.tasks
```

4. Load the process for debugging.

TASK.RUN *<process>*	Load *<process>* (process not running)
TASK.select *<id>*	Attach to the process (process already running)

If the process is not running, the command **TASK.RUN** can be used to load the process for debugging.

```
; Load process sieve from the Linux file system and prepare it for
; debugging

TASK.RUN /bin/sieve
```

If the process is already running, the command **TASK.select** can be used to attach to it.

```
TASK.select /bin/sieve
```

5. Load the symbol and debug information for the process.

Data.LOAD.*<file_format> <file> <space_id>***:0 /NOCODE /NoClear**

Since processes of Linux may reside virtually on the same addresses, the symbol and debug information has to be loaded for the address space of the process by using the *<space_id>*.

/NOCODE - load only symbol information.

/NoClear - obtain the symbol information loaded for other processes.

```
Data.LOAD.Elf sieve.elf 0x91:0 /NOCODE /NoClear

; Stop sieve at main and display source listing
Go main
List
```

TASK.PROC.SPACEID(*<process>***)**	This function returns the *<space_id>* of a process. This is required for PRACTICE scripts.

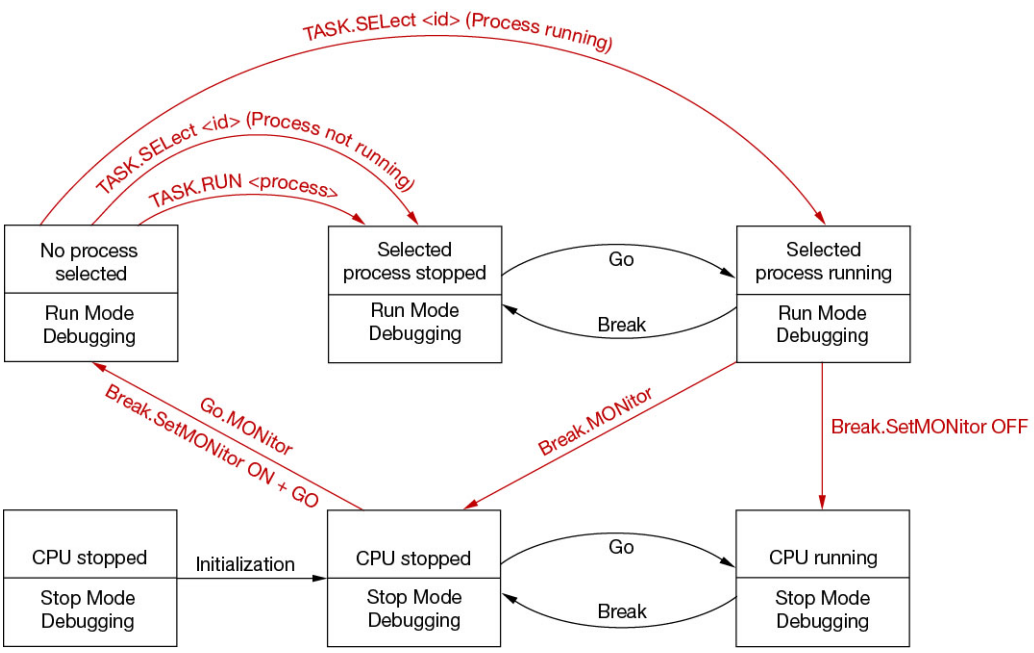
Example for a PRACTICE script:

```
LOCAL &sid
&sid=TASK.PROC.SPACEID("sieve")

TASK.RUN /bin/sieve

Data.LOAD.Elf sieve.elf &sid:0 /NOCODE /NoClear /NOREG
```

Switching between Run & Stop Mode Debugging



The graphic above shows a simple schema of the switching between Run Mode and Stop Mode debugging. Not all transitions are covered.

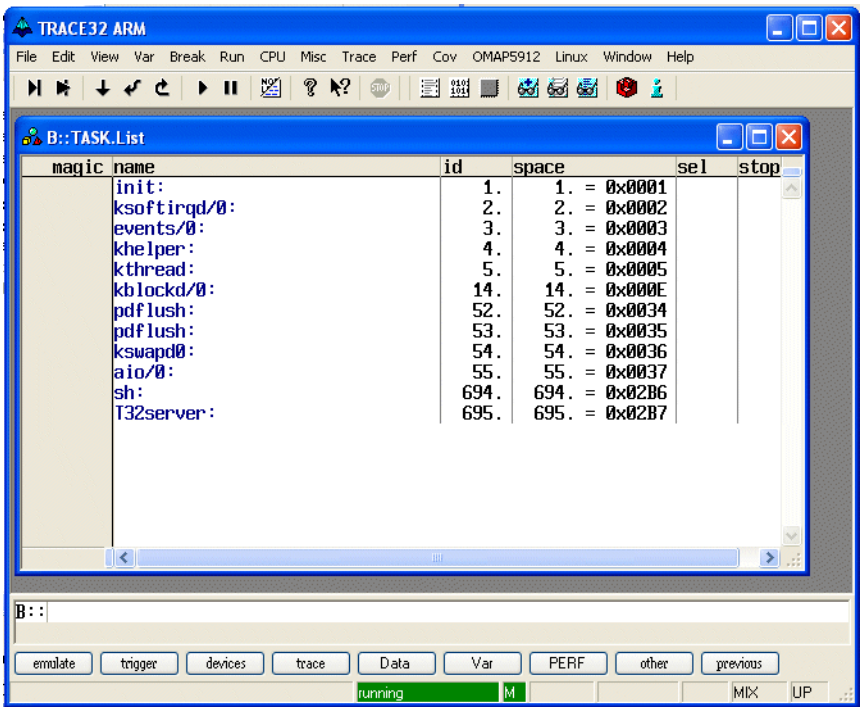
The following commands are used to switch between Run & Stop Mode Debugging:

Go.MONitor	If the CPU is stopped, the program execution is started. Switch to Run Mode debugging. In Run Mode debugging no process is selected for debugging.
Break.MONitor	Switch to Stop Mode debugging and stop the program execution on CPU.
Break.SetMONitor ON	Switch to Run Mode debugging with the next **Go**. In Run Mode debugging no process is selected for debugging.
Break.SetMONitor OFF	Switch to Stop Mode debugging. If the selected process was running or no process was selected, the CPU stays running in Stop Mode. If the selected process was stopped, the CPU is stopped in Stop Mode.

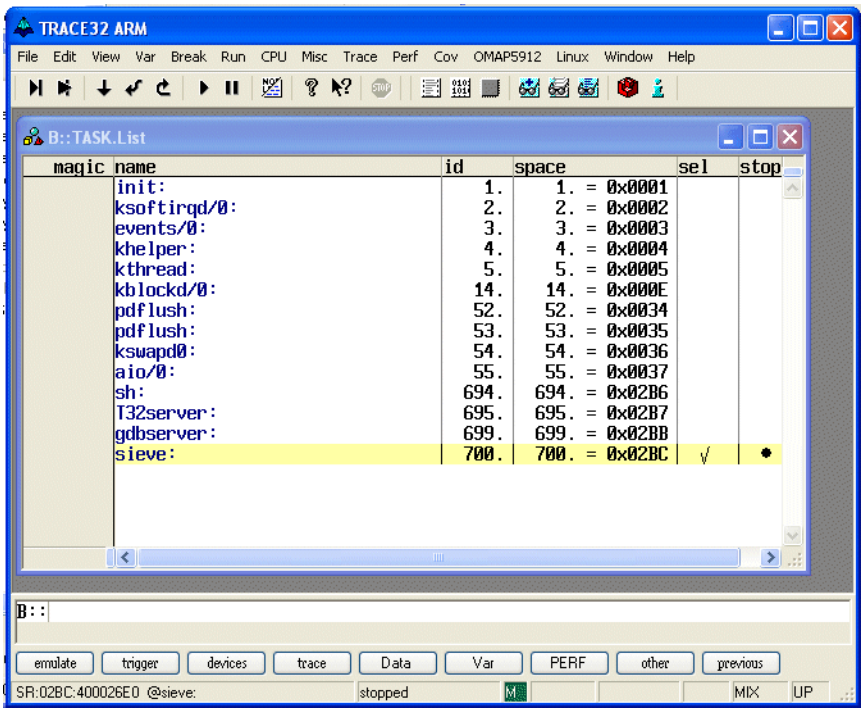
If Run Mode Debugging is active, a **green M** is displayed in the state line of TRACE32 PowerView.

The following states are possible in Run Mode Debugging:

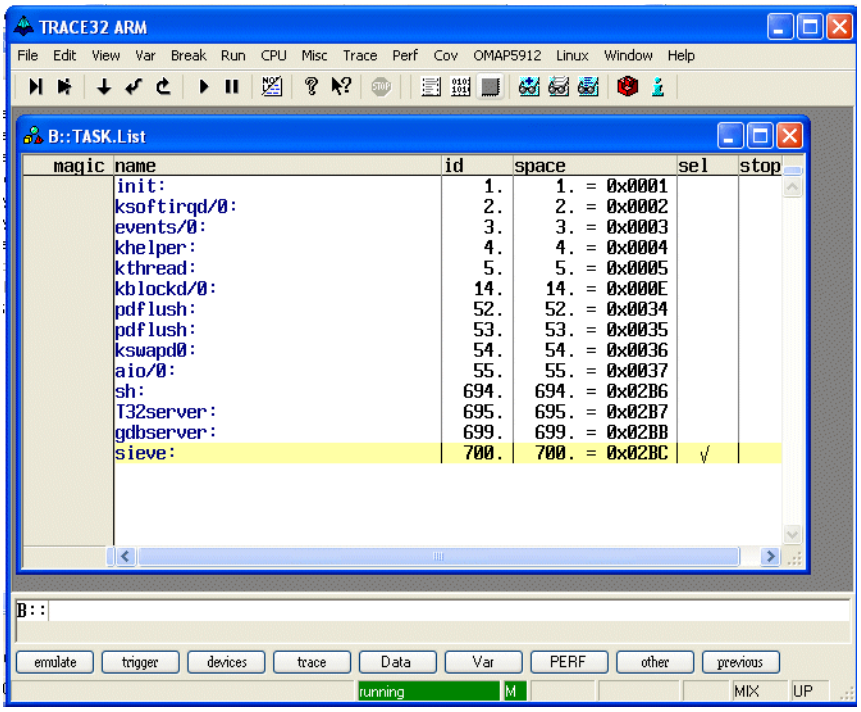
1. Run Mode debugging active (**green M**), no process selected (see **TASK.List.tasks**).



2. Run Mode debugging active (**green M**), selected process (sieve) stopped.



3. Run Mode debugging active (**green M**), selected process (sieve) running.



TASK.List.tasks	List all running processes
TASK.RUN *<process>*	Load a process for debugging

If the command **TASK.RUN** is used to load a process for debugging, the process is stopped by the `gdbserver` at its entry point.

To start process debugging, load the symbol information for the process first and then type `Go main`.

TASK.select *<id>*	Select a process for debugging

If the selected process has been started with **TASK.RUN**, it will be selected as current process. Otherwise a `gdbserver` will be started and the selected process will be attached.

TASK.KILL *<id>*	Request GDB agent to end the process

Only processes that have been started with a **TASK.RUN** or that have been attached with **TASK.select** can be killed.

TASK.COPYUP *<src> <dest>*	Copy a file from the target into the host
TASK.COPYDOWN *<src> <dest>*	Copy a file from the host into the target

Breakpoint Conventions

For Integrated Run & Stop Mode debugging please keep the following breakpoint convention:

- Use on-chip breakpoints for Stop Mode debugging

  If an on-chip breakpoint is hit in Run Mode debugging, the CPU is stopped in Stop Mode debugging (only for ARM).

- Use software breakpoints for Run Mode debugging

Examples for Stop Mode debugging:

```
; Break.Set <space_id>:<address> /Program /Onchip
Break.Set 0x0:0x4578 /Program /Onchip

Break.Set error /Program /Onchip
```

Examples for Run Mode debugging:

```
; Break.Set <space_id>:<address> /Program /SOFT
Break.Set 0x2bc:0x0xd0065789 /Program /SOFT

Break.Set 0x2bc:main /Program /SOFT
```