



TRACE32 as GDB Front-End

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents		
GDB Support		
TRACE32 as GDB Front-End	1	
History	4	
Introduction	5	
Documentation Updates	5	
Related Documents	6	
Supported Architectures	6	
TRACE32 Setup	7	
Configuration File	7	
T32Start	7	
Connection Setup	8	
Debugging Virtual Targets	8	
Example: Connecting to QEMU	8	
Protocol Extensions	9	
GNU GDBserver	10	
Connection Setup	11	
Multi-Process Debugging	14	
UndoDB Reversible Debugger	15	
KGDB	16	
Troubleshooting	17	
GDB Front-End SYStem Commands	18	
SYStem.CPU	Select target CPU	18
SYStem.Mode	Establish communication to debug agent	18
SYStem.Option.IMASKASM	Disable interrupts while single stepping	19
SYStem.Option.IMASKHLL	Disable interrupts while HLL single stepping	19
SYStem.Option.MMUSPACES	Separate address spaces by space IDs	19
SYStem.Option.OVERLAY	Enable overlay support	20
SYStem.RESetOut	Reset target	20
SYStem.GDBconfig.BREAKSOFT	Use software breakpoint	20
SYStem.GDBconfig.EXTENDED	Enable/disable gdb extended mode	21
SYStem.GDBconfig.GDBSERVER	Remote target is a gdbserver	21
SYStem.GDBconfig.INFERIORID	Set inferior ID	21

SYStem.GDBconfig.MONITOR	Send monitor command to GDB Back-End	22
SYStem.GDBconfig.NONSTOP	Enable/disable non-stop mode	22
SYStem.PORT	Set communication settings	22
SYStem.GDBSIGnal	Define signal handling	23
GDB Front-End TASK Commands		24

History

- 15-Jan-19 Revised manual.
- 06-May-14 Manual was renamed. The old name was monitor_gdb.pdf.

Introduction

The TRACE32 GDB Front-End is a software debugger solution which communicates via Ethernet or RS232 with a gdbserver /gdbstub using the GDB Remote Serial Protocol (RSP).

The TRACE32 GDB Front-End can be used:

- To connect to the GNU gdbserver in order to debug Linux user-space processes.
- To debug the Linux kernel via KGDB.
- To connect to any kind of virtual target or debugger implementing a gdbstub (e.g. QEMU)
- As a front end for the UndoDB reversible debugger

The TRACE32 GDB Front-End operates in two different modes: **Run Mode** and **Stop Mode**.

Run Mode is used when debugging a Linux user-space process using the GNU gdbserver or the UndoDB target server. In this case, only the selected task is stopped when a breakpoint is hit. The kernel and all other processes continue to run. In Run Mode, a single program may have more than one unit of execution, called threads, that share one memory address space. Each thread has its own registers and execution stack. The debugger gets and displays the information about the running tasks from the gdbserver by using the dedicated Remote Serial Protocol packets.

On the other hand, multiple virtual targets (e.g. QEMU) support the GDB Remote Serial Protocol as a debugging protocol. In this case, we talk about Stop Mode debugging since the TRACE32 GDB Front-End controls the whole target system and not a single process. A breakpoint will thus cause the target system to stop completely. In order to support Symmetrical Multi-Processing (SMP) debugging over the GDB interface, the TRACE32 GDB Front-End considers each core from an SMP system as a thread of execution. Thus, all the RSP packets relative to the multi-thread handling are used for multi-core handling. Moreover, some virtual targets support different core clusters which are considered as GDB *inferiors*. The GDB packets for multi-process handling are used for this purpose.

NOTE:	Demo scripts for the TRACE32 GDB Front-End are available in the TRACE32 installation directory under <code>~/demo/etc/gdb</code>
--------------	--

Documentation Updates

The latest version of this document is available for download from:

www.lauterbach.com/pdf/frontend_gdb.pdf

Related Documents

- For information about using TRACE32 PowerView as a GDB Back-End, please refer to [“TRACE32 as GDB Back-End”](#) (backend_gdb.pdf).
- For information about Linux Integrated Run and Stop Mode Debugging, please refer to [“Run Mode Debugging Manual Linux”](#) (rtos_linux_run.pdf).
- For information about debugging virtual targets via interfaces other than GDB (e.g. MCD or CADI) please refer to [“Virtual Targets User’s Guide”](#) (virtual_targets.pdf).

Supported Architectures

The TRACE32 GDB Front-End is available for the following architectures:

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

Other architectures can be supported on demand. Please send your request to support@lauterbach.com

TRACE32 Setup

Configuration File

To configure TRACE32 as GDB Front-End, you need to add the following lines to your TRACE32 configuration file. The default configuration file is config.t32 and is located in the TRACE32 system directory.

```
PBI=GDB
```

<- mandatory empty line

<- mandatory empty line

Example configuration for Windows:

```
PBI=GDB

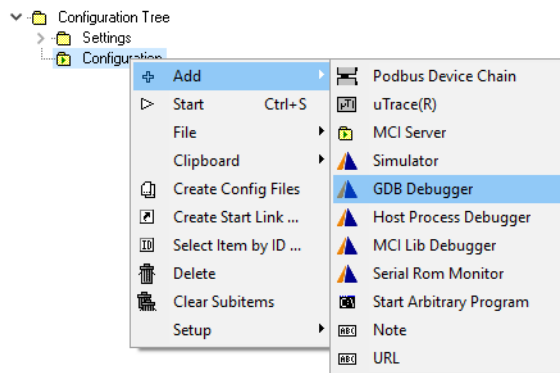
SYS=C:\T32
TMP=C:\Temp

SCREEN=
FONT=SMALL
```

For more information about the TRACE32 configuration, please refer to [“Training Basic Debugging”](#) (training_debugger.pdf).

T32Start

In case you are using the t32start utility to start the TRACE32 GDB Front-End, you need to add a **GDB Debugger** to your configuration.



Please refer to the [“T32Start”](#) (app_t32start.fm) manual for more information about the t32start utility.

Connection Setup

This chapter describes the needed configurations in order to establish a debug communication between the TRACE32 GDB Front-End and the following targets:

- Virtual targets with QEMU as example
- GNU GDBserver
- UndoDB reversible debugger
- KGDB

Debugging Virtual Targets

Multiple virtual targets includes a gdbstub and can thus be debugged using the TRACE32 GDB Front-End. We describe in the following the needed steps to establish a debug communication with QEMU.

Example: Connecting to QEMU

The following steps are required to establish a debug connection with the QEMU emulator.

1. Start QEMU using the `-gdb tcp::<port_number>` command line option or `-s` for the default port number 1234.
2. Start TRACE32 as GDB Front-End.
3. Select the target CPU from the **SYStem.state** window or using the **SYStem.CPU** command

```
SYStem.CPU ZYNQ-ULTRASCALE+-APU
```

4. Define the communication parameters in TRACE32 using the **SYStem.PORT** command.

Example for localhost and default port number 1234:

```
SYStem.PORT localhost:1234
```

5. Set the remote target type using the command **SYStem.GDBconfig GDBSERVER OFF**.

```
SYStem.GDBconfig GDBSERVER OFF
```

6. Establish the communication to the QEMU gdbstub using the **SYStem.Mode Attach** command

```
SYStem.Mode Attach
```


The Remote Serial Protocol does not provide a way to distinguish between different memory types. When the TRACE32 GDB Front-End is used in Stop Mode to debug a virtual target, 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 the following protocol extensions:

- 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 depend on the processor architecture in use. Therefore refer to the Access Class/Memory Class section of your [Processor Architecture Manual](#) for more details.

The gdbserver can be started in one of two different modes:

1. **Single-process mode**, also called **target remote mode**. In this case, the program to debug has to be specified on the gdbserver command line, or the `--attach` command line option has to be used to attach to a running process by specifying its PID.

Example 1: Start the gdbserver to debug the process `/bin/hello` from the start:

```
gdbserver :2345 /bin/hello
```

Example 2: Attach to the running process with PID 123

```
gdbserver --attach :2345 123
```

2. **Multi-process mode**, also called **target extended-remote mode**. In this case, the gdbserver can be started without supplying an initial command to run or PID to attach. The `--multi` command line option has to be used.

Example:

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

In multi-process mode, the GDB extended mode has to be enabled. This can be configured in TRACE32 using the command **SYStem.GDBconfig EXTENDED ON**.

Moreover, GDB supports two different modes for debugging multi-threaded processes: **Non-Stop Mode** and **All-Stop Mode**.

In All-Stop Mode, all threads of execution stop when the program stops. In Non-Stop Mode it is possible to stop single threads while other threads continue to execute. You can select the mode in TRACE32 using the command **SYStem.GDBconfig NONSTOP** which is per default set to **OFF**.

For more information about the different command line options and debug modes, please refer to the gdbserver documentation.

Connection Setup

The following steps are required to establish the communication with the gdbserver:

1. Select the target CPU from the **SYStem.state** window or using the **SYStem.CPU** command.

```
SYStem.CPU CortexA9
```

2. Enable the address extension in TRACE32 PowerView using the command **SYStem.Option.MMUSPACES ON**. **This step is only required if the gdbserver has been started in multi-process mode.**

```
SYStem.Option.MMUSPACES ON
```

3. Define the communication parameters in TRACE32 using the **SYStem.PORT** command.

Example for target IP address 192.168.188.50 and port number 2345:

```
SYStem.PORT 192.168.188.50:2345
```

4. Set the remote target type to gdbserver using the command **SYStem.GDBconfig.GDBSERVER ON**. If this command is not used, the TRACE32 GDB Front-End tries to detect what kind of remote target is used.

```
SYStem.GDBconfig.GDBSERVER ON
```

5. Use software breakpoints for assembly single stepping. This is required if Linux-3.x or newer is running on the target.

```
SYStem.Option.STEPSOFT ON
```

6. Enable the Non-Stop mode if it is required to stop single threads.

```
SYStem.GDBconfig.NONSTOP ON
```

7. **If the gdbserver has been started in multi-process mode**, the extended mode has to be enabled using the command **SYStem.GDBconfig.EXTENDED ON**, otherwise TRACE32 will return the error “the target is not running”.
8. Establish the communication to the gdbserver using the **SYStem.Mode Attach** command:

```
SYStem.Mode Attach
```

9. If the gdbserver has been started in multi-process mode, the commands **TASK.RUN** and **TASK.Attach** can be used to start a new process or attach to a running process.

Example:

```
; start the process hello
TASK.RUN /usr/bin/hello

; attach to the process sieve with PID 123
TASK.ATTACH 123.
```

10. Load the process debug symbols. In multi-process mode, the process space ID has to be specified.

```
; single-process mode
Data.LOAD.Elf sieve /NoCODE /NoClear

; multi-process mode (0x7b is the space ID of the process sieve)
Data.LOAD.Elf sieve 0x7b:0 /NoCODE /NoClear
```

Typical Start-up Script for the Single-Process Mode

```
; Reset all commands
RESet

; Clear all TRACE32 PowerView windows
WinCLEAR

; Select the target CPU
SYStem.CPU *

; Set the target IP address and port number
SYStem.PORT 192.168.188.50:2345

; Set the target type to "gdbserver"
SYStem.GDBconfig GDBSERVER ON

; Use software breakpoints for assembly single stepping
SYStem.Option.STEPSOFT ON

; Attach to the gdbserver
SYStem.Mode Attach

; Load the debug symbols of process "sieve"
Data.LOAD.Elf sieve /NoCODE /NoClear
```

```
; Reset all commands
RESet

; Clear all TRACE32 PowerView windows
WinCLEAR

; Select the target CPU
SYStem.CPU *

; Set the target IP address and port number
SYStem.PORT 192.168.188.50:2345

; Set the target type to "gdbserver"
SYStem.GDBconfig GDBSERVER ON

; Set the extended mode
SYStem.GDBconfig EXTENDED ON

; Use software breakpoints for assembly single stepping
SYStem.Option.STEPSOFT ON

; Attach to the gdbserver
SYStem.Mode Attach

; View the list of processes
TASK.List.tasks

; Start process sieve, wait 1.s and read its space ID
TASK.RUN /bin/sieve
WAIT 1.s
&spaceid=TASK.SPACEID("sieve")

; Load the debug symbols of process "sieve"
Data.LOAD.Elf hello &spaceid:0 /NoCODE /NoClear
```

NOTE:

The TRACE32 GDB Front-End for Intel x86 can be used together with the x86 gdbserver to debug native x86 Linux applications.

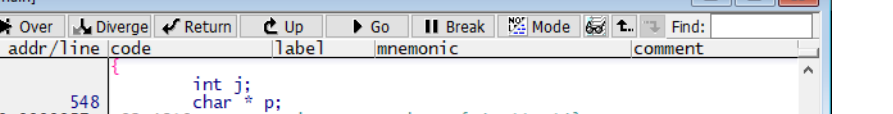
Multi-Process Debugging

Processes of Linux may reside virtually on the same addresses. To distinguish those addresses, the debugger uses an additional identifier called **space ID** (memory space ID) that specifies to which virtual memory space an address refers to. The space ID is equal to the process ID. Threads of a particular process use the same memory space. Consequently, threads of the same process have the same space ID.

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

Space ID

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



The screenshot shows the WinDbg interface with the following details:

- Registers:** The 'Space ID' register is highlighted in the 'Registers' pane on the left.
- Disassembly:** The 'Disassembly' pane on the right shows the assembly code for the function. The code is as follows:


```

int j;
char * p;
main:
    push    {r4,r11,r14}
    add     r11,r13,#0x8
    sub     r13,r13,#0x3C
    mov     r3,#0x0
    str     r3,[r11,#-0x18]
    vtripplearray[0][0][0] = 1;
    vtripplearray[1][0][0] = 2;
    vtripplearray[0][1][0] = 3;
    ldr     r3,0x97D0
    mov     r2,#0x1
      
```

A breakpoint to `main` in the process `sieve` can be set with one of the following commands:

```
Break.Set \\sieve\global\main
Break.Set 0x300:0x957C
Break.Set 786.:0x957C
```

Please be aware that process-specific breakpoints are set as soon as the process is started by **Go**.

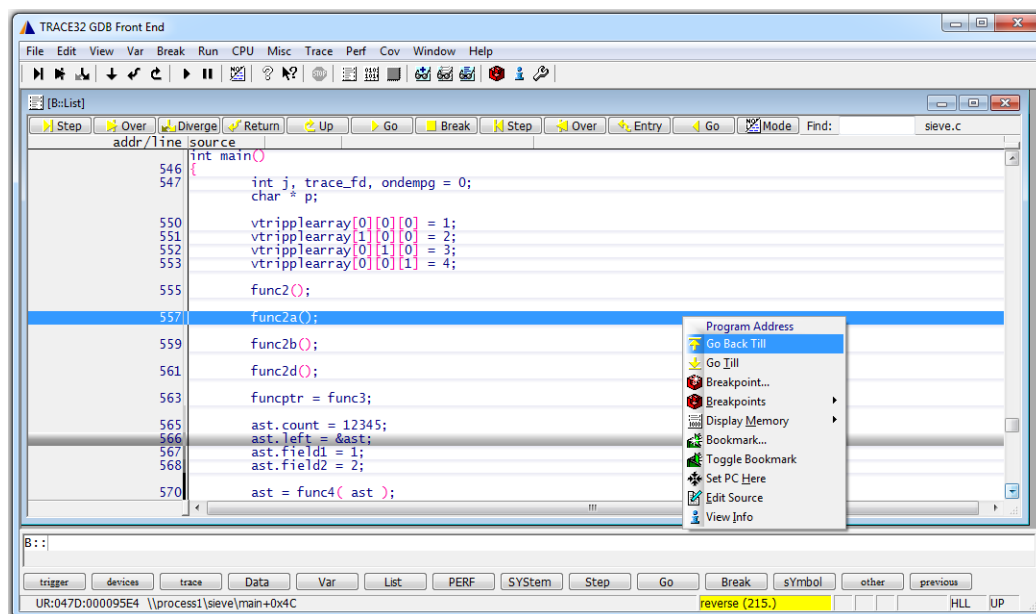
UndoDB Reversible Debugger

The TRACE32 GDB Front-End can be used as a front end for the UndoDB reversible debugger. The UndoDB target server allows to debug a Linux user space application as well as to records details of its execution. In addition to controlling the debugging process, the TRACE32 GDB Front-End also takes over the task of displaying the recorded data of the UndoDB target server in the TRACE32 PowerView user interface. Like a trace recording, the user has the ability to debug the application going through the code both forward and backward (“reverse-debugging”).

The TRACE32 GDB Front-End automatically detects if it is communicating with an UndoDB target server and enables the reverse-debugging commands.

Go.Back	Go back in program
Go.BackEntry	Go back in program to function entry
Step.Back	Step back
Step.BackChange	Step back till expression changes
Step.BackOver	Step back over call
Step.BackTill	Step back till expression is true

In reverse-debugging mode, all control buttons in the **List** window are displayed in yellow. In addition, the time of recording displayed in the TRACE32 PowerView **state line** is referenced in reverse.



The TRACE32 GDB Front-End supports debugging the Linux kernel using KGDB. Please note that it is only possible to debug the kernel space. Debugging the user-space as supported by the TRACE32 Stop Mode Debugging for Linux is not possible.

Example start-up script for Linux kernel debugging via KGDB:

```
; Reset all commands
RESet

; Clear all TRACE32 PowerView windows
WinCLEAR

; Select the CPU
SYStem.CPU *

; Enable address extension
SYStem.Option.MMUSPACES ON

; Set the communication parameter (serial port)
SYStem.PORT COM1 baud=115200 KGDB

; Attach to the KGDB stub in the Linux kernel
SYStem.Mode Attach

; Load the kernel symbols
Data.LOAD.Elf vmlinux /GNU /NoCODE

; Load the Linux Awareness (e.g Linux-3.x. for ARM)
TASK.CONFIG ~/demo/arm/kernel/linux/linux-3.x/linux3.t32
MENU.ReProgram ~/demo/arm/kernel/linux/linux-3.x/linux.men
```

The program execution can be stopped by writing 'g' to the file /proc/sysrq-trigger

```
echo g > /proc/sysrq-trigger
```


No information available until yet.

SYStem.CPU

Select target CPU

Format:

SYStem.CPU <type>

Selects the processor type.

SYStem.Mode

Establish communication to debug agent

Format:

SYStem.Mode <mode>

SYStem.Attach (alias for SYStem.Mode Attach)

SYStem.Down (alias for SYStem.Mode Down)

SYStem.Up (alias for SYStem.Mode Up)

<mode>:

Down

NoDebug

Attach

Up

Default: Down

Down	The TRACE32 GDB Front-End has no connection to the gdbserver / gdbstub. When switching from Up to Down , the TRACE32 GDB Front-End sends a GDB “kill” packet to the gdbserver / gdbstub.
NoDebug	The TRACE32 GDB Front-End has no connection to the gdbserver / gdbstub. When switching from Up to NoDebug , the TRACE32 GDB Front-End detaches from the gdbserver / gdbstub.
Attach	The TRACE32 GDB Front-End establishes the connection to gdbserver / gdbstub.
Up	The TRACE32 GDB Front-End is connected to gdbserver / gdbstub. Up cannot be used to establish the connection.
Go StandBy	Not available.

Format: **SYStem.Option.IMASKASM** [ON | OFF]

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.

Format: **SYStem.Option.IMASKHLL** [ON | OFF]

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After a single step, the interrupt mask bits are restored to the value before the step.

Format: **SYStem.Option.MMUSPACES** [ON | OFF]

Default: OFF.

Enables the use of [space IDs](#) for logical addresses to support **multiple** address spaces. This options need to be enabled if the gdbserver has been started in multi-process mode.

Format:SYSystem.Option.OVERLAY [ON | OFF | WithOVS]

Default: OFF.

- ON

Activates the overlay extension and extends the address scheme of the debugger with a 16 bit virtual overlay ID. Addresses therefore have the format `<overlay_id>:<address>`. This enables the debugger to handle overlaid program memory.
- OFF

Disables support for code overlays.
- WithOVS

Like option **ON**, but also enables support for software breakpoints. This means that TRACE32 writes software breakpoint opcodes to both, the *execution area* (for active overlays) and the *storage area*. This way, it is possible to set breakpoints into inactive overlays. Upon activation of the overlay, the target's runtime mechanisms copies the breakpoint opcodes to the execution area. For using this option, the storage area must be readable and writable for the debugger.

Example:

```
SYSystem.Option.OVERLAY ON
Data.List 0x2:0x11c4           ; Data.List <overlay_id>:<address>
```

SYSystem.RESetOut

Reset target

Format:SYSystem.RESetOut

Restarts the program being debugged by sending a GDB 'R' packet.

SYSystem.GDBconfig.BREAKSOFT

Use software breakpoint

Format:SYSystem.GDBconfig.BREAKSOFT ON | OFF

Default: OFF.

When set to **ON**, TRACE32 GDB Front-End sets breakpoints by patching the corresponding address by a breakpoint instruction instead of sending a "z/Z" RSP packet.

SYStem.GDBconfig.EXTENDED

Enable/disable gdb extended mode

Format:

SYStem.GDBconfig.EXTENDED ON | OFF
SYStem.Option.gdbEXTENDED ON | OFF (deprecated)

Default: OFF.

When set to **ON**, this command enables the GDB extended mode which makes the remote server persistent. If the remote server is a gdbserver started in multi-process mode, the extended mode has to be enabled. Otherwise TRACE32 will return the error “the target is not running” when establishing the connection.

SYStem.GDBconfig.GDBSERVER

Remote target is a gdbserver

Format:

SYStem.GDBconfig.GDBSERVER ON | OFF | AUTO

Default: AUTO.

- ON, OFF**

This option has to be set to **ON** if the TRACE32 GDB Front-End is connected to a GNU gdbserver. Otherwise set to **OFF**.
- AUTO**

The default value is **AUTO** which means that the TRACE32 GDB Front-End tries to detect what kind of remote target is used.

SYStem.GDBconfig.INFERIORID

Set inferior ID

Format:

SYStem.GDBconfig.INFERIORID <id>

Sets the inferior ID in case of multiple inferiors are supported by the gdbstub.

Format:

SYStem.GDBconfig.MONITOR *<string>*

This command is used to send monitor commands to the GDB Back-End.

SYStem.GDBconfig.NONSTOP

Enable/disable non-stop mode

Format:

SYStem.GDBconfig.NONSTOP ON | OFF
SYStem.Option.gdbNONSTOP ON | OFF (deprecated)

Default: OFF.

Enabling non-stop mode is only possible with gdbserver. Thus **SYStem.GDBconfig.NONSTOP ON** will automatically set **SYStem.GDBconfig.GDBSERVER** to **ON**.

NOTE:

Non-stop mode is supported for multi-thread debugging with the following known limitations:

- Resuming from a breakpoint is not supported for non-current threads.
- For the Intel x86/x64 architecture, if a non-current thread hits a breakpoint, a program counter alignment issue could occur.

Thus, it is always recommended to focus on debugging only one thread (current thread) at a given time.

SYStem.PORT

Set communication settings

Format:

SYStem.PORT *<settings>*

<settings>:

<com> **BAUD=***<baudrate>*
<host>:*<port>*

Sets the communication parameters. You can use a serial or a TCP communication.

```
SYStem.PORT COM1 baud=9600  
SYStem.PORT 10.1.2.99:2345
```

Format:	SYStem.GDBSIGnal <i><mode></i> <i><signum></i>
<i><mode></i> :	STOP NOSTOP PASS NOPASS PRINT NOPRINT

Defines the gdbserver/TRACE32 behavior when the application sends the signal *<signum>*.

<i><signum></i>	Signal number
NOPASS	Do not allow the program to see this signal.
NOPRINT	No message is printed.
NOSTOP	The signal is ignored and the application continue running.
PASS	Allow the program to see this signal; the program can handle the signal, or else it may terminate if the signal is fatal and not handled.
PRINT	A message is printed in the AREA window when the signal is received.
STOP	The debugger should stop on this signal.

GDB Front-End TASK Commands

The following TASK commands are available for the TRACE32 GDB Front-End:

TASK.List.tasks	List the running processes on the target.
TASK.COPYUP	Copy a file from the target to the host.
TASK.COPYDOWN	Copy a file from the host to the target.

The following TASK commands can additionally be used for **multi-process** mode:

TASK.select	Select a task for debugging.
TASK.ATTACH	Attach to a running process.
TASK.RUN	Start a new process.
TASK.KILL	Kill a running process.
TASK.DETACH	Detach from a process.
TASK.Go	Start the execution of a single task.
TASK.Break	Stop the execution of a single task.