

GRMON4

GRMON4 User's Manual



Table of Contents

1. Introduction	5
1.1. Overview	5
1.2. Supported platforms and system requirements	5
1.3. Obtaining GRMON	5
1.4. Installation	5
1.5. License	6
1.6. NOEL-V Support	6
1.6.1. Limitations	6
1.7. GRMON Evaluation version	6
1.8. Problem reports	6
2. Debugging concept	8
2.1. Overview	8
2.2. Target initialization	8
2.2.1. LEON2 target initialization	10
2.2.2. Configuration file target initialization	10
2.3. Memory register reset values	10
2.4. Hardware reset	10
3. Operation	11
3.1. Overview	11
3.2. Starting GRMON	11
3.2.1. Debug link options	11
3.2.2. Debug driver options	11
3.2.3. General options	12
3.3. GRMON command-line interface (CLI)	13
3.4. Common debug operations	14
3.4.1. Examining the hardware configuration	14
3.4.2. Uploading application and data to target memory	16
3.4.3. Running applications	16
3.4.4. Inserting breakpoints and watchpoints	17
3.4.5. Displaying processor registers	17
3.4.6. Backtracing function calls	18
3.4.7. Displaying memory contents	18
3.4.8. Instruction disassembly	19
3.4.9. Using the trace buffer	20
3.4.10. Profiling	21
3.4.11. Attaching to a target system without initialization	22
3.4.12. Attaching to a target system without Plug and Play scanning	22
3.4.13. Multi-processor support	22
3.4.14. Stack and entry point	23
3.4.15. Memory Management Unit (MMU) support	23
3.4.16. CPU cache support	24
3.5. Tcl integration	24
3.5.1. Shells	24
3.5.2. Commands	24
3.5.3. API	25
3.6. Symbolic debug information	25
3.6.1. Multi-processor symbolic debug information	25
3.7. GDB interface	26
3.7.1. Connecting GDB to GRMON	26
3.7.2. Executing GRMON commands from GDB	26
3.7.3. Running applications from GDB	27
3.7.4. Running SMP applications from GDB	27
3.7.5. Running AMP applications from GDB	28
3.7.6. GDB Thread support	29
3.7.7. Virtual memory	31
3.7.8. Specific GDB optimization	33
3.7.9. GRMON GUI considerations	33

3.7.10. Limitations of GDB interface	33
3.8. Thread support	33
3.8.1. GRMON thread options	33
3.8.2. GRMON thread commands	34
3.9. Forwarding application console I/O	35
3.10. EDAC protection	36
3.10.1. Using EDAC protected memory	36
3.10.2. LEON3-FT error injection	36
3.11. PROM programming	37
3.11.1. EEPROM	37
3.11.2. MRAM	37
3.11.3. CFI compatible Flash PROM	37
3.11.4. SPI memory device	38
3.12. Automated operation	39
3.12.1. Tcl commanding during CPU execution	39
3.12.2. Communication channel between target and monitor	39
3.12.3. Test suite driver	39
4. Graphical user interface	41
4.1. Overview	41
4.2. Starting GRMON GUI	41
4.3. Connect to target	42
4.3.1. Debug link	43
4.3.2. Options	43
4.3.3. Argument contribution	43
4.3.4. Configurations	43
4.3.5. Connect	44
4.4. Launch configurations	44
4.4.1. Target image setup	44
4.4.2. Launch properties	45
4.5. Perspectives	47
4.6. C/C++ source level debugging	47
4.6.1. GDB interface	47
4.7. Views	48
4.7.1. Debug View	48
4.7.2. System Information View	48
4.7.3. IO Registers View	49
4.7.4. CPU Registers View	51
4.7.5. Registers View	52
4.7.6. Router View	52
4.7.7. Source Editor	54
4.7.8. Terminals View	55
4.7.9. Memory View	57
4.7.10. Breakpoints View	57
4.7.11. Disassembly View	60
4.7.12. Messages View	63
4.7.13. Executables View	63
4.7.14. Outline View	64
4.7.15. Variables View	65
4.7.16. Expressions View	65
4.8. Target communication	66
4.8.1. Memory view update	66
4.9. Limitations	66
4.10. Troubleshooting the GUI	66
5. Debug link	68
5.1. UART debug link	68
5.2. Ethernet debug link	69
5.3. JTAG debug link	70
5.3.1. Xilinx parallel cable III/IV	71
5.3.2. Xilinx Platform USB cable	71

5.3.3. Altera USB Blaster or Byte Blaster	73
5.3.4. FTDI FT4232/FT2232	74
5.3.5. Amontec JTAGkey	75
5.3.6. Actel FlashPro 3/3x/4/5	75
5.3.7. Digilent HS1/HS2/HS3/SMT2/SMT3	75
5.4. USB debug link	75
5.5. GRESB debug link	77
5.5.1. AGGA4 SpaceWire debug link	78
5.6. User defined debug link	78
5.6.1. API	78
6. Debug drivers	80
6.1. AMBA AHB trace buffer driver	80
6.2. Clock gating	80
6.2.1. Switches	80
6.3. Debug support drivers	80
6.3.1. Switches	81
6.3.2. Commands	81
6.3.3. Tcl variables	82
6.4. Ethernet controller	82
6.4.1. Commands	82
6.5. GRPWM core	83
6.6. USB Host Controller	83
6.6.1. Switches	83
6.6.2. Commands	83
6.7. I ² C	83
6.8. I/O Memory Management Unit	83
6.9. Multi-processor interrupt controller	84
6.10. L2-Cache Controller	84
6.10.1. Switches	85
6.10.2. Errata	85
6.11. Statistics Unit	85
6.12. LEON2 support	87
6.12.1. Switches	87
6.13. On-chip logic analyzer driver	88
6.14. Memory controllers	88
6.14.1. Switches	89
6.14.2. Commands	91
6.15. Memory scrubber	91
6.16. MIL-STD-1553B Interface	91
6.17. PCI	93
6.17.1. PCI Trace	96
6.18. GR716B Real-Time Accelerator	97
6.18.1. Switches	98
6.19. SPI	98
6.20. SpaceWire router	99
6.21. SVGA frame buffer	99
7. Support	100
A. Command index	101
B. Command syntax	105
C. Tcl API	264
D. Fixed target configuration file format	273
E. License key installation	275
F. Appending environment variables	276
G. Compatibility	277
G.1. Compatibility notes for GRMON3	277
G.2. Compatibility notes for GRMON2	277
G.3. Compatibility notes for GRMON1	277
H. Third-party licenses	279

1. Introduction

1.1. Overview

GRMON is a general debug monitor for the LEON (SPARC V7/V8) processor, NOEL-V (RISC-V) processor and for SOC designs based on the GRLIB IP library. GRMON includes the following functions:

- Read/write access to all system registers and memory
- Built-in disassembler and trace buffer management
- Downloading and execution of LEON applications
- Breakpoint and watchpoint management
- Remote connection to GNU debugger (GDB)
- Support for USB, JTAG, UART, Ethernet and SpaceWire debug links
- Tcl interface (scripts, procedures, variables, loops etc.)
- Graphical user interface

1.2. Supported platforms and system requirements

GRMON is currently provided for platforms: Linux (GLIBC >2.11), Windows 7 and Windows 10.

To run the GUI Java 11 (64-bit) is required. A free to use Java runtime environment (JRE) based on OpenJDK can be downloaded from <https://adoptium.net/>.

The professional version use a Sentinel LDK license key which has additional system requirements, which can be found the the README that is included in Sentinel LDK Runtime installation package. See Appendix E, *License key installation* for more information.

The available debug communication links for each platform vary and they may have additional third-party dependencies that have additional system requirements. See Chapter 5, *Debug link* for more information.

1.3. Obtaining GRMON

The latest version of GRMON can be ordered and evaluation versions downloaded from the website [frontgrade.com/gaisler].

1.4. Installation

Follow these steps to install GRMON. Detailed information can be found further down.

1. Extract the archive
2. Install the Sentinel LDK Runtime (GRMON Pro version)
3. Install the Java runtime environment 11
4. Optionally install third-party drivers for the debug interfaces.
5. Optionally setup the path for shared libraries (Linux only)
6. Optionally add GRMON to the environment variable PATH

To install GRMON, extract the archive anywhere on the host computer. There is one archive for each OS that GRMON supports, and they may be extracted to the same location, Each archive contains the directories as described in the list below.

```
grmon-pro-4.0.XX/<OS>/bin32
grmon-pro-4.0.XX/<OS>/bin64
grmon-pro-4.0.XX/<OS>/lib32
grmon-pro-4.0.XX/<OS>/lib64
grmon-pro-4.0.XX/<OS>/share
```

The professional version use a Sentinel LDK license key. See Appendix E, *License key installation* for installation of the Sentinel LDK runtime.

Some debug interfaces requires installation of third-party drivers, see Chapter 5, *Debug link* for more information.

The `bin<BITS>` directory contains the executable. For convenience it is recommended to add the `bin<BITS>` directory of the host OS to the environment variable `PATH`. See Appendix F, *Appending environment variables* for instructions on how to append environment variables.

The `lib<BITS>` directory contains some additional libraries that GRMON requires. On the Windows platform the `lib<BITS>` directory is not available. On the Linux platform, if GRMON fails to start because of some missing libraries that are located in this directory, then add this path to the environment variable `LD_LIBRARY_PATH` or add it the `ld.so.cache` (see man pages about `ldconfig` for more information).

GRMON must find the `share` directory to work properly. GRMON will try to automatically detect the location of the folder. A warning will be printed when starting GRMON if it fails to find the `share` folder. If it fails to automatically detect the folder, then the environment variable `GRMON_SHARE` can be set to point the `share/grmon` folder. For example on Windows it could be set to `c:\opt\grmon-pro\windows\share\grmon` or on Linux it could be set to `/opt/grmon-pro/linux/share/grmon`.

1.5. License

The GRMON license file can be found in the `share` folder of the installation. For example on Windows it can be found in `c:\opt\grmon-pro\windows\share\grmon` or on Linux it could be found in `/opt/grmon-pro/linux/share/grmon`.

1.6. NOEL-V Support

Both the Pro and the evaluation version of GRMON supports the NOEL-V processor.

Many examples in this manual shows output from a LEON target system, however many of the commands will work with NOEL-V as well.

1.6.1. Limitations

- Backtrace requires DWARF information.
- No support for commands **ahb**, **profile**
- No support for switches `-nb`, `-mpgsz`

1.7. GRMON Evaluation version

The evaluation version of GRMON can be downloaded from the website [frontgrade.com/gaisler]. The evaluation version may be used during a period of 21 days without purchasing a license. After this period, any commercial use of GRMON is not permitted without a valid license.

The GRMON evaluation version can be used to connect to FPGA designs that contains GRLIB GPL cores.

The example bitstreams of LEON5 and NOEL-V are supported as well. See the Quick-start guides for additional information on how use GRMON with the NOEL-V example bitstreams. <https://www.gaisler.com/index.php/products/processors/noel-v-examples> <https://www.gaisler.com/index.php/products/processors/leon-examples>

Borrowed GR716-MINI, GR716B-MINI and GR740-MINI boards are also supported by the GRMON evaluation version.

The following features are in general *not* supported in the evaluation version:

- GUI
- Support for LEON2, LEON3-FT, LEON4, LEON5-FT NOEL-V FT
- GRLIB commercial licensed cores
- FT support and error injection
- Custom JTAG configuration
- Profiling
- TCL API (drivers, init scripts, hooks, I/O forward to TCL channel etc)

1.8. Problem reports

Please send bug reports or comments to support@gaisler.com.

Customers with a valid support agreement may send questions to support@gaisler.com. Include a GRMON log when sending questions, please. A log can be obtained by starting GRMON with the command line switch `-log filename`.

The GRLIB Discourse [<http://discourse.grlib.community/>] may also be a source to find solutions to problems.

2. Debugging concept

2.1. Overview

The GRMON debug monitor is intended to debug system-on-chip (SOC) designs based on the GRLIB IP library. The monitor connects to a dedicated debug interface on the target hardware, through which it can perform read and write cycles on the on-chip bus (AHB). The debug interface can be of various types: the processor supports debugging over a serial UART, 32-bit PCI, JTAG, Ethernet and SpaceWire (using the GRESB Ethernet to SpaceWire bridge) debug interfaces. On the target system, all debug interfaces are realized as AHB masters with the Debug protocol implemented in hardware. There is thus no software support necessary to debug a target system, and a target system does in fact not even need to have a processor present.

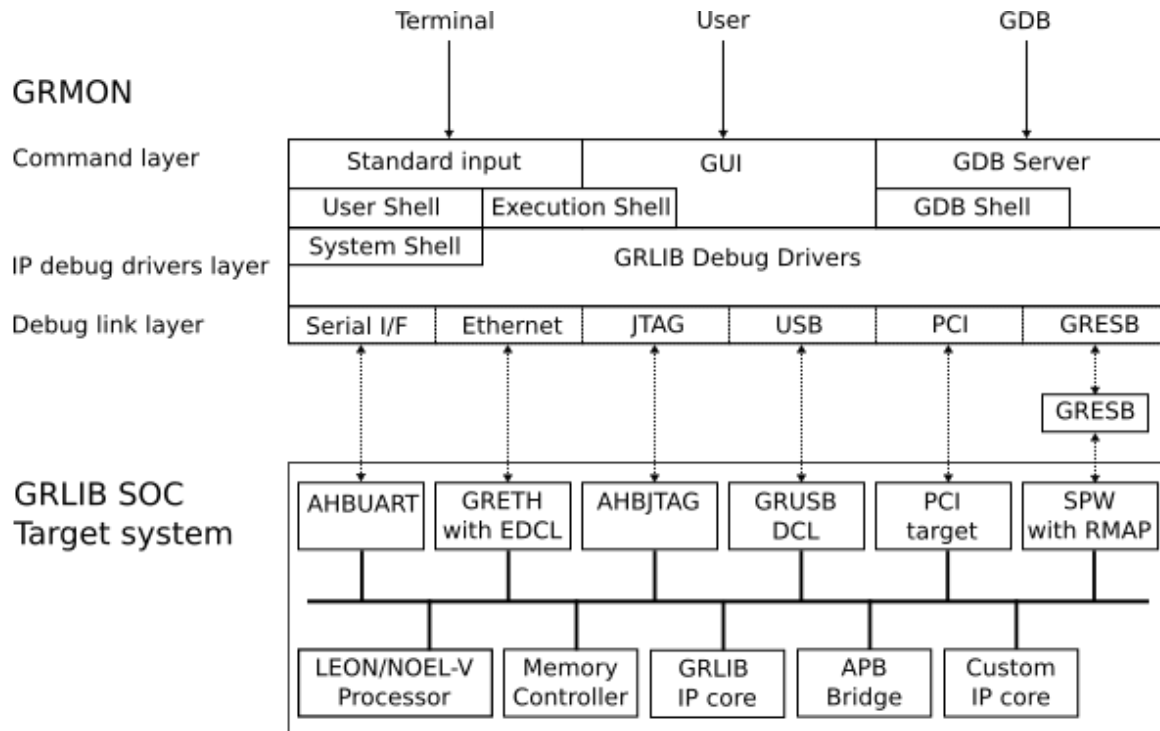


Figure 2.1. GRMON concept overview

GRMON can operate in three modes: graphical user interface, command-line interface or GDB mode. In command-line mode, GRMON commands are entered through a terminal window. It can also be used for automated scripts. In GDB mode, GRMON acts as a GDB server and translates the GDB remote server protocol to debug commands on the target system.

GRMON is implemented using three functional layers: command layer, debug driver layer, and debug link layer. The command layer takes input from the user and parses it in a Tcl Shell. It is also possible to start a GDB server service, which has its own shell, that takes input from GDB. Each shell has its own set of commands and variables. Many commands depend on drivers and will fail if the core is not present in the target system. More information about Tcl integration can be found in the Section 3.5, “Tcl integration”.

The debug driver layer implements drivers that probe and initialize the cores. GRMON will scan the target system at start-up and detect which IP cores are present. The drivers may also provide information to the commands.

The debug link layer implements the debug link protocol for each supported debug interface. Which interface to use for a debug session is specified through command line options during the start of GRMON. Only interfaces based on JTAG support 8-/16-bit accesses, all other interfaces access subwords using read-modify-write. 32-bit accesses are supported by all interfaces. More information can be found in Chapter 5, *Debug link*.

2.2. Target initialization

When GRMON first connects to the target system, it scans the system to detect which IP cores are present. This is done by reading the plug and play information which is normally located at address 0xffff000 on the AHB bus. A

debug driver for each recognized IP core is then initialized, and performs a core-specific initialization sequence if required. For a memory controller, the initialization sequence would typically consist of a memory probe operation to detect the amount of attached RAM. For a UART, it could consist of initializing the baud rate generator and flushing the FIFOs. After the initialization is complete, the system configuration is printed:

```
GRMON3 LEON debug monitor v3.0.0 32-bit professional version
```

```
Copyright (C) 2018 Frontgrade Gaisler - All rights reserved.
For latest updates, go to http://www.gaisler.com/
Comments or bug-reports to support@gaisler.com
```

```
GRLIB build version: 4111
Detected frequency: 40 MHz
```

Component	Vendor
LEON3 SPARC V8 Processor	Frontgrade Gaisler
AHB Debug UART	Frontgrade Gaisler
JTAG Debug Link	Frontgrade Gaisler
GRSPW2 SpaceWire Serial Link	Frontgrade Gaisler
LEON2 Memory Controller	European Space Agency
AHB/APB Bridge	Frontgrade Gaisler
LEON3 Debug Support Unit	Frontgrade Gaisler
Generic UART	Frontgrade Gaisler
Multi-processor Interrupt Ctrl.	Frontgrade Gaisler
Modular Timer Unit	Frontgrade Gaisler
General Purpose I/O port	Frontgrade Gaisler

```
Use command 'info sys' to print a detailed report of attached cores
```

```
grmon3>
```

More detailed system information can be printed using the **'info sys'** command as listed below. The detailed system view also provides information about address mapping, interrupt allocation and IP core configuration. Information about which AMBA AHB and APB buses a core is connected to can be seen by adding the **-v** option. GRMON assigns a unique name to all cores, the core name is printed to the left. See Appendix C, *Tcl API* for information about Tcl variables and device names.

```
grmon3> info sys
cpu0      Frontgrade Gaisler  LEON3 SPARC V8 Processor
          AHB Master 0
ahbuart0  Frontgrade Gaisler  AHB Debug UART
          AHB Master 1
          APB: 80000700 - 80000800
          Baudrate 115200, AHB frequency 40000000.00
ahbjtag0  Frontgrade Gaisler  JTAG Debug Link
          AHB Master 2
grspw0    Frontgrade Gaisler  GRSPW2 SpaceWire Serial Link
          AHB Master 3
          APB: 80000A00 - 80000B00
          IRQ: 10
          Number of ports: 1
mctrl0    European Space Agency  LEON2 Memory Controller
          AHB: 00000000 - 20000000
          AHB: 20000000 - 40000000
          AHB: 40000000 - 80000000
          APB: 80000000 - 80000100
          8-bit prom @ 0x00000000
          32-bit sdram: 1 * 64 Mbyte @ 0x40000000
          col 9, cas 2, ref 7.8 us
apbmst0   Frontgrade Gaisler  AHB/APB Bridge
          AHB: 80000000 - 80100000
dsu0      Frontgrade Gaisler  LEON3 Debug Support Unit
          AHB: 90000000 - A0000000
          AHB trace: 128 lines, 32-bit bus
          CPU0: win 8, hwbp 2, itrace 128, V8 mul/div, srmmu, lddel 1
          stack pointer 0x43fffff0
          icache 2 * 4096 kB, 32 B/line lru
          dcache 1 * 4096 kB, 16 B/line
uart0     Frontgrade Gaisler  Generic UART
          APB: 80000100 - 80000200
          IRQ: 2
          Baudrate 38461
irqmp0    Frontgrade Gaisler  Multi-processor Interrupt Ctrl.
          APB: 80000200 - 80000300
gptimer0  Frontgrade Gaisler  Modular Timer Unit
          APB: 80000300 - 80000400
          IRQ: 8
          8-bit scalar, 2 * 32-bit timers, divisor 40
grgpio0   Frontgrade Gaisler  General Purpose I/O port
          APB: 80000800 - 80000900
```

2.2.1. LEON2 target initialization

The plug and play information was introduced in the LEON3 processor (GRLIB), and is not available for LEON2 systems. LEON2 is supported by starting GRMON with the `-sys leon2` switch or one of the switches that correspond to a known LEON2 device, see Section 6.12, “LEON2 support”.

A LEON2 system has a fixed set of IP cores and address mapping. GRMON will use an internal plug and play table that describes this configuration. The plug and play table used for LEON2 is fixed, and no automatic detection of present cores is attempted. Only those cores that need to be initialized by GRMON are included in the table, so the listing might not correspond to the actual target. It is however possible to load a custom configuration file that describes the target system configuration using see Section 2.2.2, “Configuration file target initialization”

2.2.2. Configuration file target initialization

It is possible to provide GRMON with a configuration file that describes a static configuration by starting GRMON with the switch `-cfg filename`.

The format of the plug and play configuration file is described in section Appendix D, *Fixed target configuration file format*. It can be used for both LEON3 and LEON2 systems. An example configuration file is also supplied with the GRMON professional distribution in `share/src/cfg/leon3.xml`.

2.3. Memory register reset values

To ensure that the memory registers has sane values, GRMON will reset the registers when commands that access the memories are issued, for example **run**, **load** commands and similar commands. To modify the reset values, use the commands listed in Section 6.14.2, “Commands”.

2.4. Hardware reset

The behaviour of GRMON will be undefined after a hardware resets signal is asserted. Most debug links will drop the connection, but some will remain connected, for example JTAG. Therefor it is recommended that you always restart GRMON after a reset signal has been asserted.

3. Operation

This chapter describes how GRMON can be controlled by the user in a terminal based interactive debug session and how it can be automated with scripts for batch execution. The first sections describe and exemplifies typical operations for interactive use. The later sections describe automation concepts. Most interactive commands are applicable also for automated use.

GRMON graphical user interface is described in Chapter 4, *Graphical user interface*.

3.1. Overview

An interactive GRMON debug session typically consists of the following steps:

1. Starting GRMON and attaching to the target system
2. Examining the hardware configuration
3. Uploading application program
4. Setup debugging, for example insert breakpoints and watchpoints
5. Executing the application
6. Debugging the application and examining the CPU and hardware state

Step 2 though 6 is performed using the GRMON terminal interface or by attaching GDB and use the standard GDB interface. The GDB section describes how GRMON specific commands are accessed from GDB.

The following sections will give an overview how the various steps are performed.

3.2. Starting GRMON

On a Linux host, GRMON is started by giving the **grmon** command together with command line options in a terminal window. It can run either the GUI or a commandline interface depending on if a debug link option is provided or not.

On Windows hosts, there are two executable provided. The file **grmon.exe** is intended to be started in a Windows command prompt (**cmd.exe**). It can run either the GUI or a commandline interface depending on if a debug link option is provided or not. The executable **grmon-gui.exe** will always spawn a GUI.

Command line options are grouped by function as indicated below.

- The debug link options: setting up a connection to GRLIB target
- General options: debug session behavior options
- Debug driver options: configure the hardware, skip core auto-probing etc.

If any debug-link option is given to **grmon** or **grmon.exe**, then the GRMON command line interface will be started.

If *no* debug-link option is given to **grmon** or **grmon.exe**, then the GRMON graphical user interface will be started. For more information, see Chapter 4, *Graphical user interface*.

Below is an example of GRMON connecting to a GR712 evaluation board using the FTDI USB serial interface, tunneling the UART output of APBUART0 to GRMON and specifying three RAM wait states on read and write:

```
$ grmon -ftdi -u -ramws 3
```

To connect to a target using the AHBUART debug link, the following example can be used:

```
$ grmon -uart -u
```

The `-uart` option uses the first UART of the host (ttyS0 or COM1) with a baud rate of 115200 baud by default.

3.2.1. Debug link options

GRMON connects to a GRLIB target using one debug link interface, the command line options selects which interface the PC uses to connect to the target and optionally how the debug link is configured. All options are described in Chapter 5, *Debug link*.

3.2.2. Debug driver options

The debug drivers provide an interface to view and access AMBA devices during debugging and they offer device specific ways to configure the hardware when connecting and before running the executable. Drivers usually au-

to-probe their devices for optimal configuration values, however sometimes it is useful to override the auto-probed values. Some options affects multiple drivers. The debug driver options are described in Chapter 6, *Debug drivers*.

3.2.3. General options

The general options are mostly target independent options configuring the behavior of GRMON. Some of them affects how the target system is accessed both during connection and during the whole debugging session. All general options are described below.

`grmon [options]`

Options:

- abaud *baudrate*
Set baud-rate for all UARTs in the system, (except the debug-link UART). By default, 38400 baud is used.
- ambamb [*maxbuses*]
Enable auto-detection of AHBCTRL_MB system and (optionally) specifies the maximum number of buses in the system if an argument is given. The optional argument to -ambamb is decoded as below:
0, 1: No Multi-bus (MB) (max one bus)
2..3: Limit MB support to 2 or 3 AMBA PnP buses
4 or no argument: Selects Full MB support
- batch *filename*
Only run startup scripts then quit. Use -c or -e to set the startup scripts.
- c *filename*
Run the commands in the batch file at start-up.
- cfg *filename*
Load fixed PnP configuration from a xml-file. See Appendix D, *Fixed target configuration file format*.
- e *string*
Run a command at start-up.
- echo
Echo all the commands in the batch file at start-up. Has no effect unless -c or -e is also set.
- edac
Enable EDAC operation in memory controllers that support it.
- freq *sysclk*
Overrides the detected system frequency. The frequency is specified in MHz.
- gdb [*port*]
Listen for GDB connection directly at start-up. Optionally specify the port number for GDB communications. Default port number is 2222.
- gui
Start the GRMON graphical user interface. This option can be combined with an option which specifies the debug-link to use. See Chapter 4, *Graphical user interface* for more information.
- guiport *port*
Force the GUI to connect via the specified TCP port. See Chapter 4, *Graphical user interface* for more information.
- ioarea *address*
Specify the location of the I/O area. (Default is 0xffff0000).
- log *filename*
Log session to the specified file. If the file already exists the new session is appended. This should be used when requesting support.
- nologheader
Suppress writing header to log.
- nologsys
Suppress writing system information to the logfile during startup.
- nologtag
Suppress writing timetag to log.
- nologtime
Suppress writing tags to log.
- ni
Read plug n' play and detect all system device, but don't do any target initialization. See Section 3.4.11, "Attaching to a target system without initialization" for more information.

-nopnp

Disable the plug n' play scanning. GRMON won't detect any hardware and any hardware dependent functionality won't work. See Section 3.4.12, “Attaching to a target system without Plug and Play scanning”.

-u [*device*]

Put UART 1 in FIFO debug mode if hardware supports it, else put it in loop-back mode. Debug mode will enable both reading and writing to the UART from the monitor console. Loop-back mode will only enable reading. See Section 3.9, “Forwarding application console I/O”. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

-ucli [*device*]

Put UART in debug or loop-back mode to forward application console I/O to GRMON CLI on startup. The command line shell will be used for forwarding instead of the regular prompt. See Section 3.9, “Forwarding application console I/O”. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

-udm [*device*]

Put UART 1 in FIFO debug mode if hardware supports it. Debug mode will enable both reading and writing to the UART from the monitor console. See Section 3.9, “Forwarding application console I/O”. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

-ulb [*device*]

Put UART 1 in loop-back mode. Loop-back mode will only enable reading from the UART to the monitor console. See Section 3.9, “Forwarding application console I/O”. The optional device parameter is used to select a specific UART to be put in debug mode. The device parameter is an index starting with 0 for the first UART and then increasing with one in the order they are found in the bus scan. If the device parameter is not used the first UART is selected.

-ucmd *filename*

Load script specified by filename into all shells, including the system shell.

-udrv *filename*

Load script specified by filename into system shell.

3.3. GRMON command-line interface (CLI)

The GRMON4 command-line interface features a Tcl 8.6 interpreter which will interpret all entered commands substituting variables etc. before GRMON is actually called. Variables exported by GRMON can also be used to access internal states and hardware registers without going through commands. The GRMON Tcl interface is described in Section 3.5, “Tcl integration”.

Short forms of the commands are allowed, e.g lo, loa, or load, are all interpreted as load. Tab completion is available for commands, Tcl variables, text-symbols, file names, etc.

The commands can be separated in to three categories:

- Tcl internal commands and reserved key words
- GRMON built-in commands always available regardless of target
- GRMON commands accessing debug drivers

Tcl internal and GRMON built-in commands are available regardless of target hardware present whereas debug driver commands may only be present on supported systems. The Tcl and driver commands are described in Section 3.5, “Tcl integration” and Chapter 6, *Debug drivers* respectively. In Table 3.1 is a summary of all GRMON built-in commands. For the full list of commands, see Appendix A, *Command index*.

Table 3.1. BUILT-IN commands

about	Show information about GRMON
amem	Asynchronous bus read
batch	Execute batch script
bdump	Dump memory to a file

blod	Load a binary file
disassemble	Disassemble memory
dtb	Setup a DTB to be uploaded or print filenames of DTB files
dump	Dump memory to a file
eeload	Load a file into an EEPROM
execsh	Run commands in the execution shell
exit	Exit GRMON
fpgaload	Upload bitstream to GR740-MINI board FPGA
gdb	Control the built-in GDB remote server
gui	Control the graphical user interface
help	Print all commands or detailed help for a specific command
info	Show information
load	Load a file or print filenames of uploaded files
memb	AMBA bus 8-bit memory read access, list a range of addresses
memd	AMBA bus 64-bit memory read access, list a range of addresses
memh	AMBA bus 16-bit memory read access, list a range of addresses
mem	AMBA bus 32-bit memory read access, list a range of addresses
nolog	Suppress stdout of a command
quit	Quit the GRMON console
reset	Reset drivers
rtg4fddr	Print initialization sequence
rtg4serdes	Print initialization sequence
sf2mddr	Print initialization sequence
sf2serdes	Print initialization sequence
shell	Execute shell process
silent	Suppress stdout of a command
symbols	Load, print or lookup symbols
system	Attach or detach devices
tps	Control the TPS service
usrsh	Run commands in threaded user shell
verify	Verify that a file has been uploaded correctly
wash	Clear or set memory areas
wmemb	AMBA bus 8-bit memory write access
wmemd	AMBA bus 64-bit memory write access
wmemh	AMBA bus 16-bit memory write access
wmems	Write a string to an AMBA bus memory address
wmem	AMBA bus 32-bit memory write access

3.4. Common debug operations

This section describes and gives some examples of how GRMON is typically used, the full command reference can be found in Appendix A, *Command index*.

3.4.1. Examining the hardware configuration

When connecting for the first time it is essential to verify that GRMON has auto-detected all devices and their configuration correctly. At start-up GRMON will print the cores and the frequency detected. From the command line one can examine the system by executing **info sys** as below:

```

grmon3> info sys
cpu0      Frontgrade Gaisler  LEON3-FT SPARC V8 Processor
          AHB Master 0
cpul      Frontgrade Gaisler  LEON3-FT SPARC V8 Processor
          AHB Master 1
greth0    Frontgrade Gaisler  GR Ethernet MAC
          AHB Master 3
          APB: 80000E00 - 80000F00
          IRQ: 14
grspw0    Frontgrade Gaisler  GRSPW2 SpaceWire Serial Link
          AHB Master 5
          APB: 80100800 - 80100900
          IRQ: 22
          Number of ports: 1
grspw1    Frontgrade Gaisler  GRSPW2 SpaceWire Serial Link
          AHB Master 6
          APB: 80100900 - 80100A00
          IRQ: 23
          Number of ports: 1
mctrl0    Frontgrade Gaisler  Memory controller with EDAC
          AHB: 00000000 - 20000000
          AHB: 20000000 - 40000000
          AHB: 40000000 - 80000000
          APB: 80000000 - 80000100
          8-bit prom @ 0x00000000
          32-bit static ram: 1 * 8192 kbyte @ 0x40000000
          32-bit sdram: 2 * 128 Mbyte @ 0x60000000
          col 10, cas 2, ref 7.8 us
apbmst0   Frontgrade Gaisler  AHB/APB Bridge
          AHB: 80000000 - 80100000
dsu0      Frontgrade Gaisler  LEON3 Debug Support Unit
          AHB: 90000000 - A0000000
          AHB trace: 256 lines, 32-bit bus
CPU0:    win 8, hwbp 2, itrace 256, V8 mul/div, srmmu, lddel 1, GRFPU
          stack pointer 0x407ffff0
          icache 4 * 4096 kB, 32 B/line lru
          dcache 4 * 4096 kB, 16 B/line lru
CPU1:    win 8, hwbp 2, itrace 256, V8 mul/div, srmmu, lddel 1, GRFPU
          stack pointer 0x407ffff0
          icache 4 * 4096 kB, 32 B/line lru
          dcache 4 * 4096 kB, 16 B/line lru
uart0     Frontgrade Gaisler  Generic UART
          APB: 80000100 - 80000200
          IRQ: 2
          Baudrate 38461, FIFO debug mode
irqmp0    Frontgrade Gaisler  Multi-processor Interrupt Ctrl.
          APB: 80000200 - 80000300
          EIRQ: 12
gptimer0  Frontgrade Gaisler  Modular Timer Unit
          APB: 80000300 - 80000400
          IRQ: 8
          16-bit scalar, 4 * 32-bit timers, divisor 80
grgpio0   Frontgrade Gaisler  General Purpose I/O port
          APB: 80000900 - 80000A00
uart1     Frontgrade Gaisler  Generic UART
          APB: 80100100 - 80100200
          IRQ: 17
          Baudrate 38461
...

```

The memory section for example tells us that GRMON are using the correct amount of memory and memory type. The parameters can be tweaked by passing memory driver specific options on start-up, see Section 3.2, “Starting GRMON”. The current memory settings can be viewed in detail by listing the registers with **info reg** or by accessing the registers by the Tcl variables exported by GRMON:

```

grmon3> info sys
...
mctrl0    Frontgrade Gaisler  Memory controller with EDAC
          AHB: 00000000 - 20000000
          AHB: 20000000 - 40000000
          AHB: 40000000 - 80000000
          APB: 80000000 - 80000100
          8-bit prom @ 0x00000000
          32-bit static ram: 1 * 8192 kbyte @ 0x40000000
          32-bit sdram: 2 * 128 Mbyte @ 0x60000000
          col 10, cas 2, ref 7.8 us
...
grmon3> info reg
...
Memory controller with EDAC
0x80000000 Memory config register 1          0x1003c0ff
0x80000004 Memory config register 2          0x9ac05463
0x80000008 Memory config register 3          0x0826e000

```



```

...
grmon3> puts [format 0x%08x $mctrl0::          [TAB-COMPLETION]
mctrl0::mcfg1 mctrl0::mcfg2 mctrl0::mcfg3 mctrl0::pnp::
mctrl0::mcfg1:: mctrl0::mcfg2:: mctrl0::mcfg3::
grmon3> puts [format 0x%08x $mctrl0::mcfg1]
0x0003c0ff

grmon3> puts [format 0x%08x $mctrl0::mcfg2 ::          [TAB-COMPLETION]
mctrl0::mcfg2::d64 mctrl0::mcfg2::sdramcmd
mctrl0::mcfg2::rambanksz mctrl0::mcfg2::sdramcolsz
mctrl0::mcfg2::ramrws mctrl0::mcfg2::sdramrf
mctrl0::mcfg2::ramwidth mctrl0::mcfg2::sdramtcas
mctrl0::mcfg2::ramwvs mctrl0::mcfg2::sdramtrfc
mctrl0::mcfg2::rbrdy mctrl0::mcfg2::sdramtrp
mctrl0::mcfg2::rmw mctrl0::mcfg2::se
mctrl0::mcfg2::sdpb mctrl0::mcfg2::si
mctrl0::mcfg2::sdrambanksz
grmon3> puts [format %x $mctrl0::mcfg2::ramwidth]
2

```

3.4.2. Uploading application and data to target memory

A software application can be uploaded to the target system memory using the **load** command:

```

grmon3> load v8/stanford.exe
40000000 .text          54.8kB /  54.8kB  [=====>] 100%
4000DB30 .data         2.9kB /   2.9kB  [=====>] 100%
Total size: 57.66kB (786.00kbit/s)
Entry point 0x40000000
Image /home/daniel/examples/v8/stanford.exe loaded

```

The supported file formats are SPARC ELF-32, SPARC ELF-64 (MSB truncated to 32-bit addresses), RISC-V ELF-64, RISC-V ELF-32, srecord and a.out binaries. Each section is loaded to its link address. The program entry point of the file is used to set the %PC, %NPC when the application is later started with run. It is also possible to load binary data by specifying file and target address using the **load** command.

One can use the **verify** command to make sure that the file has been loaded correctly to memory as below. Any discrepancies will be reported in the GRMON console.

```

grmon3> verify v8/stanford.exe
40000000 .text          54.8kB /  54.8kB  [=====>] 100%
4000DB30 .data         2.9kB /   2.9kB  [=====>] 100%
Total size: 57.66kB (726.74kbit/s)
Entry point 0x40000000
Image of /home/daniel/examples/v8/stanford.exe verified without errors

```

On-going DMA can be turned off to avoid that hardware overwrites the loaded image by issuing the **reset** command prior to **load**. This is important after the CPU has been executing using DMA in for example Ethernet network traffic.

3.4.3. Running applications

After the application has been uploaded to the target with **load** the **run** command can be used to start execution. The entry-point taken from the ELF-file during loading will serve as the starting address, the first instruction executed. The **run** command issues a driver reset, however it may be necessary to perform a reset prior to loading the image to avoid that DMA overwrites the image. See the **reset** command for details. Applications already located in FLASH can be started by specifying an absolute address. The **cont** command resumes execution after a temporary stop, e.g. a breakpoint hit. **go** also affects the CPU execution, the difference compared to **run** is that the target device hardware is not initialized before starting execution.

```

grmon3> reset
grmon3> load v8/stanford.exe
40000000 .text          54.8kB /  54.8kB  [=====>] 100%
4000DB30 .data         2.9kB /   2.9kB  [=====>] 100%
Total size: 57.66kB (786.00kbit/s)
Entry point 0x40000000
Image /home/daniel/examples/v8/stanford.exe loaded

grmon3> run
Starting
  Perm Towers Queens Intmm   Mm  Puzzle  Quick Bubble  Tree  FFT
    34    67    33   117  1117   367    50    50   250  1133

Nonfloating point composite is      144

Floating point composite is        973

```



```
CPU 0: Program exited normally.
CPU 1: Power down mode
```

The output from the application normally appears on the target system UARTs and thus not in the GRMON console. However, if GRMON is started with the `-u` switch, the UART is put into debug mode and the output is tunneled over the debug-link and finally printed on the console by GRMON. See Section 3.9, “Forwarding application console I/O”. Note that older hardware (GRLIB 1.0.17-b2710 and older) has only partial support for `-u`, it will not work when the APBUART software driver uses interrupt driven I/O, thus Linux and vxWorks are not supported on older hardware. Instead, a terminal emulator should be connected to UART 1 of the target system.

Since the application changes (at least) the `.data` segment during run-time the application must be reloaded before it can be executed again. If the application uses the MMU (e.g. Linux) or installs data exception handlers (e.g. eCos), GRMON should be started with `-nb` to avoid going into break mode on a page-fault or data exception. Likewise, when a software debugger is running on the target (e.g. GDB natively in Linux user-space or WindRiver Workbench debugging a task) soft breakpoints (“TA 0x01” instruction) will result in traps that the OS will handle and tell the native debugger. To prevent GRMON from interpreting it as its own breakpoints and stop the CPU one must use the `-nswb` switch.

3.4.4. Inserting breakpoints and watchpoints

All breakpoints are inserted with the `bp` command. The subcommand (soft, hard, watch, bus, data, delete) given to `bp` determine which type of breakpoint is inserted, if no subcommand is given `bp` defaults to a software breakpoint.

Instruction breakpoints are inserted using `bp soft` or `bp hard` commands. Inserting a software breakpoint will add a (SPARC “ta 0x1” or RISC-V “ebreak”) instruction by modifying the target’s memory before starting the CPU, while `bp hard` will insert a hardware breakpoint using one of the LEON IU watchpoint registers or RISC-V triggers.. To debug instruction code in read-only memories or memories which are self-modifying the only option is hardware breakpoints. Note that it’s possible to debug any RAM-based code using software breakpoints, even where traps are disabled such as in trap handlers. Since hardware breakpoints triggers on the CPU instruction address one must be aware that when the MMU is turned on, virtual addresses are triggered upon.

CPU data address watchpoints (read-only, write-only or read-write) are inserted using the `bp watch` command. Watchpoints can be setup to trigger within a range determined by a bit-mask where a one means that the address must match the address pattern and a zero mask indicate don’t care. The lowest 2-bits are not available, meaning that 32-bit words are the smallest address that can be watched. Byte accesses can still be watched but accesses to the neighboring three bytes will also be watched.

AMBA-bus watchpoints can be inserted using `bp bus` or `bp data`. When a bus watchpoint is hit the trace buffer will freeze. The processor can optionally be put in debug mode when the bus watchpoint is hit. This is controlled by the `tmode` command:

```
grmon3> tmode break N
```

If $N = 0$, the processor will not be halted when the watchpoint is hit. A value > 0 will break the processor and set the AHB trace buffer delay counter to the same value.

For hardware supported break/watchpoints the target must have been configured accordingly, otherwise a failure will be reported. Note also that the number of watchpoints implemented varies between designs.

3.4.5. Displaying processor registers

The registers of the processor can be displayed using the `reg` command (for a LEON processor it will print the current window).

For a LEON processor the other register windows can be displayed using `reg wN`, when N denotes the window number.

Example 3.1. LEON Registers

```
grmon3> reg
      INS      LOCALS      OUTS      GLOBALS
0:  00000008  0000000C  00000000  00000000
1:  80000070  00000020  00000000  00000001
2:  00000000  00000000  00000000  00000002
3:  00000000  00000000  00000000  00300003
```

```

4: 00000000 00000000 00000000 00040004
5: 00000000 00000000 00000000 00005005
6: 407FFFF0 00000000 407FFFF0 00000606
7: 00000000 00000000 00000000 00000077

psr: F34010E0  wim: 00000002  tbr: 40000060  y: 00000000

pc: 40003E44  be 0x40003FB8
npc: 40003E48  nop

```

Example 3.2. RISC-V Registers

```

grmon3> reg
a0: 0000000000000000    t0: 0000000000000000    s0: 0000000000000000
a1: 0000000000000000    t1: 0000000000000000    s1: 0000000000000000
a2: 0000000000000000    t2: 0000000000000000    s2: 0000000000000000
a3: 0000000000000000    t3: 0000000000000000    s3: 0000000000000000
a4: 0000000000000000    t4: 0000000000000000    s4: 0000000000000000
a5: 0000000000000000    t5: 0000000000000000    s5: 0000000000000000
a6: 0000000000000000    t6: 0000000000000000    s6: 0000000000000000
a7: 0000000000000000    s7: 0000000000000000
                                s8: 0000000000000000
                                s9: 0000000000000000
Machine mode                sp: 000000003FFFFFF0    s10: 0000000000000000
FPU dirty state            tp: 0000000000000000    s11: 0000000000000000
IRQ disabled                gp: 0000000000000000

ra: 0000000000000000
pc: 0000000000000000  nop

```

Individual registers can be accessed by providing the name of the register to the **reg** command. See command **reg wN** documentation for more information about supported register names.

The registers are also available as Tcl variables in the the Tcl `cpu` namespace that GRMON provides. GRMON exports `cpu` and `cpuN` namespaces, where *N* selects which CPU's registers are accessed, the `cpu` namespace points to the active CPU selected by the **cpu** command.

```

grmon3> puts [format %x $::cpu::iu::o6]
407ffff0

```

Use the **float** command to show the FPU registers (if present).

3.4.6. Backtracing function calls

When debugging an application it is often most useful to view how the CPU entered the current function. The **bt** command analyze the previous stack frames to determine the backtrace. GRMON reads the register windows and then switches to read from the stack depending on the `%WIM` and `%PSR` register.

The backtrace is presented with the caller's program counter (`%PC`) to return to (below where the `CALL` instruction was issued) and the stack pointer (`%SP`) at that time. The first entry (frame #0) indicates the current location of the CPU and the current stack pointer. The right most column print out the `%PC` address relative the function symbol, i.e. if symbols are present.

```

grmon3> bt
      %pc          %sp
#0  0x40003e24    0x407ffdb8  <Fft+0x4>
#1  0x40005034    0x407ffe28  <main+0xfc4>
#2  0x40001064    0x407fff70  <_start+0x64>
#3  0x4000cf40    0x407fffb0  <_hardreset_real+0x78>

```

On a LEON system, in order to display a correct backtrace for optimized code, where optimized leaf functions are present, a symbol table must exist.

NOEL-V requires DWARF information to display backtrace correctly

In a MP system the backtrace of a specific CPU can be printed, either by changing the active CPU with the **cpu** command or by passing the CPU index to **bt**.

3.4.7. Displaying memory contents

Any memory location can be displayed and written using the commands listed in the table below. Memory commands that are prefixed with a *v* access the virtual address space seen by doing MMU address lookups for active CPU.

Table 3.2. Memory access commands

Command Name	Description
mem	AMBA bus 32-bit memory read access, list a range of addresses
wmem	AMBA bus 32-bit memory write access
vmem	AMBA bus 32-bit virtual memory read access, list a range of addresses
memb	AMBA bus 8-bit memory read access, list a range of addresses
memh	AMBA bus 16-bit memory read access, list a range of addresses
vmemb	AMBA bus 8-bit virtual memory read access, list a range of addresses
vmemh	AMBA bus 16-bit virtual memory read access, list a range of addresses
vwmemb	AMBA bus 8-bit virtual memory write access
vwmemh	AMBA bus 16-bit virtual memory write access
vwmems	Write a string to an AMBA bus virtual memory address
vwmem	AMBA bus 32-bit virtual memory write access
wmemb	AMBA bus 8-bit memory write access
wmemh	AMBA bus 16-bit memory write access
wmems	Write a string to an AMBA bus memory address
amem	AMBA bus 32-bit asynchronous memory read access

Most debug links only support 32-bit accesses, only JTAG links support unaligned access. An unaligned access is when the address or number of bytes are not evenly divided by four. When an unaligned data read request is issued, then GRMON will read some extra bytes to align the data, but only return the requested data. If a write request is issued, then an aligned read-modify-write sequence will occur.

The **mem** command requires an address and an optional length, if the length is left out 64 bytes are displayed. If a program has been loaded, text symbols can be used instead of a numeric address. The memory content is displayed in hexadecimal-decimal format, grouped in 32-bit words. The ASCII equivalent is printed at the end of the line.

```
grmon> mem 0x40000000

40000000 a0100000 29100004 81c52000 01000000 ... ).....
40000010 91d02000 01000000 01000000 01000000 .
40000020 91d02000 01000000 01000000 01000000 .
40000030 91d02000 01000000 01000000 01000000 .

grmon> mem 0x40000000 16

40000000 a0100000 29100004 81c52000 01000000 ... ).....

grmon> mem main 48

40003278 9de3bf98 2f100085 31100037 90100000 .../...1..7...
40003288 d02620c0 d025e178 11100033 4000b4b & .%.x...3@..K
40003298 901223b0 11100033 4000af4 901223c0 ..#....3@.....#.
```

The memory access commands listed in Table 3.2 are not restricted to memory: they can be used on any bus address accessible by the debug link. However, for access to peripheral control registers, the command **info reg** can provide a more user-friendly output.

All commands in Table 3.2, , except for **amem**, return to the caller when the bus access has completed, which means that a sequence of these commands generates a sequence of bus accesses with the same ordering. In situations where the bus accesses order is not critical, the command **amem** can be used to schedule multiple concurrent read accesses whose results can be retrieved at a later time. This is useful when GRMON is automated using Tcl scripts.

3.4.8. Instruction disassembly

If the memory contents is machine code of the target processor, the contents can be displayed in assembly code using the **disassemble** command:

```

grmon3> disassemble 0x40000000 10
0x40000000: 88100000  clr  %g4          <start+0>
0x40000004: 09100034  sethi %hi(0x4000d000), %g4 <start+4>
0x40000008: 81c12034  jmp  %g4 + 0x34    <start+8>
0x4000000c: 01000000  nop                <start+12>
0x40000010: a1480000  mov  %psr, %l0     <start+16>
0x40000014: a7500000  mov  %wim, %l3     <start+20>
0x40000018: 10803401  ba   0x4000d01c   <start+24>
0x4000001c: ac102001  mov  l, %l6        <start+28>
0x40000020: 91d02000  ta   0x0           <start+32>
0x40000024: 01000000  nop                <start+36>

grmon3> dis main
0x40004070: 9de3beb8  save %sp, -328, %sp <main+0>
0x40004074: 15100035  sethi %hi(0x4000d400), %o2 <main+4>
0x40004078: d102a3f4  ld   [%o2 + 0x3f4], %f8 <main+8>
0x4000407c: 13100035  sethi %hi(0x4000d400), %o1 <main+12>
0x40004080: 39100088  sethi %hi(0x40022000), %i4 <main+16>
0x40004084: 3710003a  sethi %hi(0x4000e800), %i3 <main+20>
0x40004088: d126e2e0  st  %f8, [%i3 + 0x2e0] <main+24>
0x4000408c: d1272398  st  %f8, [%i4 + 0x398] <main+28>
0x40004090: 400006a9  call 0x40005b34    <main+32>
0x40004094: 901262f0  or   %o1, 0x2f0, %o0 <main+36>
0x40004098: 11100035  sethi %hi(0x4000d400), %o0 <main+40>
0x4000409c: 40000653  call 0x400059e8    <main+44>
0x400040a0: 90122300  or   %o0, 0x300, %o0 <main+48>
0x400040a4: 7ffff431  call 0x40001168    <main+52>
0x400040a8: 3510005b  sethi %hi(0x40016c00), %i2 <main+56>
0x400040ac: 2510005b  sethi %hi(0x40016c00), %i2 <main+60>

```

3.4.9. Using the trace buffer

The processor and associated debug support unit (DSU or RVDMM) can be configured with trace buffers to store both the latest executed instructions and the latest AHB bus transfers. The trace buffers are automatically enabled by GRMON during start-up, but can also be individually enabled and disabled using **tmode** command. The command **ahb** is used to show the AMBA buffer. The command **inst** is used to show the instruction buffer. The command **hist** is used to display the contents of the instruction and the AMBA buffers mixed together. Below is an example debug session that shows the usage of breakpoints, watchpoints and the trace buffer:

```

grmon3> lo v8/stanford.exe
40000000 .text          54.8kB / 54.8kB  [=====] 100%
4000DB30 .data           2.9kB / 2.9kB  [=====] 100%
Total size: 57.66kB (786.00kbit/s)
Entry point 0x40000000
Image /home/daniel/examples/v8/stanford.exe loaded

grmon3> bp Fft
Software breakpoint 1 at <Fft>

grmon3> bp watch 0x4000eae0
Hardware watchpoint 2 at 0x4000eae0

grmon3> bp
NUM  ADDRESS      MASK          TYPE          SYMBOL
  1  : 0x40003e20  (soft)       Fft+0
  2  : 0x4000eae0  0xffffffffc (watch rw)  floated+0

grmon3> run

CPU 0: watchpoint 2 hit
      0x40001024: c0388003  std  %g0, [%g2 + %g3] <_start+36>
CPU 1: Power down mode

grmon3> inst
TIME  ADDRESS  INSTRUCTION  RESULT
84675 40001024  std  %g0, [%g2 + %g3]  [4000eaf8 00000000 00000000]
84678 4000101c  subcc %g3, 8, %g3     [00000440]
84679 40001020  bge,a 0x4000101c     [00000448]
84682 40001024  std  %g0, [%g2 + %g3]  [4000eaf0 00000000 00000000]
84685 4000101c  subcc %g3, 8, %g3     [00000438]
84686 40001020  bge,a 0x4000101c     [00000440]
84689 40001024  std  %g0, [%g2 + %g3]  [4000eae8 00000000 00000000]
84692 4000101c  subcc %g3, 8, %g3     [00000430]
84693 40001020  bge,a 0x4000101c     [00000438]
84694 40001024  std  %g0, [%g2 + %g3]  [ TRAP ]

grmon3> ahb
TIME  ADDRESS  TYPE  D[31:0]  TRANS  SIZE  BURST  MST  LOCK  RESP  HIRQ
84664 4000eb08  write 00000000  2  2  1  0  0  0  0000
84667 4000eb0c  write 00000000  3  2  1  0  0  0  0000
84671 4000eb00  write 00000000  2  2  1  0  0  0  0000

```

```

84674 4000eb04 write 00000000 3 2 1 0 0 0 0000
84678 4000eaf8 write 00000000 2 2 1 0 0 0 0000
84681 4000eafc write 00000000 3 2 1 0 0 0 0000
84685 4000eaf0 write 00000000 2 2 1 0 0 0 0000
84688 4000eaf4 write 00000000 3 2 1 0 0 0 0000
84692 4000eae8 write 00000000 2 2 1 0 0 0 0000
84695 4000eaec write 00000000 3 2 1 0 0 0 0000

```

```

grmon3> reg
INS          LOCALS          OUTS          GLOBALS
0: 80000200  00000000  00000000  00000000
1: 80000200  00000000  00000000  00000000
2: 0000000c  00000000  00000000  4000E6B0
3: FFF00000  00000000  00000000  00000430
4: 00000002  00000000  00000000  4000CC00
5: 800FF010  00000000  00000000  4000E680
6: 407FFFb0  00000000  407FFF70  4000CF34
7: 4000CF40  00000000  00000000  00000000

psr: F30010E7  wim: 00000002  tbr: 40000000  y: 00000000

pc: 40001024  std  %g0, [%g2 + %g3]
npc: 4000101c  subcc %g3, 8, %g3

```

```
grmon3> bp del 2
```

```

grmon3> cont
Towers  Queens  Intmm      Mm  Puzzle  Quick  Bubble  Tree  FFT
CPU 0:  breakpoint 1 hit
        0x40003e24: a0100018  mov  %i0, %l0 <Fft+4>
CPU 1:  Power down mode

```

```

grmon3>
grmon3> hist
TIME      ADDRESS  INSTRUCTIONS/AHB SIGNALS  RESULT/DATA
30046975  40003e20  AHB read  mst=0  size=2  [9de3bf90]
30046976  40005030  or  %l2, 0x1e0, %o3  [40023de0]
30046980  40003e24  AHB read  mst=0  size=2  [91d02001]
30046981  40005034  call 0x40003e20  [40005034]
30046985  40003e28  AHB read  mst=0  size=2  [b136201f]
30046990  40003e2c  AHB read  mst=0  size=2  [f83fbff0]
30046995  40003e30  AHB read  mst=0  size=2  [82040018]
30047000  40003e34  AHB read  mst=0  size=2  [d11fbff0]
30047005  40003e38  AHB read  mst=0  size=2  [9a100019]
30047010  40003e3c  AHB read  mst=0  size=2  [9610001a]

```

When printing executed instructions, the value within brackets denotes the instruction result, or in the case of store instructions the store address and store data. The value in the first column displays the relative time, equal to the DSU timer. The time is taken when the instruction completes in the last pipeline stage (write-back) of the processor. In a mixed instruction/AHB display, AHB address and read or write value appears within brackets. The time indicates when the transfer completed, i.e. when HREADY was asserted.

As the AHB trace is disabled when a breakpoint is hit, AHB accesses related to instruction cache fetches after the time of break can be missed. The command **ahb force** can be used enable AHB tracing even when the processor is in debug mode.

When switching between tracing modes with **tmode** the contents of the trace buffer will not be valid until execution has been resumed and the buffer refilled.

3.4.10. Profiling

GRMON supports profiling of target applications when run on real hardware. The profiling function collects (statistical) information on the amount of execution time spent in each function. Due to its non-intrusive nature, the profiling data does not take into consideration if the current function is called from within another procedure. Even so, it still provides useful information and can be used for application tuning.

To increase the number of samples, use the fastest debug link available on the target system. I.a. do not use I/O forwarding (start GRMON *without* the -u commandline option)

```

grmon3> lo v8/stanford.exe
40000000 .text 54.8kB / 54.8kB [=====] 100%
4000DB30 .data 2.9kB / 2.9kB [=====] 100%
Total size: 57.66kB (786.00kbit/s)
Entry point 0x40000000

```

```

Image /home/daniel/examples/v8/stanford.exe loaded

grmon3> profile on

grmon3> run
Starting
  Perm Towers Queens Intmm      Mm Puzzle Quick Bubble Tree FFT

CPU 0: Interrupted!
      0x40003ee4: 95a0c8a4 fsubs %f3, %f4, %f10 <Fft+196>
CPU 1: Interrupted!
      0x40000000: 88100000 clr %g4 <start+0>

grmon3> prof
FUNCTION          SAMPLES  RATIO(%)
Trial             000000096 27.35
__window_overflow_rettseq_ret 000000060 17.09
main              000000051 14.52
__window_overflow_slow1 000000026 7.40
Fft               000000023 6.55
Insert            000000016 4.55
Permute           000000013 3.70
tower             000000013 3.70
Try               000000013 3.70
Quicksort         000000011 3.13
Checktree         000000007 1.99
__malloc_r        000000005 1.42
start             000000004 1.13
outbyte           000000003 0.85
Towers            000000002 0.56
__window_overflow_rettseq 000000002 0.56
__st_pthread_mutex_lock 000000002 0.56
_start           000000001 0.28
Perm              000000001 0.28
__malloc_lock     000000001 0.28
__st_pthread_mutex_trylock 000000001 0.28

```

3.4.11. Attaching to a target system without initialization

When GRMON connects to a target system, it probes the configuration and initializes memory and registers. To determine why a target has crashed, or to debug an application that is running, it might be desirable to connect to the target without performing a (destructive) initialization. This can be done by specifying the `-ni` switch during the start-up of GRMON. The CPUs cores still be forced into debug mode during GRMON startup.

The system information print-out (**info sys**) will not be able to display information correctly that depends on initialization, for example the correct memory settings.

To continue the execution of the application, issue the **cont** or the **detach** command.

The **run** and **reset** commands may not have the intended effect since the debug drivers have not been initialized during start-up.

If the user runs a software bootloader or initializes the system manually, then **go** command can be used to restart the application. Otherwise it recommended to restart GRMON without `-ni` to restart the application.

3.4.12. Attaching to a target system without Plug and Play scanning

When GRMON connects to a target system, it will scan the Plug and play information (or read a configuration file), to determine which kind of system it has connected to. If GRMON is started with the `-nopnp` switch, then this will be skipped, and GRMON will not have any knowledge about the hardware. Therefore GRMON will not make any access to the system during startup.

Most hardware dependent commands will have undefined behaviour. But the basic memory access commands will work, i.e. **mem**, **wmem** and similar.

This mode can be used to check register values after a hardware reset has been asserted, or other situations when you want stop GRMON from doing any accesses during startup.

3.4.13. Multi-processor support

In systems with more than one processor, the **cpu** command can be used to control the state and debugging focus of the processors. In MP systems, the processors are enumerated with 0..N-1, where *N* is the number of processors.

Each processor can be in two states; enabled or disabled. When enabled, a processor can be started by LEON software or by GRMON. When disabled, the processor will remain halted regardless. One can pause a MP operating system and disable a CPU to debug a hanged CPU for example.

Most per-CPU debugging commands such as displaying registers, backtrace or adding breakpoints will be directed to the active processor only. Switching active processor can be done using the '**cpu active N**' command, see example below. The Tcl `cpu` namespace exported by GRMON is also changed to point to the active CPU's namespace, thus accessing `cpu` will be the same as accessing `cpu1` if CPU1 is the currently active CPU.

```
grmon3> cpu
cpu 0: enabled active
cpu 1: enabled

grmon3> cpu act 1

grmon3> cpu
cpu 0: enabled
cpu 1: enabled active

grmon3> cpu act 0

grmon3> cpu dis 1

grmon3> cpu
cpu 0: enabled active
cpu 1: disabled

grmon3> puts $cpu::fpu::f1
-1.984328031539917

grmon3> puts $cpu0::fpu::f1
-1.984328031539917

grmon3> puts $cpu1::fpu::f1
2.3017966689845248e+18
```

Non-MP software can still run on the first CPU unaffected of the additional CPUs since it is the target software that is responsible for waking other CPUs. All processors are enabled by default.

Note that it is possible to debug MP systems using GDB, but the user are required to change CPU itself. GRMON specific commands can be entered from GDB using the **monitor** command.

3.4.14. Stack and entry point

The stack pointer is located in %O6 (%SP) register of SPARC CPUs. GRMON sets the stack pointer before starting the CPU with the **run** command. The address is auto-detected to end of main memory, however it is overridable using the `-stack` when starting GRMON or by issuing the **stack** command. Thus stack pointer can be used by software to detect end of main memory.

The entry point (EP) determines at which address the CPU start its first instruction execution. The EP defaults to main memory start and normally overridden by the **load** command when loading the application. ELF-files has support for storing entry point. The entry point can manually be set with the **ep** command.

In a MP systems if may be required to set EP and stack pointer individual per CPU, one can use the **cpu** command in conjunction with **ep** and **stack**.

3.4.15. Memory Management Unit (MMU) support

The target processor may optionally implements an MMU. GRMON has support for the reference MMU (SRM-MU) described in the SPARCv8 specification. It also support the RISC-V Sv32, Sv39 and Sv48 schemes. GRMON support viewing and changing the MMU registers through the DSU, using the **mmu** command. GRMON also supports address translation by reading the MMU table from memory similar to the MMU. The **walk** command looks up one address by walking the MMU table printing out every step taken and the result. To simply print out the result of such a translation, use the **va** command.

The memory commands that are prefixed with a `v` work with virtual addresses, the addresses given are translated before listing or writing physical memory. If the MMU is not enabled, the **vmem** command for example is an alias for **mem**. See Section 3.4.7, “Displaying memory contents” for more information.

Many commands are affected by that the MMU is turned on, such as the **disassemble** command.

3.4.16. CPU cache support

The target system optionally implements Level-1 instruction-cache and data-cache. GRMON supports the CPU's cache by adopting certain operations depending on if the cache is activated or not. The user may also be able to access the cache directly. This is however not normally needed, but may be useful when debugging or analyzing different cache aspects. By default the L1-cache is turned on by GRMON, the **cctrl** command can be used to change the cache control register. The commandline switches **-nic** and **-ndc** disables instruction and data cache respectively.

With the **icache** and **dcache** commands it is possible to view and modify the current content of the cache or check if the cache is consistent with the memory. Both caches can be flushed instantly using the commands **cctrl flush**. The data cache can be flushed instantly using the commands **dcache flush**. The instruction cache can be flushed instantly using the commands **icache flush**.

The GRLIB Level-2 cache is supported using the **l2cache** command.

3.5. Tcl integration

GRMON has built-in support for Tcl 8.6. All commands lines entered in the terminal will pass through a Tcl-interpreter. This enables loops, variables, procedures, scripts, arithmetic and more for the user. I.a. it also provides an API for the user to extend GRMON.

3.5.1. Shells

GRMON creates several independent TCL shells, each with its own set of commands and variables. I.e. changing active CPU in one shell does not affect any other shell. In the commandline version there one shell available for the user by default, the CLI shell, which is accessed from the terminal. In the GUI is possible to create and view multiple shells.

Additional custom user shells for the commandline interface can be created with the command **usrsh**. Each custom user shell has an associated Tcl interpreter running in a separate execution thread.

When the GDB service is running, a GDB shell is also available from GDB by using the command **mon**.

There is also a system shell and an execution shell running in the background that GRMON uses internally. Some hooks must be loaded into these shells to work, see Appendix C, *Tcl API* for more information.

3.5.2. Commands

There are two groups of commands, the native Tcl commands and GRMON's commands. Information about the native Tcl commands and their syntax can be found at the Tcl website [<http://www.tcl.tk/>]. The GRMON commands' syntax documentation can be found in Appendix B, *Command syntax*.

The commands have three types of output:

1. **Standard output.** GRMON's commands prints information to standard output. This information is often structured in a human readable way and cannot be used by other commands. Most of the GRMON commands print some kind of information to the standard output, while very few of the Tcl commands does that. Setting the variable `::grmon::settings::suppress_output` to 1 will stop GRMON commands from printing to the standard output, i.e. the TCL command **puts** will still print it's output. It is also possible to put the command **silent** in front of another GRMON command to suppress the output of a single command, e.g. `grmon3> puts [expr [silent mem 0x40000000 4] + 4]`
2. **Return values.** The return value from GRMON is seldom the same as the information that is printed to standard output, it's often the important data in a raw format. Return values can be used as input to other commands or to be saved in variables. All TCL commands and many GRMON commands have return values. The return values from commands are normally not printed. To print the return value to standard output one can use the Tcl command **puts**. I.a. if the variable `::grmon::settings::echo_result` to 1, then GRMON will always print the result to stdout.
3. **Return code.** The return code from a command can be accessed by reading the variable `errorCode` or by using the Tcl command **catch**. Both Tcl and GRMON commands will have an error message as return

value if it fails, which is also printed to standard output. More about error codes can be read about in the Tcl tutorial or on the Teler's Wiki [<http://wiki.tcl.tk/>].

For some of the GRMON commands it is possible to specify which core the commands is operation on. This is implemented differently depending for each command, see the commands' syntax documentation in Appendix B, *Command syntax* for more details. Some of these commands use a device name to specify which core to interact with, see Appendix C, *Tcl API* for more information about device names.

3.5.3. API

It is possible to extend GRMON using Tcl. GRMON provides an API that makes it possible do write own device drivers, implement hooks and to write advanced commands. See Appendix C, *Tcl API* for a detailed description of the API.

3.6. Symbolic debug information

GRMON will automatically extract the symbol information from ELF-files, debug information is never read from ELF-files. The symbols can be used to GRMON commands where an address is expected as below. Symbols are tab completed.

```
grmon3> load v8/stanford.exe
40000000 .text          54.8kB /  54.8kB  [=====] 100%
4000DB30 .data         2.9kB /   2.9kB  [=====] 100%
Image /home/daniel/examples/v8/stanford.exe loaded

grmon3> bp main
Software breakpoint 1 at <main>

grmon3> dis strlen 5
0x40005b88: 808a2003 andcc %o0, 0x3, %g0      <strlen+0>
0x40005b8c: 12800012 bne 0x40005BD4      <strlen+4>
0x40005b90: 94100008 mov %o0, %o2          <strlen+8>
0x40005b94: 033fbfbf sethi %hi(0xFEFEFC00), %g1 <strlen+12>
0x40005b98: da020000 ld [%o0], %o5       <strlen+16>
```

The **symbols** command can be used to display all symbols, lookup the address of a symbol, or to read in symbols from an alternate (ELF) file:

```
grmon3> symbols load v8/stanford.exe

grmon3> symbols lookup main
Found address 0x40004070

grmon3> symbols list
0x40005ab8 GLOBAL FUNC putchar
0x4000b6ac GLOBAL FUNC __mprec_log10
0x4000d9d0 GLOBAL OBJECT __mprec_tinytens
0x4000bbe8 GLOBAL FUNC cleanup_glue
0x4000abfc GLOBAL FUNC _hi0bits
0x40005ad4 GLOBAL FUNC _puts_r
0x4000c310 GLOBAL FUNC _lseek_r
0x4000eaac GLOBAL OBJECT piecemax
0x40001aac GLOBAL FUNC Try
0x40003c6c GLOBAL FUNC Uniform11
0x400059e8 GLOBAL FUNC printf
...
```

Reading symbols from alternate files is necessary when debugging self-extracting applications (MKPROM), when switching between virtual and physical address space (Linux) or when debugging a multi-core ASMP system where each CPU has its own symbol table. It is recommended to clear old symbols with **symbols clear** before switching symbol table, otherwise the new symbols will be added to the old table.

3.6.1. Multi-processor symbolic debug information

When loading symbols into GRMON it is possible to associate them with a CPU. When all symbols/images are associated with CPU index 0, then GRMON will assume its a single-core or SMP application and lookup all symbols from the symbols table associated with CPU index 0.

If different CPU indexes are specified (by setting active CPU or adding `cpu#` argument to the commands) when loading symbols/images, then GRMON will assume its an AMP application that has been loaded. GRMON will use the current active CPU (or `cpu#` argument) to determine which CPU index to lookup symbols from.

```
grmon3> cpu active 1
```

```

grmon3> symbols ../tests/threads/rtems-mp2
Loaded 1630 symbols

grmon3> bp _Thread_Handler
Software breakpoint 1 at <_Thread_Handler>

grmon3> symbols ../tests/threads/rtems-mp1 cpu0
Loaded 1630 symbols

grmon3> bp _Thread_Handler cpu0
Software breakpoint 2 at <_Thread_Handler>

grmon3> bp
NUM ADDRESS MASK TYPE CPU SYMBOL
1 : 0x40418408 (soft) 1 _Thread_Handler+0
2 : 0x40019408 (soft) 0 _Thread_Handler+0

```

3.7. GDB interface

This section describes the GDB interface support available in GRMON. GRMON supports GDB version 6.3, 6.4, 6.8 and 8.2. Other tools that communicate over the GDB protocol may also attach to GRMON, some tools such as Eclipse Workbench and DDD communicate with GRMON via GDB.

GDB must be built for the target architecture, a native PC GDB does not work together with GRMON. The toolchains that we distributes comes with a patched and tested version of GDB.

Please see the GDB documentation available from the official GDB homepage [<http://www.gnu.org/software/gdb/>].

3.7.1. Connecting GDB to GRMON

GRMON can act as a remote target for GDB, allowing symbolic debugging of target applications. To initiate GDB communications, start the monitor with the `-gdb` switch or use the GRMON `gdb start` command:

```

$ grmon -gdb
...
Started GDB service on port 2222.
...
grmon3> gdb status
GDB Service is waiting for incoming connection
Port: 2222

```

Then, start GDB in a different window and connect to GRMON using the extended-remote protocol. By default, GRMON listens on port 2222 for the GDB connection:

```

$ sparc-gaisler-elf-gdb /opt/bcc-2.0.7-rc.1-gcc/src/examples/stanford/stanford
GNU gdb (GDB) 8.2
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=sparc-gaisler-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /opt/bcc-2.0.7-rc.1-gcc/src/examples/stanford/stanford...done.
(gdb) target extended-remote :2222
Remote debugging using :2222
__bcc_entry_point () at /opt/bcc-2.0.7-rc.1-gcc/src/libbcc/shared/trap/trap_table_mvt.S:81
81 RESET_TRAP(__bcc_trap_reset_mvt);          ! 00 reset
(gdb)

```

3.7.2. Executing GRMON commands from GDB

While GDB is attached to GRMON, most GRMON commands can be executed using the GDB monitor command. Output from the GRMON commands is then displayed in the GDB console like below. Some DSU commands are naturally not available since they would conflict with GDB. All commands executed from GDB are executed in a separate Tcl interpreter, thus variables created from GDB will not be available from the GRMON terminal.

```
(gdb) monitor hist
```

TIME	ADDRESS	INSTRUCTIONS/AHB SIGNALS	RESULT/DATA
30046975	40003e20	AHB read mst=0 size=2	[9de3bf90]
30046976	40005030	or %l2, 0x1e0, %o3	[40023de0]
30046980	40003e24	AHB read mst=0 size=2	[91d02001]
30046981	40005034	call 0x40003e20	[40005034]
30046985	40003e28	AHB read mst=0 size=2	[b136201f]
30046990	40003e2c	AHB read mst=0 size=2	[f83fbff0]
30046995	40003e30	AHB read mst=0 size=2	[82040018]
30047000	40003e34	AHB read mst=0 size=2	[d11fbff0]
30047005	40003e38	AHB read mst=0 size=2	[9a100019]
30047010	40003e3c	AHB read mst=0 size=2	[9610001a]

(gdb)

3.7.3. Running applications from GDB

To load and start an application, use the GDB **load** and **run** command. When using the GDB **load** command to upload an application, one should also call **mon gdb postload** to ensure that GRMON initializes the system properly.

```
$ sparc-rtems-gdb v8/stanford.exe
(gdb) target extended-remote :2222
Remote debugging using :2222
main () at stanford.c:1033
1033 {
(gdb) mon gdb reset
(gdb) load
(gdb) mon gdb postload
Loading section .text, size 0xdb30 lma 0x40000000
Loading section .data, size 0xb78 lma 0x4000db30
Start address 0x40000000, load size 59048
Transfer rate: 18 KB/sec, 757 bytes/write.
(gdb) b main
Breakpoint 1 at 0x40004074: file stanford.c, line 1033.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/daniel/examples/v8/stanford.exe

Breakpoint 1, main () at stanford.c:1033
1033 {
(gdb) list
1028     /* Printcomplex( 6, 99, z, 1, 256, 17 ); */
1029     };
1030 } /* oscar */ ;
1031
1032 main ()
1033 {
1034     int i;
1035     fixed = 0.0;
1036     floated = 0.0;
1037     printf ("Starting \n");
(gdb)
```

To interrupt execution, Ctrl-C can be typed in GDB terminal (similar to GRMON). The program can be restarted using the GDB **run** command but the program image needs to be reloaded first using the **load** command. Software trap 1 (TA 0x1) is used by GDB to insert breakpoints and should not be used by the application.

GRMON translates SPARC traps, or RISC-V exceptions, into (UNIX) signals which are properly communicated to GDB. If the application encounters a fatal trap, execution will be stopped exactly before the failing instruction. The target memory and register values can then be examined in GDB to determine the error cause.

GRMON implements the GDB breakpoint and watchpoint interface and makes sure that memory and cache are synchronized.

3.7.4. Running SMP applications from GDB

If GRMON is running on the same computer as GDB, or if the executable is available on the remote computer that is running GRMON, it is recommended to issue the GDB command **set remote exec-file <remote-file-path>**. After this has been set, GRMON will automatically load the file, and symbols if available, when the GDB command **run** is issued.

```
$ sparc-rtems-gdb /opt/rtems-4.11/src/rtems-4.11/testsuites/libtests/ticker/ticker.exe
GNU gdb 6.8.0.20090916-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
```

```

This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-rtems"...
(gdb) target extended-remote :2222
Remote debugging using :2222
0x00000000 in ?? ()
(gdb) set remote exec-file /opt/rtems-4.11/src/rtems-4.11/testsuites/libtests/ticker/ticker.exe
(gdb) break Init
Breakpoint 1 at 0x40001318: file ../../../../../../leon3smp/lib/include/rtems/score/thread.h, line 627.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/rtems-4.11/src/rtems-4.11/testsuites/libtests/ticker/ticker.exe

```

If the executable is not available on the remote computer where GRMON is running, then the GDB command **load** can be used to load the software to the target system.

```

$ sparc-rtems-gdb /opt/rtems-4.11/src/rtems-4.11/testsuites/libtests/ticker/ticker.exe
GNU gdb 6.8.0.20090916-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-rtems"...
(gdb) target extended-remote :2222
Remote debugging using :2222
trap_table () at /opt/rtems-4.11/src/rtems-4.11/c/src/lib/libbsp/sparc/leon3/../../../../sparc/shared/start
/start.S:69
69 /opt/rtems-4.11/src/rtems-4.11/c/src/lib/libbsp/sparc/leon3/../../../../sparc/shared/start/start.S: No
such file or directory.
in /opt/rtems-4.11/src/rtems-4.11/c/src/lib/libbsp/sparc/leon3/../../../../sparc/shared/start/start.S
Current language: auto; currently asm
(gdb) mon gdb reset
(gdb) load
(gdb) mon gdb postload
Loading section .text, size 0x1aed0 lma 0x40000000
Loading section .data, size 0x5b0 lma 0x4001aed0
Start address 0x40000000, load size 111744
Transfer rate: 138 KB/sec, 765 bytes/write.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/rtems-4.11/src/rtems-4.11/testsuites/libtests/ticker/ticker.exe

```

3.7.5. Running AMP applications from GDB

If GRMON is running on the same computer as GDB, or if the executables are available on the remote computer that is running GRMON, it is recommended to issue the GDB command **set remote exec-file <remote-file-path>**. When this is set, GRMON will automatically load the file, and symbols if available, when the GDB command **run** is issued. The second application needs to be loaded into GRMON using the GRMON command **load <remote-file-path> cpu1**. In addition the stacks must also be set manually in GRMON using the command **stack <address> cpu#** for both CPUs.

```

$ sparc-rtems-gdb /opt/rtems-4.10/src/samples/rtems-mp1
GNU gdb 6.8.0.20090916-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-rtems"...
(gdb) target extended-remote :2222
Remote debugging using :2222
(gdb) set remote exec-file /opt/rtems-4.10/src/samples/rtems-mp1
(gdb) mon stack 0x403fff00 cpu0
CPU 0 stack pointer: 0x403fff00
(gdb) mon load /opt/rtems-4.10/src/samples/rtems-mp2 cpu1
Total size: 177.33kB (1.17Mbit/s)
Entry point 0x40400000
Image /opt/rtems-4.10/src/samples/rtems-mp2 loaded
(gdb) mon stack 0x407fff00 cpu1
CPU 1 stack pointer: 0x407fff00
(gdb) run
Starting program: /opt/rtems-4.10/src/samples/rtems-mp1
NODE[0]: is Up!
NODE[0]: Waiting for Semaphore A to be created (0x53454d41)
NODE[0]: Waiting for Semaphore B to be created (0x53454d42)
NODE[0]: Waiting for Task A to be created (0x54534b41)
^C[New Thread 151060481]

Program received signal SIGINT, Interrupt.
[Switching to Thread 151060481]

```

```

pwdloop () at /opt/rtems-4.10/src/rtems-4.10/c/src/lib/libbsp/sparc/leon3/startup/bspidle.S:26
warning: Source file is more recent than executable.
26      retl
Current language: auto; currently asm
(gdb)

```

If the executable is not available on the remote computer where GRMON is running, then the GDB command **file** and **load** can be used to load the software to the target system. Use the GRMON command **cpu act <num>** before issuing the GDB command **load** to specify which CPU is the target for the software being loaded. In addition the stacks must also be set manually in GRMON using the command **stack <address> cpu#** for both CPUs.

```

$ sparc-rtems-gdb
GNU gdb 6.8.0.20090916-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-rtems".
(gdb) target extended-remote :2222
Remote debugging using :2222
0x40000000 in ?? ()
(gdb) mon gdb reset
(gdb) file /opt/rtems-4.10/src/samples/rtems-mp2
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Reading symbols from /opt/rtems-4.10/src/samples/rtems-mp2...done.
(gdb) mon cpu act 1
(gdb) load
Loading section .text, size 0x2b3e0 lma 0x40400000
Loading section .data, size 0x1170 lma 0x4042b3e0
Loading section .jcr, size 0x4 lma 0x4042c550
Start address 0x40400000, load size 181588
Transfer rate: 115 KB/sec, 759 bytes/write.
(gdb) file /opt/rtems-4.10/src/samples/rtems-mp1
A program is being debugged already.
Are you sure you want to change the file? (y or n) y

Load new symbol table from "/opt/rtems-4.10/src/samples/rtems-mp1"? (y or n) y
Reading symbols from /opt/rtems-4.10/src/samples/rtems-mp1...done.
(gdb) mon cpu act 0
(gdb) load
Loading section .text, size 0x2b3e0 lma 0x40001000
Loading section .data, size 0x1170 lma 0x4002c3e0
Loading section .jcr, size 0x4 lma 0x4002d550
Start address 0x40001000, load size 181588
Transfer rate: 117 KB/sec, 759 bytes/write.
(gdb) mon stack 0x407fff00 cpu1
CPU 1 stack pointer: 0x407fff00
(gdb) mon stack 0x403fff00 cpu0
CPU 0 stack pointer: 0x403fff00
(gdb) mon gdb postload
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /opt/rtems-4.10/src/samples/samples/rtems-mp1

```

3.7.6. GDB Thread support

GDB is capable of listing a operating system's threads, however it relies on GRMON to implement low-level thread access. GDB normally fetches the threading information on every stop, for example after a breakpoint is reached or between single-stepping stops. GRMON have to access the memory rather many times to retrieve the information, GRMON. See Section 3.8, "Thread support" for more information.

Start GRMON with the **-nothreads** switch to disable threads in GRMON and thus in GDB too.

Note that GRMON must have access to the symbol table of the operating system so that the thread structures of the target OS can be found. The symbol table can be loaded from GDB by one must bear in mind that the path is relative to where GRMON has been started. If GDB is connected to GRMON over the network one must make the symbol file available on the remote computer running GRMON.

```

(gdb) mon puts [pwd]
/home/daniel
(gdb) pwd
Working directory /home/daniel.
(gdb) mon sym load /opt/rtems-4.10/src/samples/rtems-hello
(gdb) mon sym
0x00016910 GLOBAL FUNC      imfs_dir_lseek

```

```
0x00021f00 GLOBAL OBJECT Device_drivers
0x0001c6b4 GLOBAL FUNC _mprec_log10
...
```

When a program running in GDB stops GRMON reports which thread it is in. The command **info threads** can be used in GDB to list all known threads, **thread N** to switch to thread *N* and **bt** to list the backtrace of the selected thread.

```
Program received signal SIGINT, Interrupt.
[Switching to Thread 167837703]
```

```
0x40001b5c in console_outbyte_polled (port=0, ch=113 `q`) at rtems/.../leon3/console/debugputs.c:38
38 while ((LEON3_Console_Uart[LEON3_Cpu_Index+port]->status & LEON_REG_UART_STATUS_THE) == 0);
```

```
(gdb) info threads
```

```
8 Thread 167837702 (FTPD Wevnt) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
7 Thread 167837701 (FTPa Wevnt) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
6 Thread 167837700 (DCTX Wevnt) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
5 Thread 167837699 (DCRX Wevnt) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
4 Thread 167837698 (ntwk ready) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
3 Thread 167837697 (UI1 ready) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
2 Thread 151060481 (Int. ready) 0x4002f760 in _Thread_Dispatch () at rtems/.../threaddispatch.c:109
* 1 Thread 167837703 (HTPD ready) 0x40001b5c in console_outbyte_polled (port=0, ch=113 `q`)
  at .../rtems/c/src/lib/libbsp/sparc/leon3/console/debugputs.c:38
(gdb) thread 8
```

```
[Switching to thread 8 (Thread 167837702)]#0 0x4002f760 in _Thread_Dispatch ()
at rtems/.../threaddispatch.c:109
```

```
109 _Context_Switch( &executing->Registers, &heir->Registers );
```

```
(gdb) bt
```

```
#0 0x4002f760 in _Thread_Dispatch () at rtems/cpukit/score/src/threaddispatch.c:109
#1 0x40013ee0 in rtems_event_receive(event_in=33554432, option_set=0, ticks=0, event_out=0x43fecc14)
at .../leon3/lib/include/rtems/score/thread.inl:205
#2 0x4002782c in rtems_bsdnet_event_receive (event_in=33554432, option_set=2, ticks=0,
event_out=0x43fecc14) at rtems/cpukit/libnetworking/rtems/rtems_glue.c:641
#3 0x40027548 in soconnsleep (so=0x43f0cd70) at rtems/cpukit/libnetworking/rtems/rtems_glue.c:465
#4 0x40029118 in accept (s=3, name=0x43feccf0, namelen=0x43feccec) at rtems/.../rtems_syscall.c:215
#5 0x40004028 in daemon () at rtems/c/src/libnetworking/rtems_servers/ftpd.c:1925
#6 0x40053388 in _Thread_Handler () at rtems/cpukit/score/src/threadhandler.c:123
#7 0x40053270 in __res_mkquery (op=0, dname=0x0, class=0, type=0, data=0x0, datalen=0, newrr_in=0x0,
buf=0x0, buflen=0)
at .../rtems/cpukit/libnetworking/libc/res_mkquery.c:199
#8 0x00000008 in ?? ()
#9 0x00000008 in ?? ()
Previous frame identical to this frame (corrupt stack?)
```

In comparison to GRMON the **frame** command in GDB can be used to select a individual stack frame. One can also step between frames by issuing the **up** or **down** commands. The CPU registers can be listed using the **info registers** command. Note that the **info registers** command only can see the following registers for an inactive task: g0-g7, i0-i7, o0-o7, PC and PSR. The other registers will be displayed as 0:

```
(gdb) frame 5
```

```
#5 0x40004028 in daemon () at rtems/.../rtems_servers/ftpd.c:1925
1925 ss = accept(s, (struct sockaddr *)&addr, &addrLen);
```

```
(gdb) info reg
```

```
g0          0x0          0
g1          0x0          0
g2          0xffffffff   -1
g3          0x0          0
g4          0x0          0
g5          0x0          0
g6          0x0          0
g7          0x0          0
o0          0x3          3
o1          0x43feccf0   1140772080
o2          0x43feccec   1140772076
o3          0x0          0
o4          0xf3400e4    -213909276
o5          0x4007cc00   1074252800
sp          0x43fecc88     0x43fecc88
o7          0x40004020   1073758240
i0          0x4007ce88   1074253448
i1          0x4007ce88   1074253448
i2          0x400048fc   1073760508
i3          0x43feccf0   1140772080
i4          0x3          3
i5          0x1          1
i6          0x0          0
```

```

17          0x0      0
i0          0x0      0
i1          0x40003f94    1073758100
i2          0x0      0
i3          0x43ffa8c8    1140830152
i4          0x0      0
i5          0x4007cd40    1074253120
fp          0x43fec808    0x43fec808
i7          0x40053380    1074082688
y           0x0      0
psr         0xf34000e0    -213909280
wim         0x0      0
tbr         0x0      0
pc          0x40004028    0x40004028 <daemon+148>
npc         0x4000402c    0x4000402c <daemon+152>
fsr         0x0      0
csr         0x0      0

```

It is not supported to set thread specific breakpoints. All breakpoints are global and stops the execution of all threads. It is not possible to change the value of registers other than those of the current thread.

3.7.7. Virtual memory

There is no way for GRMON to determine if an address sent from GDB is physical or virtual. If an MMU unit is present in the system and it is enabled, then GRMON will assume that all addresses are virtual and try to translate them. When debugging an application that uses the MMU one typically have an image with physical addresses used to load data into the memory and a second image with debug-symbols of virtual addresses. It is therefore important to make sure that the MMU is enabled/disabled when each image is used.

The example below will show a typical case on how to handle virtual and physical addresses when debugging with GDB. The application being debugged is Linux and it consists of two different images created with Linuxbuild. The file `image.ram` contains physical addresses and a small loader, that among others configures the MMU, while the file `image` contains all the debug-symbols in virtual address-space.

First start GRMON and start the GDB server.

```
$ grmon -nb -gdb
```

Then start GDB in a second shell, load both files into GDB, connect to GRMON and then upload the application into the system. The addresses will be interpreted as physical since the MMU is disabled when GRMON starts.

```

$ sparc-linux-gdb
GNU gdb 6.8.0.20090916-cvs
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=sparc-linux".
(gdb) file output/images/image.ram
Reading symbols from /home/user/linuxbuild-1.0.2/output/images/image.ram...(no d
ebugging symbols found)...done.
(gdb) symbol-file output/images/image
Reading symbols from /home/user/linuxbuild-1.0.2/output/images/image...done.
(gdb) target extended-remote :2222
Remote debugging using :2222
t_tflt () at /home/user/linuxbuild-1.0.2/linux/linux-2.6-git/arch/sparc/kernel/h
ead_32.S:88
88 t_tflt: SPARC_TFAULT          /* Inst. Access Exception
*/
Current language: auto; currently asm
(gdb) mon gdb reset
(gdb) load
(gdb) monitor gdb postload
Loading section .text, size 0x10b0 lma 0x40000000
Loading section .data, size 0x50 lma 0x400010b0
Loading section .vmlinux, size 0x3f1a60 lma 0x40004000
Loading section .startup_prom, size 0x7ee0 lma 0x403f5a60
Start address 0x40000000, load size 4172352
Transfer rate: 18 KB/sec, 765 bytes/write.

```

The program must reach a state where the MMU is enabled before any virtual address can be translated. Software breakpoints cannot be used since the MMU is still disabled and GRMON won't translate them into a physical. Hardware breakpoints don't need to be translated into physical addresses, therefore set a hardware assisted breakpoint at `0xf0004000`, which is the virtual start address for the Linux kernel.


```
(gdb) hbreak *0xf0004000
Hardware assisted breakpoint 1 at 0xf0004000: file /home/user/linuxbuild-1.0.2/linu
x/linux-2.6-git/arch/sparc/kernel/head_32.S, line 87.
(gdb) cont
Continuing.
```

```
Breakpoint 1, trapbase_cpu0 () at /home/user/linuxbuild-1.0.2/linux/linux-2.6-gi
t/arch/sparc/kernel/head_32.S:87
87 t_zero: b gokernel; nop; nop; nop;
```

At this point the loader has enabled the MMU and both software breakpoints and symbols can be used.

```
(gdb) break leon_init_timers
Breakpoint 2 at 0xf03cff14: file /home/user/linuxbuild-1.0.2/linux/linux-2.6-gi
t/arch/sparc/kernel/leon_kernel.c, line 116.
```

```
(gdb) cont
Continuing.
```

```
Breakpoint 2, leon_init_timers (counter_fn=0xf00180c8 <timer_interrupt>)
  at /home/user/linuxbuild-1.0.2/linux/linux-2.6-git/arch/sparc/kernel/leon_ke
rnel.c:116
116 leondebug_irq_disable = 0;
Current language: auto; currently c
(gdb) bt
#0 leon_init_timers (counter_fn=0xf00180c8 <timer_interrupt>)
  at /home/user/linuxbuild-1.0.2/linux/linux-2.6-git/arch/sparc/kernel/leon_ke
rnel.c:116
#1 0xf03ce944 in time_init () at /home/user/linuxbuild-1.0.2/linux/linux-2.6-gi
t/arch/sparc/kernel/time_32.c:227
#2 0xf03cc13c in start_kernel () at /home/user/linuxbuild-1.0.2/linux/linux-2.6
-git/init/main.c:619
#3 0xf03cb804 in sun4c_continue_boot ()
#4 0xf03cb804 in sun4c_continue_boot ()
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
(gdb) info locals
eirq = <value optimized out>
rootnp = <value optimized out>
np = <value optimized out>
pp = <value optimized out>
len = 13
ampopts = <value optimized out>
(gdb) print len
$2 = 13
```

If the application for some reason need to be reloaded, then the MMU must first be disabled via the GRMON command **gdb reset**. It is similar to the regular **reset** command, but it will retain some of the state which GDB expects to be intact.

In addition all software breakpoints should be deleted before the application is restarted since the MMU has been disabled and GRMON won't translate virtual addresses anymore.

```
(gdb) mon mmu mctrl 0
mctrl: 006E0000 ctx: 00000000 ctxptr: 40440800 fsr: 00000000 far: 00000000
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) monitor gdb reset

(gdb) load
Loading section .text, size 0x10b0 lma 0x40000000
Loading section .data, size 0x50 lma 0x400010b0
Loading section .vmlinux, size 0x3f1a60 lma 0x40004000
Loading section .startup_prom, size 0x7ee0 lma 0x403f5a60
Start address 0x40000000, load size 4172352
Transfer rate: 18 KB/sec, 765 bytes/write.
(gdb) monitor gdb postload
(gdb) hbreak *0xf0004000
Hardware assisted breakpoint 3 at 0xf0004000: file /home/user/linuxbuild-1.0.2/linu
x/linux-2.6-git/arch/sparc/kernel/head_32.S, line 87.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/linuxbuild-1.0.2/output/images/image.ram

Breakpoint 3, trapbase_cpu0 () at /home/user/linuxbuild-1.0.2/linux/linux-2.6-gi
t/arch/sparc/kernel/head_32.S:87
87 t_zero: b gokernel; nop; nop; nop;
Current language: auto; currently asm
(gdb) break leon_init_timers
Breakpoint 4 at 0xf03cff14: file /home/user/linuxbuild-1.0.2/linux/linux-2.6-gi
t/arch/sparc/kernel/leon_kernel.c, line 116.
(gdb) cont
Continuing.
```



```
Breakpoint 4, leon_init_timers (counter_fn=0xf00180c8 <timer_interrupt>)
  at /home/user/linuxbuild-1.0.2/linux/linux-2.6-git/arch/sparc/kernel/leon_ke
rnel.c:116
116 leondebug_irq_disable = 0;
Current language: auto; currently c
```

3.7.8. Specific GDB optimization

GRMON detects GDB access to register window frames in memory which are not yet flushed and only reside in the processor register file. When such a memory location is read, GRMON will read the correct value from the register file instead of the memory. This allows GDB to form a function trace-back without any (intrusive) modification of memory. This feature is disabled during debugging of code where traps are disabled, since no valid stack frame exist at that point.

3.7.9. GRMON GUI considerations

The Graphical User Interface of GRMON can be used in parallel with GDB C/C++ level debugging. More details are described in Section 4.6.1, “GDB interface”.

3.7.10. Limitations of GDB interface

GDB must be built for the target architecture, a native PC GDB does not work together with GRMON. The toolchains that we distributes comes with a patched and tested version of GDB.

Do not use the GDB **where** commands in parts of an application where traps are disabled (e.g. trap or exception handlers). Since the stack pointer is not valid at this point, GDB might go into an infinite loop trying to unwind false stack frames. The thread support might not work either in some trap handler cases.

The step instruction commands **si** or **stepi** may be implemented by GDB inserting software breakpoints through GRMON. This is an approach that is not possible when debugging in read-only memory such as boot sequences executed in PROM/FLASH. One can instead use hardware breakpoints using the GDB command **hbreak** manually.

3.8. Thread support

GRMON has thread support for the operating systems shown below. The thread information is accessed using the GRMON **thread** command. The GDB interface of GRMON is also thread aware and the related GDB commands are described in the GDB documentation and in Section 3.7.6, “GDB Thread support”.

Supported operating systems

- RTEMS
- VXWORKS
- PikeOS
- Bare-metal

GRMON needs the symbolic information of the image that is being debugged in order to retrieve the addresses of the thread information. Therefore the symbols of the OS must be loaded automatically by the ELF-loader using **load** or manually by using the **symbols** command. GRMON will traverse the thread structures located in the target's memory when the **thread** command is issued or when GDB requests thread information. Bare-metal threads are used by default if no OS threads can be found. In addition the startup switch `-bmtthreads` can be used to force bare-metal threads.

The target's thread structures are never changed, and they are only accessed when the **thread** command is executed. Starting GRMON with the `-nothreads` switch disables the thread support in GRMON and the GDB server.

During debugging sessions it can help the developer a lot to view all threads, their stack traces and their states to understand what is happening in the system.

3.8.1. GRMON thread options

The following command-line options are available for selecting how GRMON4 will handle threads.

```
-nothreads
  Disable thread support.
-bmthreads
  Force bare-metal thread support
-rtems version
  Set RTEMS version for thread support, where the required argument version is one of the following:
  rcc-1.3.0
  rcc-1.3-rc9
  rcc-1.3-rc8
  rcc-1.3-rc7
  rcc-1.3-rc6
  rtems-6.0
  rtems-5.1
  rtems-5.0
  rtems-4.12
  rtems-4.11
  rtems-4.10
  rtems-4.8
  rtems-4.6
  edisoft-4.8
  The edisoft-4.8 option enabled RTEMS-4.8 edisoft configuration with hard FPU enabled RTEMS kernel
  (no AMP support). NOTE: TIME field of thread command is not implemented.
```

3.8.2. GRMON thread commands

thread info lists all threads currently available in the operating system. The currently running thread is marked with an asterisk.

```
grmon> thread info
```

Name	Type	Id	Prio	Ticks	Entry point	PC	State
Int.	internal	0x09010001	255	138	_CPU_Thread_Idle_body	0x4002f760	READY
UI1	classic	0x0a010001	120	290	Init	0x4002f760	READY
ntwk	classic	0x0a010002	100	11	rtems_bsdnet_schedneti	0x4002f760	READY
DCrx	classic	0x0a010003	100	2	rtems_bsdnet_schedneti	0x4002f760	Wevnt
Dctx	classic	0x0a010004	100	4	rtems_bsdnet_schedneti	0x4002f760	Wevnt
FTPa	classic	0x0a010005	10	1	split_command	0x4002f760	Wevnt
FTPD	classic	0x0a010006	10	1	split_command	0x4002f760	Wevnt
* HTPD	classic	0x0a010007	40	79	rtems_initialize_webse	0x40001b60	READY

thread bt ?id? lists the stack backtrace. **bt** lists the backtrace of the currently executing thread as usual.

```
grmon> thread bt 0x0a010003
```

```
%pc
#0 0x4002f760 _Thread_Dispatch + 0x11c
#1 0x40013ed8 rtems_event_receive + 0x88
#2 0x40027824 rtems_bsdnet_event_receive + 0x18
#3 0x4000b664 websFooter + 0x484
#4 0x40027708 rtems_bsdnet_schednetisr + 0x158
```

A backtrace of the current thread (equivalent to the **bt** command):

```
grmon> thread bt 0x0a010007
```

```
%pc          %sp
#0 0x40001b60 0x43fea130 console_outbyte_polled + 0x34
#1 0x400017fc 0x43fea130 console_write_support + 0x18
#2 0x4002dde8 0x43fea198 rtems_termios_puts + 0x128
#3 0x4002df60 0x43fea200 rtems_termios_puts + 0x2a0
#4 0x4002dfe8 0x43fea270 rtems_termios_write + 0x70
#5 0x400180a4 0x43fea2d8 rtems_io_write + 0x48
```

```
#6 0x4004eb98 0x43fea340 device_write + 0x2c
#7 0x40036ee4 0x43fea3c0 write + 0x90
#8 0x4001118c 0x43fea428 trace + 0x38
#9 0x4000518c 0x43fea498 websOpenListen + 0x108
#10 0x40004fb4 0x43fea500 websOpenServer + 0xc0
#11 0x40004b0c 0x43fea578 rtems_initialize_webserver + 0x204
#12 0x40004978 0x43fea770 rtems_initialize_webserver + 0x70
#13 0x40053380 0x43fea7d8 _Thread_Handler + 0x10c
#14 0x40053268 0x43fea840 __res_mkquery + 0x2c8
```

3.9. Forwarding application console I/O

If GRMON is started with `-u [N]` (N defaults to zero - the first UART), the target system APBUART[N] is placed in FIFO debug mode or in loop-back mode. Debug mode was added in GRLIB 1.0.17-b2710 and is reported by **info sys** in GRMON as *"DSU mode (FIFO debug)"*, older hardware is still supported using loop-back mode. In both modes flow-control is enabled. Both in loop-back mode and in FIFO debug mode the UART is polled regularly by GRMON during execution of an application and all console output is printed on the GRMON console. When `-u` is used there is no point in connecting a separate terminal to UART1.

In addition it is possible to enable or disable UART forwarding using the command **forward**. Optionally it is also possible to forward the I/O to a custom TCL channel using this command.

With FIFO debug mode it is also possible to enter text in GRMON which is inserted into the UART receive FIFO. These insertions will trigger interrupts if receiver FIFO interrupts are enabled. This makes it possible to use GRMON as a terminal when running an interrupt-driven O/S such as Linux or VxWorks.

The following restrictions must be met by the application to support either loop-back mode or FIFO debug mode:

1. The UART control register must not be modified such that neither loop-back nor FIFO debug mode is disabled
2. In loop-back mode the UART data register must not be read

This means that `-u` cannot be used with PROM images created by MKPROM. Also loop-back mode can not be used in kernels using interrupt driven UART consoles (e.g. Linux, VxWorks).

The **forward start**, or the commandline option `-uc li [N]`, can be used to make the current shell start forwarding I/O. This can be used when running applications from GDB to redirect I/O to the GRMON terminal instead of the GDB terminal.

RXVT must be disabled for debug mode to work in a MSYS console on Windows. This can be done by deleting or renaming the file `rxvt.exe` inside the bin directory, e.g., `C:\msys\1.0\bin`. Starting with MSYS-1.0.11 this will be the default.

3.9.1. UART debug mode

When the application is running with UART debug mode enabled the following key sequences will be available. The sequences can be used to adjust the input to what the target system expects. For a key sequence to take effect, both key presses must be pressed within 1.5 seconds of each other. Otherwise, they will be forwarded as is.

Table 3.3. Uart control sequences

Key sequence	Action
Ctrl+A B	Toggle delete to backspace conversion
Ctrl+A C	Send break (Ctrl+C) to the running application
Ctrl+A D	Toggle backspace to delete conversion
Ctrl+A E	Toggle local echo on/off
Ctrl+A H	Show a help message
Ctrl+A N	Enable/disable newline insertion on carriage return
Ctrl+A S	Show current settings
Ctrl+A Z	Send suspend (Ctrl+Z) to the running application
Ctrl+A Ctrl+A	Send a single Ctrl+A to the running application

3.10. EDAC protection

3.10.1. Using EDAC protected memory

Some LEON Fault-Tolerant (FT) systems use EDAC protected memory. To enable the memory EDAC during execution, GRMON should be started with the `-edac` switch. Before any application is loaded, the **wash** command might be issued to write all RAM memory locations and thereby initialize the EDAC check-sums. If a LEON CPU is present in the system GRMON will instruct the CPU to clear memory, clearing memory on a CPU-less system over a slow debug-link can be very time consuming.

```
$ grmon -edac
...
grmon3> wash
40000000          8.0MB /   8.0MB  [=====>] 100%
60000000          256.0MB / 256.0MB [=====>] 100%
Finished washing!
```

By default **wash** writes to all EDAC protected writable memory (SRAM, SDRAM, DDR, etc.) areas which has been detected or forced with a command line switch. *start* and *stop* parameters can also be given to wash a range. Washing memory with EDAC disabled will not generate check bits, however it can be used to clear or set a memory region even if the memory controller does not implement EDAC.

```
grmon3> wash 0x40000000 0x41000000
40000000          16.0MB /  16.0MB  [=====>] 100%
Finished washing!
```

If the memory controller has support for EDAC with 8-bit wide SRAM memory, the upper part of the memory will consist of check bits. In this case the wash will only write to the data area (the check bits will automatically be written by the memory controller). The amount of memory written will be displayed in GRMON. See Section 3.11, “PROM programming” for more information.

GRMON has support to write checkbits automatically to some PROM technologies when EDAC is enabled. If not supported the check bits for 32-bit memory must be written by the user with a diagnostic access in the memory controller registers. For 8-bit PROMs, the check bits can be generated by the `mkprom2` utility and included in the image.

3.10.2. LEON3-FT error injection

All RAM blocks (cache and register-file memory) in LEON3-FT are Single Event Upset (SEU) protected. Error injection function emulates SEU in LEON3-FT memory blocks and lets the user test the fault-tolerant operation of LEON3-FT by inserting random bit errors in LEON3-FT memory blocks during program execution. An injected error flips a randomly chosen memory bit in one of the memory blocks, effectively emulating a SEU. The user defines error rate and can choose between two error distribution modes:

1. **Uniform error distribution mode.** The `'ei un NR T'` command instructs GRMON to insert *NR* errors during the time period of *T* minutes. After *T* minutes has expired no more errors are inserted, but the application will continue its execution.
2. **Average error rate mode.** With the `'ei av R'` command the user selects at which rate errors are injected. Average error rate is *R* errors per second. Randomly generated noise is added to every error injection sample. The time between two samples vary between zero up to two periods depending on the noise, where one period is *1/R* seconds. Errors are inserted during the whole program execution.

GRMON can also perform error correction monitoring and report error injection statistics including number of detected and injected errors and error coverage, see `ei` command reference.

Error injection is performed during the run-loop of GRMON, to improve the performance and accuracy other services in the run-loop should be disabled. For example profiling and UART tunneling should be disabled, and one should select the fastest debug-link.

```
grmon> load rtems-tasks
40000000, .text          113.9kB / 113.9kB  [=====>] 100%
4001c7a0, .data          2.7kB /   2.7kB  [=====>] 100%
Total size: 116.56kB (786.00kbit/s)
Entry point 0x40000000
Image /home/daniel/examples/v8/stanford.exe loaded

grmon> ei un 100 1
Error injection enabled
100 errors will be injected during 1.0 min

grmon> ei stat en
```

Error injection statistics enabled

```
grmon> run
...
grmon> ei stat
itag :    5/   5 (100.0%)   idata:    5/  18 ( 27.8%)
dtag :    1/   1 (100.0%)   ddata:    4/  22 ( 18.2%)
IU RF :    4/  10 ( 25.0%)
FPU RF:    0/   4 (  0.0%)
Total :   19/  60 ( 31.7%)
grmon>
```

The real time elapsed is always greater than LEON CPU experienced since the LEON is stopped during error injection. Times and rates given to GRMON are relative the experienced time of the LEON. The time the LEON is stopped is taken into account by GRMON, however minor differences is to be expected.

3.11. PROM programming

3.11.1. EEPROM

GRMON supports programming of EEPROM attached to the external memory bus, through the **eeload** command.

Programming checkbits for 32-bit memories will be handled automatically by the memory controller. Programming checkbits automatically for 8-bit memories is not supported. Checkbits for 8-bit memories can be programmed by creating a section the ELF-file with the checkbits in it, or by loading a binary image with the checkbits in it separately.

3.11.2. MRAM

GRMON supports programming of MRAM attached to the external memory bus, through the **load** command with the option `-wprot`.

Programming checkbits for 32-bit memories will be handled automatically by the memory controller. Programming checkbits automatically for 8-bit memories is not supported. Checkbits for 8-bit memories can be programmed by creating a section the ELF-file with the checkbits in it, or by loading a binary image with the checkbits in it separately.

3.11.3. CFI compatible Flash PROM

GRMON supports programming of CFI compatible flash PROMs attached to the external memory bus, through the **flash** command. Flash programming is only supported if the target system contains one of the following memory controllers MCTRL, FTMCTRL, FTSRCTRL or SSRCTRL. The PROM bus width can be 8-, 16- or 32-bit. It is imperative that the PROM width in the MCFG1 register correctly reflects the width of the external PROM.

To program 8-bit and 16-bit PROMs, GRMON must be able to do byte (or half-word) accesses to the target system. To support this either connect with a JTAG debug link or have at least one working SRAM/SDRAM bank and a Leon CPU available in the target system.

Programming the EDAC checkbits for 8- or 32-bit PROMs is also supported. GRMON will automatically program the checkbits if EDAC is enabled. EDAC can be enabled by the `-edac` commandline option, using the `mcfg3` command or setting the register bit via the TCL variable `mctrl# : mcfg3 : pe`. When programming 32-bit EDAC checkbits it is required that no other AHB master is accessing the memory. Other masters can for example be DMA or SpaceWire RMAP accesses. When programming 8-bit EDAC checkbits, GRMON will ignore any data that should have been written to the EDAC area of the memory.

There are many different suppliers of CFI devices, and some implements their own command set. The command set is specified by the CFI query register 14 (MSB) and 13 (LSB). The value for these register can in most cases be found in the datasheet of the CFI device. GRMON supports the command sets that are listed in Table 3.4, “Supported CFI command set”.

Table 3.4. Supported CFI command set

Q13	Q14	Description
0x01	0x00	Intel/Sharp Extended Command Set

Q13	Q14	Description
0x02	0x00	AMD/Fujitsu Standard Command Set
0x03	0x00	Intel Standard Command Set
0x00	0x02	Intel Performance Code Command

Some flash chips provides lock protection to prevent the flash from being accidentally written. The user is required to actively lock and unlock the flash. Note that the memory controller can disable all write cycles to the flash also, however GRMON automatically enables PROM write access before the flash is accessed.

The flash device configuration is auto-detected, the information is printed out like in the example below. One can verify the configuration so that the auto-detection is correct if problems are experienced. The block lock status (if implement by the flash chip) can be viewed like in the following example:

```
grmon3> flash
Manuf.      : Intel
Device      : MT28F640J3
Device ID   : 09169e01734a9981
User ID     : ffffffff

1 x 8 Mbytes = 8 Mbytes total @ 0x00000000

CFI information
Flash family : 1
Flash size   : 64 Mbit
Erase regions : 1
Erase blocks : 64
Write buffer : 32 bytes
Lock-down    : Not supported
Region 0     : 64 blocks of 128 kbytes

grmon3> flash status
Block lock status: U = Unlocked; L = Locked; D = Locked-down
Block 0 @ 0x00000000 : L
Block 1 @ 0x00020000 : L
Block 2 @ 0x00040000 : L
Block 3 @ 0x00060000 : L
...
Block 60 @ 0x00780000 : L
Block 61 @ 0x007a0000 : L
Block 62 @ 0x007c0000 : L
Block 63 @ 0x007e0000 : L
```

A typical command sequence to erase and re-program a flash memory could be:

```
grmon3> flash unlock all
Unlock complete

grmon3> flash erase all
Erase in progress
Block @ 0x007e0000 : code = 0x80 OK
Erase complete

grmon3> flash load rom_image.prom
...
grmon3> flash lock all
Lock complete
```

3.11.4. SPI memory device

GRMON supports programming of SPI memory devices that are attached to a SPICTRL or SPIMCTRL core. The flash programming commands are available through the cores' debug drivers. A SPI flash connected to the SPICTRL controller is programmed using '**spi flash**', for SPIMCTRL connected devices the '**spim flash**' command is used instead. See the command reference for respective command for the complete syntax, below are some typical use cases exemplified.

When interacting with a memory device via SPICTRL the driver assumes that the clock scaler settings have been initialized to attain a frequency that is suitable for the memory device. When interacting with a memory device via SPIMCTRL all commands are issued with the normal scaler setting unless the alternate scaler has been enabled.

A command sequence to save the original first 32 bytes of data before erasing and programming the SPI memory device connected via SPICTRL could be:

```
spi set div16
```

```
spi flash select 1
spi flash dump 0 32 32bytes.srec
spi flash erase
spi flash load romfs.elf
```

The first command initializes the SPICTRL clock scaler. The second command selects a SPI memory device configuration and the third command dumps the first 32 bytes of the memory device to the file `32bytes.srec`. The fourth command erases all blocks of the SPI flash. The last command loads the ELF-file `romfs.elf` into the device, the addresses are determined by the ELF-file section address.

Below is a command sequence to dump the data of a SPI memory device connected via SPIMCTRL. The first command tries to auto-detect the type of memory device. If auto-detection is successful GRMON will report the device selected. The second command dumps the first 128 bytes of the memory device to the file `128bytes.srec`.

```
spim flash detect
spim flash dump 0 128 128bytes.srec
```

3.12. Automated operation

GRMON can be used to perform automated non-interactive tasks. Some examples are:

- Test suite execution and checking
- Stand-alone memory test with scripted access patterns
- Generate SpaceWire or Ethernet traffic
- Peripheral register access during hardware bring-up without involving a CPU
- Evaluate how a large set of compiler option permutations affect application performance

3.12.1. Tcl commanding during CPU execution

In many situations it is necessary to execute GRMON Tcl commands at the same time as the processor is executing. For example to monitor a specific register or a memory region of interest. Another use case is to change system state independent of the processor, such as error injection.

When the target executes, the GRMON terminal is assigned to the target system console and is thus not available for GRMON shell input. Furthermore, commands such as **run** and **cont** return to the user first when execution has completed, which could be never for a non-behaving program.

Three different methods for executing Tcl commands during target execution are described below:

- *Spawn one or more user Tcl shells.* The user shells run in their own thread independent of the shell controlling CPU execution. This is done with the **usrsh** command.
- *Detach GRMON from the target.* This means that the application continues running with GRMON no longer having control over the execution. This is done with the **detach** and **attach** commands.

3.12.2. Communication channel between target and monitor

A communication channel between GRMON and the target can be created by sharing memory. Use cases include when a target produces log or trace data in memory at run-time which is continuously consumed by GRMON reading out the the data over the debug link. For this to work safely without the need to stop execution, some arbitration over the data has to be implemented, such as a wait-free software FIFO.

As an example, the target processors could produce log entries into dedicated memory buffers which are monitored by an exec hook. When new data is available for the consumer, the exec hook schedules an asynchronous bus read with **amem** to fetch all new data. When the asynchronous bus read has finished, the exec hook acknowledges that the data has been consumed so that the buffer can be reused for more produce data. One benefit of using **amem** is that multiple buffers can be defined and fetched simultaneously independent of each other.

3.12.3. Test suite driver

GRMON can be used with a driver script for automatic execution of a test suite consisting of self-checking target applications. For this purpose a script is created which contains multiple **load** and **run** commands followed by system state checking at end of each target execution. State checking could be implemented by checking an application return value in a CPU register using the **reg** command. In case an anomaly is detected by the driver script, the system state is dumped with commands such as **reg**, **bt**, **inst** and **ahb** for later inspection. All command output

is written to a log file specified with the GRMON command line option `-log`. It is also useful to implement a time-out mechanism in an exec hook to mitigate against non-terminating applications.

The example below shows a simple test suite driver which uses some of the techniques described in this section to test the applications named `test000.elf`, `test001.elf` and `test002.elf`. It can be run by issuing

```
$ grmon <debuglink> -u -c testsuite.tcl -log testsuite.log
$ grep FAIL testsuite.log
```

in the host OS shell. Target state will be dumped in the log file `testsuite.log` for each test case which returns nonzero or crashes.

Example 3.3. Test suite driver example

```
# This is testsuite.tcl
set nfail 0

proc dumpstate {} {
    bt; thread info; reg; inst 256; ahb 256; info reg
}

proc testprog {tname} {
    global nfail
    puts "### TEST $tname BEGIN"
    load $tname
    set tstart [clock seconds]
    set results [run]
    set tend [clock seconds]
    puts [format "### Test executed %d seconds" [expr $tend - $tstart]]
    set exec_ok 0
    foreach result $results {
        if {$result == "SIGTERM"} {
            set exec_ok 1
        }
    }
    if {$exec_ok == 1} {
        puts "### PASS: $tname"
    } else {
        incr nfail 1
        puts "### FAIL: $tname ($results)"
        dumpstate
    }
    puts "### TEST $tname END"
}

proc printsummary {} {
    global nfail
    if {0 == $nfail} {
        puts "### SUMMARY: ALL TESTS PASSED"
    } else {
        puts "### SUMMARY: $nfail TEST(S) FAILED"
    }
}

after 2000
testprog test000.elf
testprog test001.elf
testprog test002.elf
printsummary
exit
```


4. Graphical user interface

This chapter describes how to operate the Graphical User Interface (GUI) introduced with GRMON3.

4.1. Overview

The GUI provides the user with a fully interactive environment with the possibility to monitor and control different parts of the system in parallel. All functionality of the GRMON Tcl command line interface are accessible from the terminal emulator view.

GRMON visualizes hardware state by views includes the following functions:

- Debug-link and system configuration dialog
- Multi-core LEON/NOEL-V and OS threads execution status and backtrace view
- Disassembly view with symbol and breakpoint information
- Memory, CPU register and I/O register inspection and edit views
- Optimized SPARC/LEON and RISC-V/NOEL-V IU register view
- Basic execution control such as single-stepping, continuing, breaking
- C/C++ source level debugging including source code view, stepping and line breakpoints
- Variables and expressions views.
- Application launch dialog
- Tcl terminal views with history, tab-completion, etc.
- Application terminals via UART forwarding
- GRLIB SOC system hardware overview
- Breakpoints view showing breakpoint and watchpoint information

Users which are already familiar with the GRMON CLI can use the GUI as a drop-in replacement with the added interactive functionality.

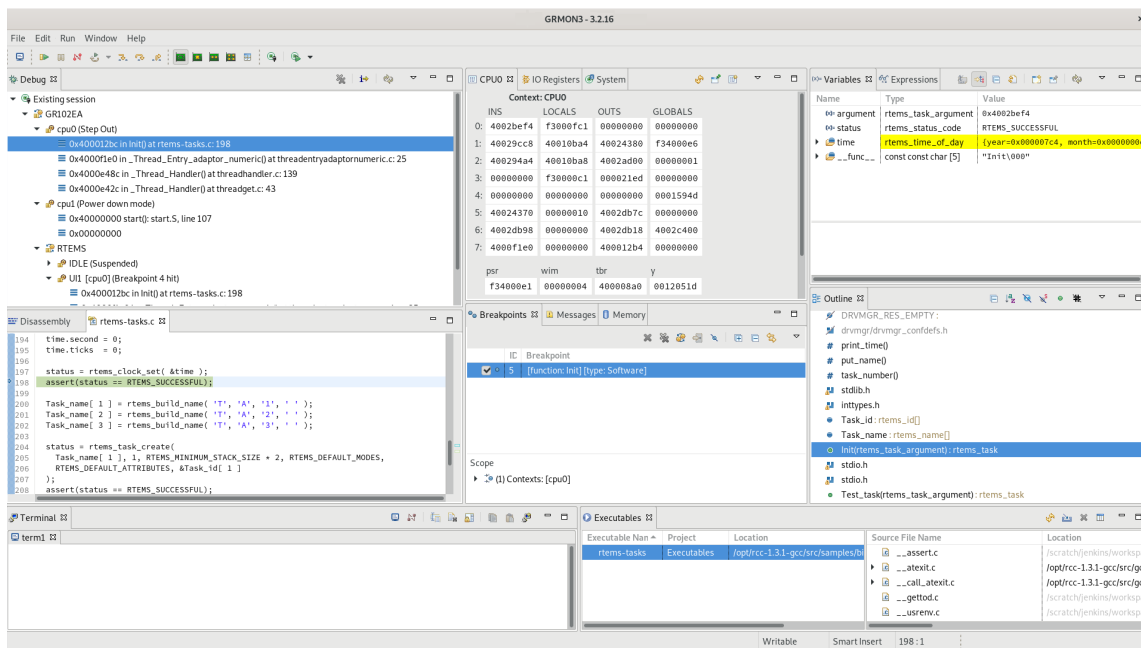


Figure 4.1. The GRMON graphical user interface

4.2. Starting GRMON GUI

GRMON is started by executing the same `grmon` or `grmon.exe` binary as the CLI version. The switches described in Section 3.2.3, “General options” determine the start-up operation.

If GRMON is started without any debug link command line options, then GRMON will start the GUI and open a dialog window which allows for selecting debug link. This is described in Section 4.3, “Connect to target”.

```
$ grmon
```

Specifying a debug link along with the `-gui` option will skip the dialog for selecting debug link. In this case the GUI will start and the connection will be done according to the full command line. Below is an example of starting the GUI and connecting to a system using the FTDI USB serial interface:

```
$ grmon -gui -ftdi
```

It is also possible to start the GUI from the command line interface by issuing the `gui` command. This is useful if GRMON is first operated as a command line tool but the user selects to continue debugging using the graphical representation of the system. The on-going debug session and hardware state will not be altered but can be operated and inspected from the GUI.

The GUI will connect to GRMON through an available TCP port chosen randomly, but this can be overridden by setting a port number with the `-guiport` option. Example:

```
$ grmon -gui -ftdi -guiport 42069
```

4.3. Connect to target

The *System Configurations* dialog is used to connect to the target system. It allows for selecting the debug link and parameters which system initialization. See Figure 4.2.

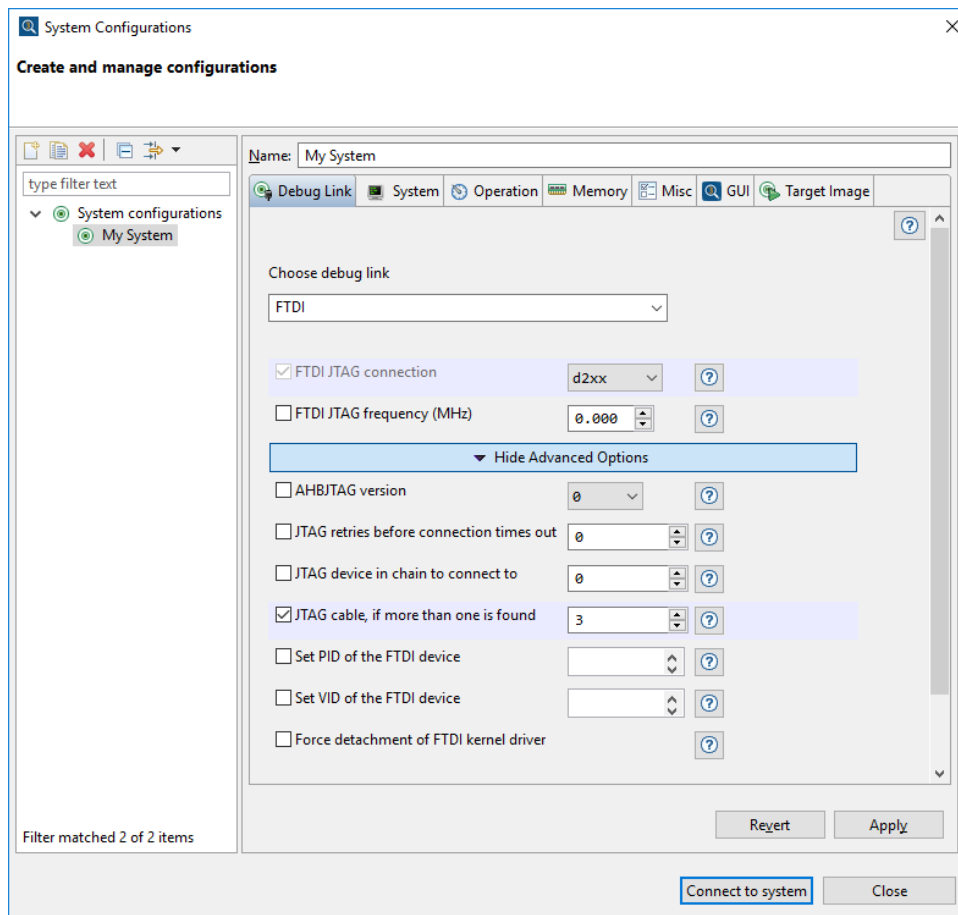


Figure 4.2. *System Configurations* dialog. The *Debug Link* tab is selected and *Advanced Parameters* are displayed.

The *System Configurations* is split into several tabs with group related settings. Here follows a brief summary of each tab. Please browse the tabs to discover relevant settings.

- Debug Link - Choose debug link and set parameters
- System - Options related to initialization performed by GRMON when connecting to the target

- Operation - Controls how GRMON interacts with the target after connect
- Memory - Options related to memory controllers
- Misc - Driver specific options which need to be determined at connect
- GUI - Options related to how the GUI views are synchronized with the current target state

4.3.1. Debug link

The debug link to use is selected in the *Choose debug link* drop down menu in the *Debug Link* tab, as illustrated in Figure 4.2. When a debug link has been selected, parameters specific for that debug link are displayed.

All debug links supported by GRMON4 are displayed in the drop down menu, including those which may not be available on the host and target system. For more information on the GRMON4 debug links and their individual options, see Chapter 5, *Debug link*.

4.3.2. Options

The options presented in *System Configurations* are equivalent to the command-line options available in GRMON4 CLI. Clicking on the "?" icon next to an option will open the option specific documentation. Target initialization and system related options can be activated in the different tabs of the connection dialog. This is done by clicking on the button to the left of the parameter name, or on the name itself. A selected parameter is marked with a button with an "X" and a different background color.

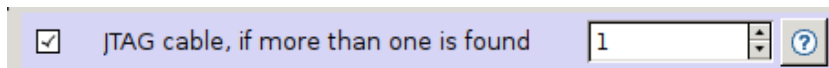


Figure 4.3. A selected parameter

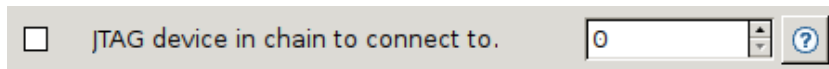


Figure 4.4. A non-selected parameter

The most common GRMON options are always displayed in the respective tab. More GRMON options can be displayed by clicking the *Show Advanced Parameters...* button.




4.3.3. Argument contribution

When a parameter or setting is selected, the corresponding command line argument is shown in the *Argument contribution* box. All selected parameters and settings from the current tab is shown in the box. Switching tabs will show different argument contributions.

Note that the debug link always contributes with an argument to the *Argument contribution* box. For example choosing the USB debug link will add the argument `-usb`.

4.3.4. Configurations

Multiple system configurations can be used and managed in the connection dialog. This is useful for example when the same host is used to connect to many different target systems. Another use case for the system configurations is as a convenient way for connecting to the same target, but with different initialization options.

To create a new System Configuration, either click the new configuration button , or clone the current configuration by clicking the copy button . Delete a configuration using the delete button . These options are also available from the context menu of the configuration in the listing on the left of the dialog.

When a configuration is modified, the two buttons *Revert* and *Apply* become enabled. Pressing the *Apply* button stores the configuration, and *Revert* will undo any changes since last storage.


4.3.5. Connect

When the System Configuration is done, press the *Connect to system* button.

4.4. Launch configurations

A Launch configuration is a combination of application images and custom system preparations for images. In its most basic form a launch configuration consists of a single application image selected by the user which is loaded and started.

4.4.1. Target image setup

Once connected to a system, an image can be uploaded to the system. To setup the image and related settings, click the Target Image Configuration button  in the main tool bar, or in the main menu under File > Launch Target Image...

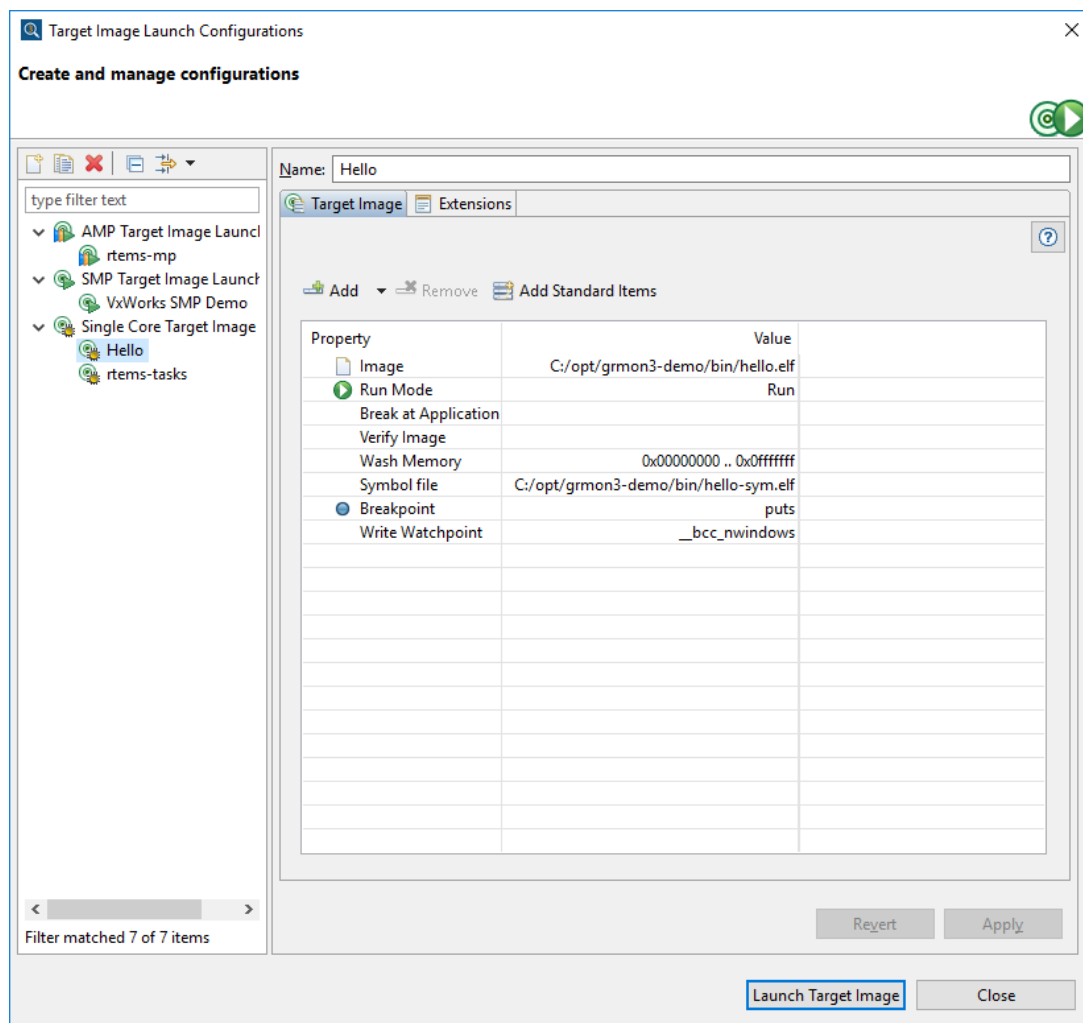


Figure 4.5. The Target Image Configuration dialog

A Target Image Launch Configuration consists of one or several application images which are associated to different CPUs. There are three different types of image launch configurations:

- Single Core Image Launch
- SMP Target Image Launch
- AMP Target Image Launch

An Single Core Target Image Launch also provides a simple interface to setup the application. This image launch type is appropriate for applications that will use a single core. In the Single Core Target Image Launch, properties are always assigned to the first cpu.

An *SMP Target Image Launch* provides an interface to setup the application for multiple cores. This image launch type is appropriate for applications based on an SMP operating system, such as RTEMS SMP or VxWorks SMP. In the *SMP Target Image Launch*, properties are always assigned to all CPUs.

In an *AMP Target Image Launch*, the user has full flexibility to assign launch properties for each CPU. This is useful when different operating system instances are executed on different processors or sets of processors.

CPUs are added or removed by the user to match the target system. Each setting has a value and some of them can be individually associated with one or more CPUs. For instance a breakpoint and symbol file might be associated with CPU1, but not CPU2.

Associate a setting with a CPU by clicking the cell that intersect the setting and the CPU and tick the check box. Untick the check-box to reverse the association. Some settings can occur more than one time for the same CPU (i.e. *Symbol file*), meaning that several symbol files can be supplied. If an incompatible or unsupported combination of settings is associated to a CPU, then the background of the cells for the settings are set to bright red.

To edit the value of a setting click the cell in the *Value* column. Different settings are edited in different ways. If the cell after clicking it shows a button with "...", this button opens a dialog to change the value. For example the *Image* setting is edited by choosing a file from a dialog.

If any of the settings are invalid, then the dialog shows an error message at the top, and the button *Launch Target Image* are disabled. Once the invalid settings are corrected, the button becomes enabled and the *Target Image* can be launched.

A *Target Image Configuration* can be launched automatically at connect to the system. This can be selected in the *System configuration* dialog option named *Launch Target Image Configuration when connected*.

4.4.2. Launch properties

A simple launch configuration typically consists of only the *Image* and *Run Mode* properties. Additional properties can be added as required. The following list describes all available properties.

Properties:

Image

File name of an image file to load. More than one image can be added and each image can be assigned to any number of CPUs.

Command: **load**

Verify Image

Verify each image after load to memory. The property is not CPU specific. Only one *Verify Image* property can be specified.

Command: **verify**

Symbol file

File name of a symbol file to load. The symbols will be loaded into GRMON but no content from the file will be loaded into the target memory. More than one image can be added and each image can be assigned to any number of CPUs.

Command: **symbols**

Run Mode

- *Run* - Reset GRMON drivers and start the execution from the beginning of the application.
- *Go* - Start the execution from the beginning of the application with the current system CPU state.

At most one *Run Mode* can be specified. The recommended mode is *Run* for most applications. *Go* is useful when the application itself is initializing the target, such as a boot loader. The property is not CPU specific.

Note that it is also possible to omit the *Run Mode* property. In that case, the application could be started with the **run** command in a shell.

Command: **run, go**

Break at Application

Inserts a breakpoint at application start. The breakpoint location is OS dependent, for example `main()` in a bare-metal application and `Init()` in an RTEMS application. At most one *Break at Application* property can be specified.

Command: **bp**

Break at Entry

Inserts a breakpoint at the entry point of the loaded image. At most one *Break at Entry* property can be specified.

Command: **bp**

Breakpoint

Inserts a software (soft) breakpoint. The *Value* field can be either an address, symbol name or filename:linenumber. More than one soft breakpoint can be added and each can be assigned to any number of CPUs.

Command: **bp soft**

HW Breakpoint

Inserts a hardware (hard) breakpoint. The *Value* field can be either an address, symbol name or filename:linenumber. More than one hard breakpoint can be added and each can be assigned to any number of CPUs. Note that the number of available hardware breakpoints is target specific.

Command: **bp hard**

R/W Watchpoint, Read Watchpoint, Write Watchpoint

Inserts a read/write, read or write hardware watchpoint. The *Value* field can be either an address, symbol name or filename:linenumber. More than one hard breakpoint can be added and each can be assigned to any number of CPUs.

Command: **bp watch**

Command: **bp watch -read**

Command: **bp watch -write**

Wash Memory

Clear all or part of memory before loading images. Multiple ranges can be defined by adding more *Wash Memory* properties. The property is not CPU specific.

Command: **wash**

Stack Pointer, Entry point

Override stack pointer or entry point setting. An address or symbol can be specified. At most one stack pointer per CPU can be assigned. At most one entry point per CPU can be assigned.

Command: **stack**


Command: **ep**

dtb

File name of a dtb file to load.

Command: `dtb`

4.5. Perspectives

There are four different *perspectives* and they have the same layout and contain the same views by default, see Figure 4.6, “Default perspective”. You can add, remove and reposition any views as you like and the changes will be saved to the active perspective automatically, so it can be reused the next session. The perspectives can be opened with the following buttons . They can also be opened from Window > Open Perspective. Each perspective can be reset to its default by clicking Window -> Reset Perspective.

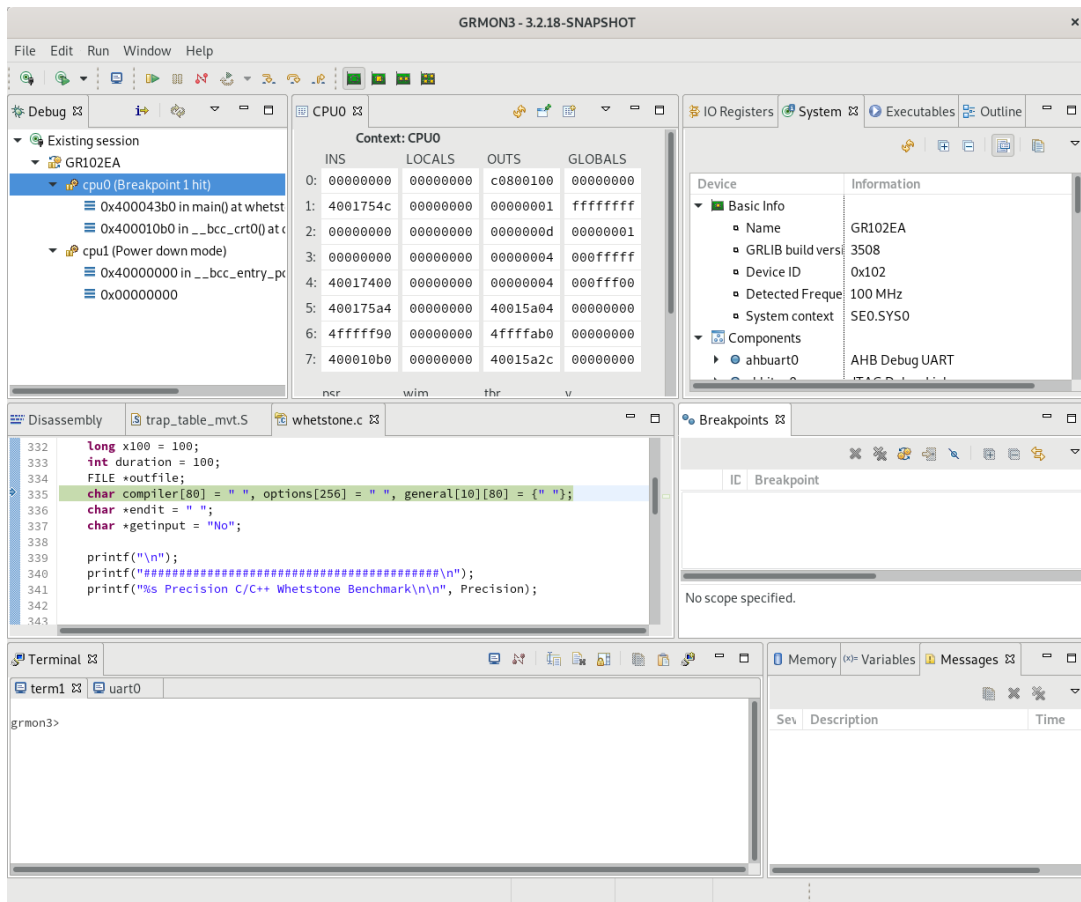


Figure 4.6. Default perspective

4.6. C/C++ source level debugging

The GUI includes the following C/C++ debugging functionality.

- *Source editor* for displaying source code
- Mixed mode in the *Disassembly* view for showing both source code and assembly instructions
- Source code stepping
- Breakpoints on source lines
- *Executables* view for inspecting files contained in an executable
- *Outline* view for checking all symbols in a source file
- *Variables* view to inspect variables of the current stackframe
- *Expressions* view to monitor user selected symbols

4.6.1. GDB interface

The GUI can also be used simultaneously with GDB as described in Section 3.7, “GDB interface”. When GDB has connected to GRMON, GDB is in control of the debugging. Similar to the command line interface it is not

possible to alter processor state or breakpoints without destroying GDB's internal representation. Doing so will leave GDB in an undefined state and cause various issues.

When GDB has stopped the execution the GUI can be used to view all hardware state. It is also possible to update hardware state not specifically controlled by GDB such as I/O registers.

The GUI will by default gain focus when GDB stops execution. To prevent this, click the drop down menu in the *Debug* view, choose *Preferences*. Click *Run/Debug* and uncheck *Activate the workbench when a breakpoint is hit* and *Active the debug view when a breakpoint is hit*.

4.7. Views

GRMON4 GUI provides different *views* for displaying and managing the target system state. *CPU*, *IO Registers*, *System*, *Router*, and *Messages* view are customized for GRLIB/LEON/NOEL-V systems. Other views are derived from the Eclipse framework.

4.7.1. Debug View

Debug view shows all CPUs and threads, called contexts, of the connected system. This view is central, as the selected context is used as input to many of the views: *Breakpoints*, *CPU*, *Disassembly*, *Expressions*, *Outline*, *Registers*, *Source*, *Variables*. Furthermore, if a stack frame is selected, it will be used as input to the mentioned views, otherwise the top frame is used.

Debug View can be opened from the main menu under *Window > Show View > Debug*.

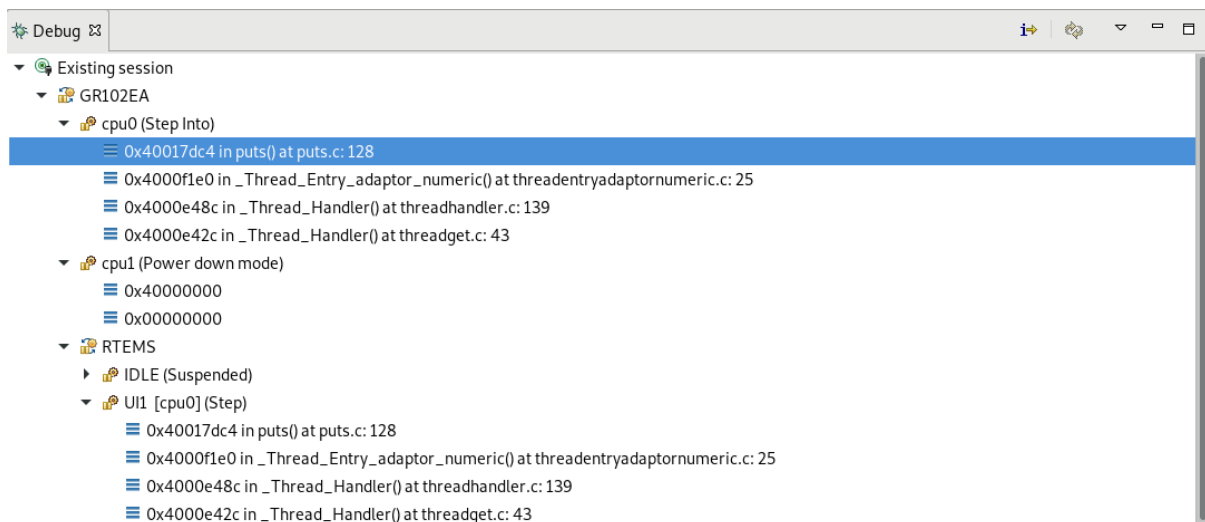


Figure 4.7. Here we see a system with two CPUs and two RTEMS threads. The top stack frame of CPU0 is selected and will therefore be used as input for the views mentioned above.

4.7.2. System Information View

This view shows a system overview with information about the IP-cores. The cores are listed with name, vendor, function, bus connection, address range when relevant and the driver info available.

The information shown is retrieved by GRMON from the system. If any changes are expected in the available IP-cores, or their related information, all the information can be retrieved anew from the system by pressing the *Refresh* button.

As default, the view shows only basic information for the IP-cores. Press the *Show Details* button in the toolbar to toggle between showing basic and all information.

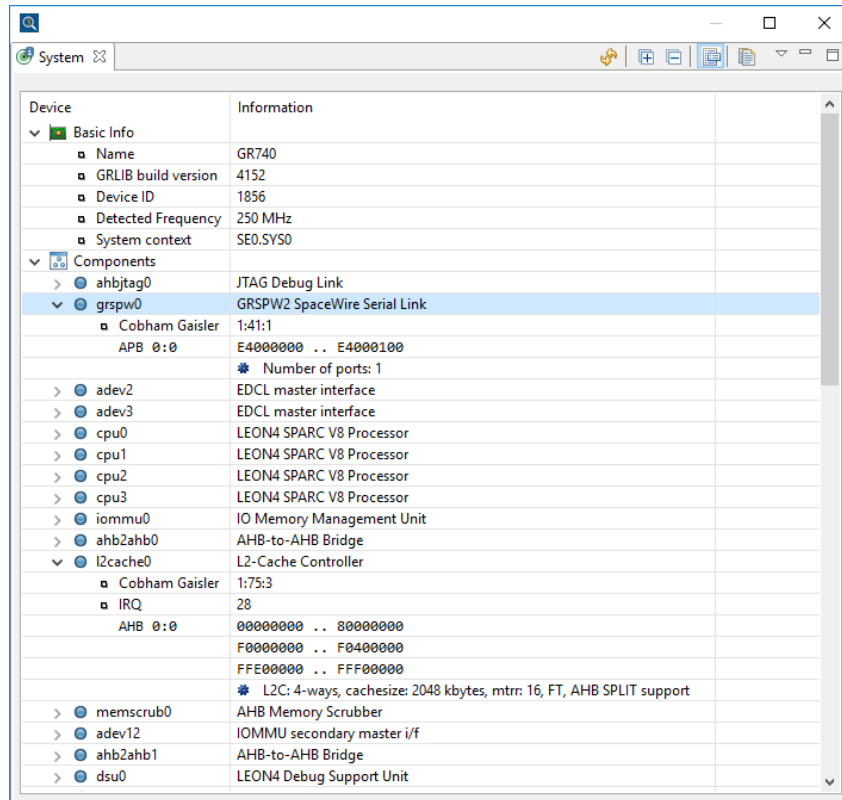


Figure 4.8. The system information view

4.7.3. IO Registers View

The IO Registers view allows users to inspect and modify I/O registers of AMBA devices available on the target hardware. Individual registers may be expanded into bit-fields which are presented as bit-masks or numbers. It functions much like the Eclipse Registers view, but is optimized for GRLIB SoC systems.

Registers are grouped under the AMBA device it belongs to. A device may have registers from both APB and AHB I/O space. The view presents the same information as the info CLI command.

The register view adapts to the register information available in GRMON. Users can add registers and bit-field declarations for custom IP-cores by means of Tcl drivers as described in Appendix C, *Tcl API*. Registers unknown to GRMON will not appear in the view.

Name	Hex	Decimal	Description	Address
> ddrsdmux0			ddrsdmux0 registers	
> mctrl0			mctrl0 registers	
> uart0			uart0 registers	
> uart1			uart1 registers	
> gpio0			gpio0 registers	
> irqmp0			irqmp0 registers	
▼ gptimer0			gptimer0 registers	
scalar	000000f9	249	Scalar value register	0xFF908000
reload	000000f9	249	Scalar reload value register	0xFF908004
> cfg	0000010d	269	Configuration register	0xFF908008
latch	00000000	0	Latch configuration register	0xFF90800C
tmr0::count	ffffffff	4294967295	Timer 0 Value register	0xFF908010
tmr0::reload	ffffffff	4294967295	Timer 0 Reload value register	0xFF908014
▼ tmr0::ctrl	00000043	67	Timer 0 Control register	0xFF908018
6 dh	1	1	Debug Halt	0xFF908018
5 ch	0	0	Chain	0xFF908018
4 ip	0	0	Interrupt Pending	0xFF908018
3 ie	0	0	Interrupt Enable	0xFF908018
2 ld	0	0	Load	0xFF908018
1 rs	1	1	Restart	0xFF908018
0 en	<input checked="" type="checkbox"/>	1	Enable	0xFF908018
tmr0::latch	00000000	0	Timer 0 Latch register	0xFF90801C
tmr1::count	3d8a970d	1032492813	Timer 1 Value register	0xFF908020
tmr1::reload	3d8a970d	1032492813	Timer 1 Reload value register	0xFF908024
> tmr1::ctrl	00000040	64	Timer 1 Control register	0xFF908028
tmr1::latch	00000000	0	Timer 1 Latch register	0xFF90802C
tmr2::count	32188df0	840470000	Timer 2 Value register	0xFF908030
tmr2::reload	32188df0	840470000	Timer 2 Reload value register	0xFF908034

Figure 4.9. The I/O registers view

Register values that are changed since last update are highlighted with yellow. When expanding the register nodes in the tree of the view, the left column is auto expanded to fit the content.

For bit registers consisting of only one bit, a flag, the value can be toggled by clicking the value, and then checking or un-checking the check-box.

By default, the most common registers are presented in the view. However all registers known to GRMON may be presented by pressing the icon on the right top corner. Note that doing so may change the state of the hardware since reading certain registers may affect the state of the hardware.

Values for registers are retrieved only when they are to be shown in the view. Unless another view or operation requires the values of the registers to be loaded, the values are not retrieved until they are visible in the view.

The register values are updated from the target hardware because of a few different reason.

- Edit - when the value of a register changes as result of an edit
- Suspend - when a process is suspended, the visible registers are updated
- Refresh - the values can be re-fetched from the target hardware by pressing the refresh button, or by using the periodic refresh.

Note that changed values will not be highlighted if the register values are updated via refresh, either by clicking the refresh button or by periodic refreshes.

Click the arrow down icon in the toolbar to access the view pull down menu. Under the Layout item are different options on how to customize the appearance of the view. For instance which columns to display can be specified under *Select Columns...*

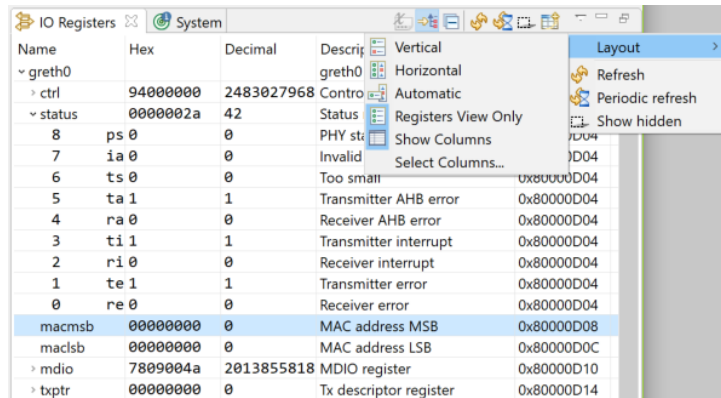


Figure 4.10. The I/O registers menu

4.7.4. CPU Registers View

The CPU Registers view shows a selection of the CPU registers in a fixed and compact format. Which CPU registers are shown for, depends on what CPU is relevant for the selected context in the Debug view.

Values are retrieved for the registers in two situations; either the register values is changed by a known mechanism such as editing the value, or when the CPU execution is suspended. Values that have changed since last retrieval are highlighted with a yellow background.

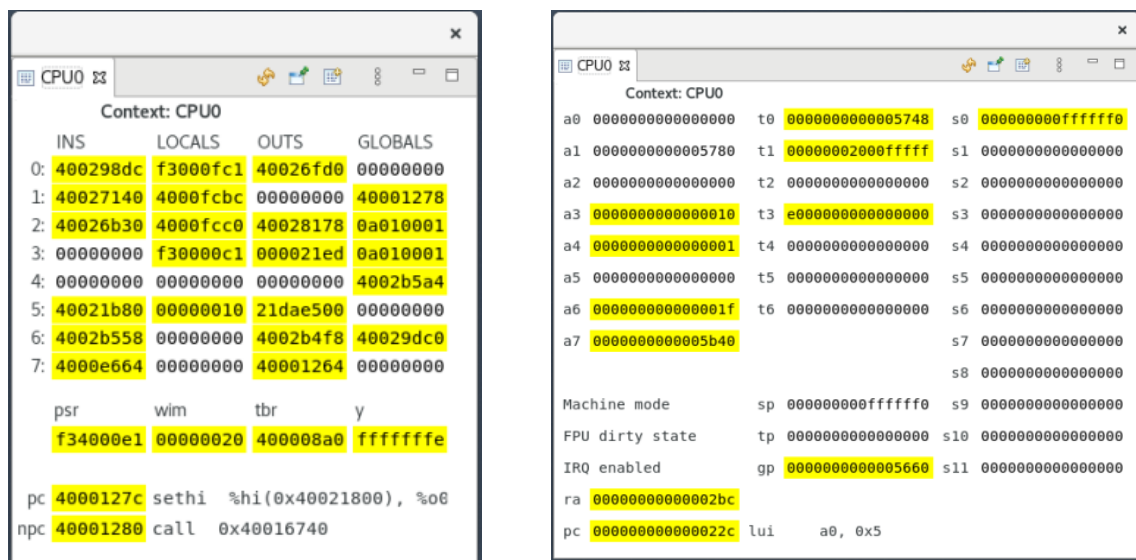


Figure 4.11. CPU registers view. The figure to the left shows registers for a LEON system and the figure to the right for an RV64 system.

4.7.4.1. Pinning

The CPU Registers view can be pinned to a context. When the view is pinned it continues to show registers for the same context even when the selection changes in the debug view. The view then shows the pinned CPU name in the top of the view, e.g. “cpu0”.

4.7.4.2. Context menu

Right click on any register value in the view to show the context menu (not relevant for the disassembly). If the text in the register value box is a valid hexadecimal value the two commands >>Open address in Memory view<< and >>Show disassembly at address<< will be available.

Open address in Memory view

Shows the Memory view and adds a memory monitor at the selected register value used as an address. The user must select the visualization, e.g. “Hex Integer”.

Show disassembly at address

If no disassembly editors are open one will be opened, otherwise any opened will be used. The editor will focus on the selected register value used as an address.

4.7.5. Registers View

The *Registers* view shows CPU registers for the selected context in the *Debug* view. The difference to the *CPU* view is that the *Registers* shows a larger set of the registers, such as floating point, mmu and cache registers, whereas the *CPU* view only show the most common registers.

Values are retrieved for the registers in two situations; either the register values is changed by a known mechanism such as editing the value, or when the CPU execution is suspended. Values that have changed since last retrieval are highlighted with a yellow background.

Name	Hex	Decimal	Description
o6	4002dab8	1073928888	Output Register 6
o7	40017dc8	1073839560	Output Register 7
l0	f3401fc0	4081065920	Local Register 0
l1	40011eb0	1073815216	Local Register 1
l2	40011eb4	1073815220	Local Register 2
l3	f34010c0	4081062080	Local Register 3
l4	00000f00	3840	Local Register 4
l5	00000008	8	Local Register 5
l6	00000000	0	Local Register 6
l7	00000000	0	Local Register 7
i0	40024380	1073890176	Input Register 0
i1	00000000	0	Input Register 1
i2	4002ad00	1073917184	Input Register 2
i3	000021ed	8685	Input Register 3
i4	00000000	0	Input Register 4
i5	21dae500	567993600	Input Register 5
i6	4002db18	1073928984	Input Register 6
i7	40001280	1073746560	Input Register 7
asr17	00000d47	3399	LEON3/4 configuration regi
mmu			MMU registers
cache			L1 Cache registers

Figure 4.12. Registers for a LEON based system. We see that o7 was recently changed.

The value of a register can be changed by clicking on its value in the *Hex* or *Decimal* tab. Press enter for the change to take effect.





4.7.5.1. Context menu

Right click on any register value in the view to show the context menu (not relevant for the disassembly). This allows the user to >>Watch in Expressions<<.

Watch in Expressions

Creates an Expression to watch from the selected register.

4.7.6. Router View

The Router view displays the general state of the Spacewire router of the connected board, if there is one. There are two tabs, *Status*, which displays the status of all the ports, including the configuration port, and *Error*, which shows any active errors. The values can be refreshed in different ways: manual refresh via *Refresh* , refresh whenever a CPU becomes suspended, *Refresh on CPU suspension* , or periodic refresh every x seconds, where x can be chosen between 0.5-30, *Periodic refresh* . Errors can be cleared with *Clear errors* .

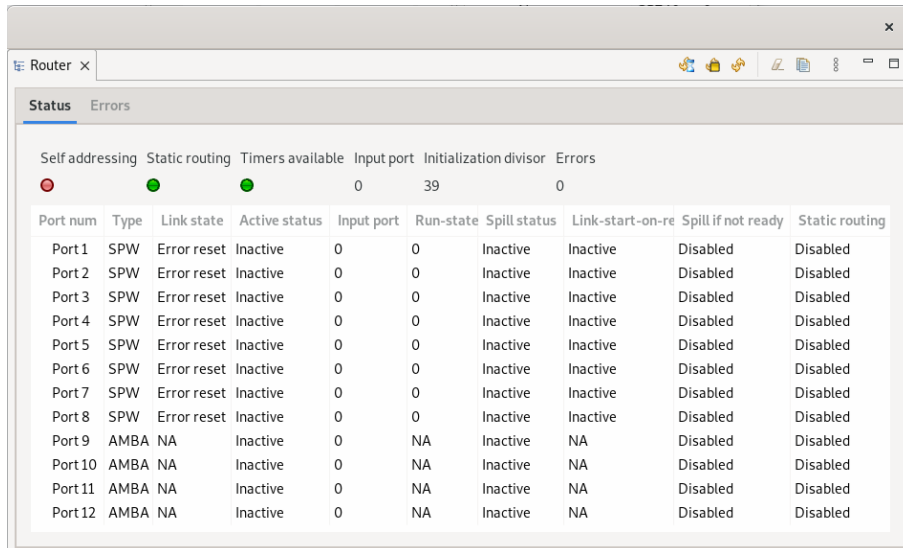


Figure 4.13. Router view

The status tab of the Router view, for a gr740 system.

4.7.6.1. Status tab

The state of the configuration port (port 0) is on the top of the view and shows:

- Self addressing Enabled or disabled.
- Static routing Enabled or disabled.
- Timers available Enabled or disabled.
- Input port The number of the last port from which a packet was routed to the configuration port.
- Initialization Clock divisor value used by all the SpaceWire links to generate the 10 Mbit/s rate during
clock divisor initialization.
- Errors How many errors are currently active for any of the ports, including the configuration port.
For possible errors, see Configuration port errors and Port errors.

The bottom shows the state of the remaining ports:

- Type SPW, AMBA or FIFO.
- Link state Error reset, Error wait, Ready, Started, Connecting or Run state.
- Active status Active or inactive.
- Input port This field shows the number of the input port for either the currently ongoing packet transfer
on this port (if RTR.PSTS.PB = 1), or for the last packet transfer on this port (if RTR.PSTS.PB
= 0).
- Run-state clock Clock divisor value used for the corresponding port's link interface when in run-state. Field
divisor is only available for the SpaceWire ports.
- Spill status Active or inactive.
- Link-start-on- Enabled or disabled. Only for SPW ports.
request
- Spill-if-not-ready Enabled or disabled
- Static routing Enabled or disabled

For the definition of these items, please consult glib or your device manual.

4.7.6.2. Error tab

Possible errors for the configuration port:

Configuration port errors

- *Early EOP*
- *Early EEP*
- *Packet type error*
- *Header CRC error*
- *Protocol ID error*
- *Error code*
- *SPW PnP error code*

Possible errors for the other ports:

Port errors

- *Timeout spill*
- *Memory error*
- *Invalid address error*
- *Credit error*
- *Escape error*
- *Disconnect error*
- *Parity error*

For the definition of these errors, please consult grlib or your device manual.

4.7.7. Source Editor

The *Source editor* shows the source file that the current stack frame in the *Debug* view points to. If the source file can't be found, there will be options to locate the file, edit the source lookup path or to show the *Disassembly view* instead. The default source lookup path can be changed from *Preferences*, which can be opened from the Window menu, under Source Lookup

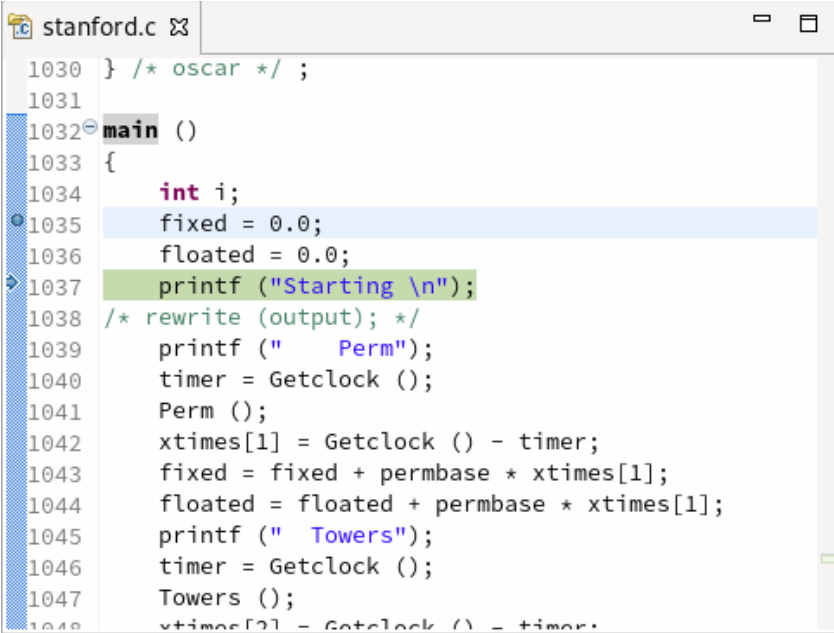
Source editor can not be opened by itself. It will open as soon as an executable is loaded and it can be opened by double-clicking a file in the *Executables view*.

Line breakpoints can be placed by double-clicking on the left margin or right-click and *Toggle Breakpoint*. The breakpoint will be placed on the active context in the *Debug* view. If the system context is selected, the breakpoint will be placed on all its CPUs except RTAs. *Add breakpoint* adds the option to place a hardware breakpoint and has an option to change the default CPU scope. Watchpoints can be added from the *Source editor* from the right-click menu and *Add watchpoint*. If a symbol is already selected, its name will be copied into the Watchpoint property page.

Symbols can be watched by right-clicking and choosing *Add Watch Expression*. If a symbol is already selected, then that symbol will be suggested to watch. The symbol will be opened in the *Expressions view*. Please read more about it in ref.

The *Source editor* has some limited editing functionality, but please be aware that you can't compile your source code with the GUI. Right-clicking and choosing *Preferences* opens up the *Preferences* menu, where you have some basic content assist options, save actions, theming and code formatting options.

Quick Outline can be used to see all symbols in the current file. It can be opened from the right-click menu and *Quick Outline*.



```
1030 } /* oscar */ ;
1031
1032 main ()
1033 {
1034     int i;
1035     fixed = 0.0;
1036     floated = 0.0;
1037     printf ("Starting \n");
1038     /* rewrite (output); */
1039     printf ("  Perm");
1040     timer = Getclock ();
1041     Perm ();
1042     xtimes[1] = Getclock () - timer;
1043     fixed = fixed + permbase * xtimes[1];
1044     floated = floated + permbase * xtimes[1];
1045     printf (" Towers");
1046     timer = Getclock ();
1047     Towers ();
1048     xtimes[2] = Getclock () - timer;
```

Figure 4.14. The Source editor showing `stanford.c`, with execution stopped at row 1037. A line breakpoint can be seen at row 1035.

Source stepping


There are several ways of source stepping through the code, and they will be described in this section.

Run to line places an internal breakpoint at the target line and does a cont. To use it, right-click on the target line in the editor and choose *Run to Line*.

Step into selection is similar to the *Run to Line* command, with the difference that it is implemented by stepping. To use it, right-click on the target line in the editor and choose *Step into Selection*.

Step into(F5) steps into the next source line, regardless if the line is in another function or not. *Step over*(F6) on the other hand, does not step into functions.

Step out(F7) will place an internal breakpoint at the return address of the current function and do a cont.

Source stepping can be turned off by clicking *Instruction stepping mode*  in the *Debug* view. This will force *Step into* and *Step over* to step just one instruction. Be aware that *Step out* will still work as normal, since it does not depend on source information: it only needs the return address, which is stored in a register.

4.7.8. Terminals View

Terminals can be opened to interact with GRMON or to display the output from an application on the target system.

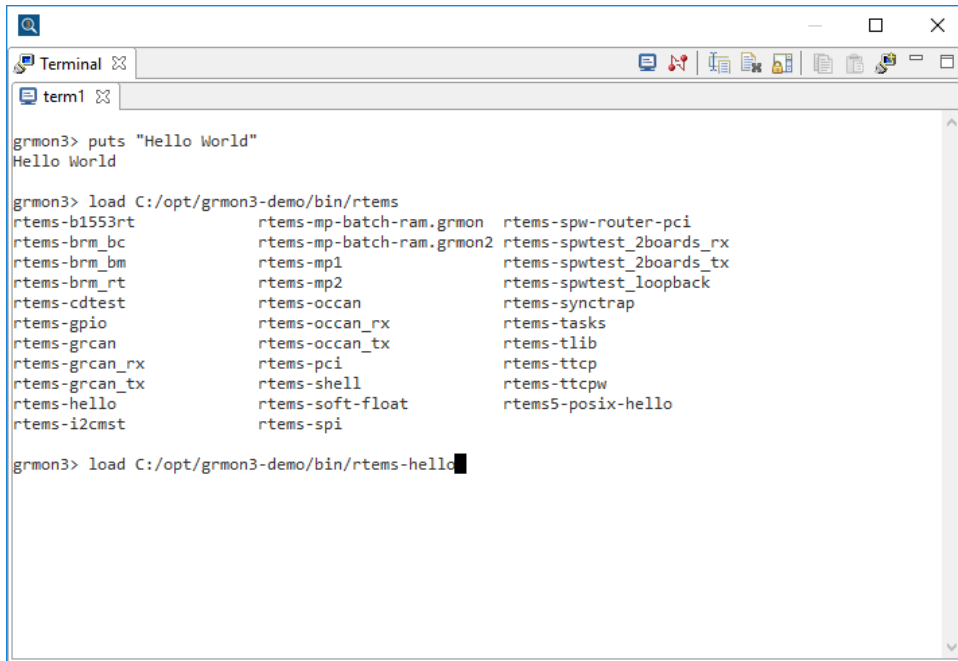



Figure 4.15. The terminals view showing a GRMON terminal

Opening a new terminal

To open a new terminal tab to either GRMON or an application on the target system, click the new terminal button  in the view's toolbar. A dialog will ask for the type of the terminal. This can be either *GRMON Terminal* or *Application terminal*. The GRMON terminal has no settings that can be made.

When opening an Application terminal the dialog ask for what UART of the system to use. Already busy UARTs will be grayed out and not selectable.

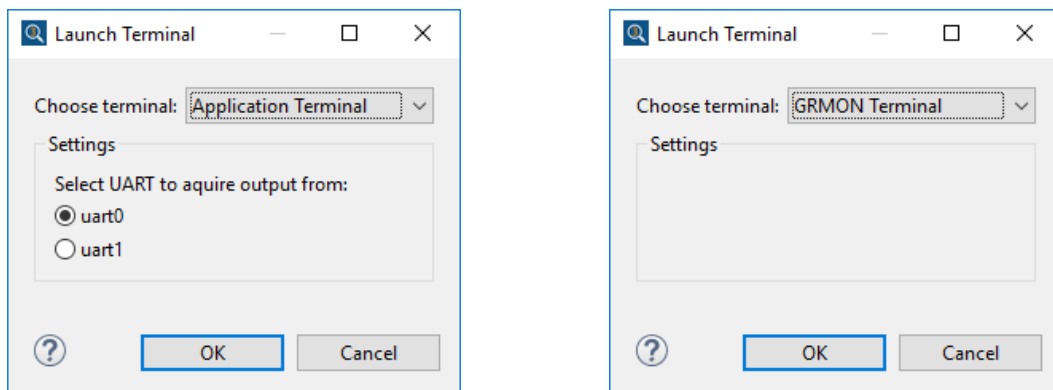


Figure 4.16. Opening a new terminal view

To open a whole new Terminals View click the new Terminal View button  in the view's toolbar.

Control characters in GRMON Terminal

The terminal accepts control characters such as Ctrl+C to break execution, and tab for auto completion, among many. Other examples are arrow up and down to access previous entries.

This also means that copy-paste can't be done via keyboard commands, and is only available from the context menu of the terminal tab.

For more information visit *Eclipse's TM Terminal site* [<http://www.eclipse.org/tm/doc/index.php>].

4.7.9. Memory View

The *Memory View* can be used to monitor and manipulate memory in the target system. It provides flexibility by allowing different presentations (renderings) of target memory area.

Memory View is available from the main menu under Window > Show View > Memory. When adding a new memory monitor, a dialog is displayed where the target address is specified.

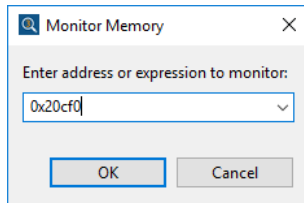


Figure 4.17. Opening a new message view

Memory content is modified by double-clicking a cell and typing in a new value. The new value is written to the target when enter is pressed.

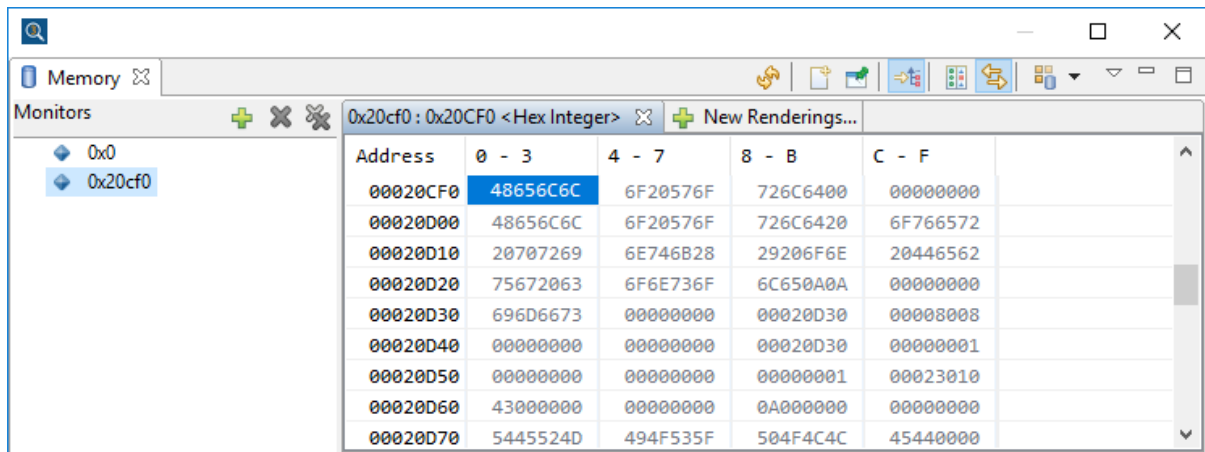


Figure 4.18. The memory view

The *Memory View* can be used to access any addressable locations as seen by the current debug link. For example IO registers in raw format. It is possible to specify the memory content presentation, for example ASCII string, by adding a new rendering from the *New Renderings...* tab. Endian can also be selected in the right-click context menu on a cell.

The *Memory View* is updated automatically at certain system events, for example when execution stops, or when a breakpoint is hit and after single-step by the user. This behavior can be disabled as described in Section 4.8.1, “*Memory view update*”.

Watchpoints can be created from a memory cell by right-clicking and choosing *Add Watchpoint*.

Even though GRMON GUI is CPU and context aware, the *Memory View* is global and always operate on physical addresses: no MMU translation is performed by the *Memory View*.

4.7.10. Breakpoints View

Breakpoints View keeps track of all breakpoints and watchpoints, regardless if they were created in the GUI or from the terminal. Here the user can modify or remove existing breakpoints and watchpoints and create new ones. To open the view, go to Window > Show View > Breakpoints.

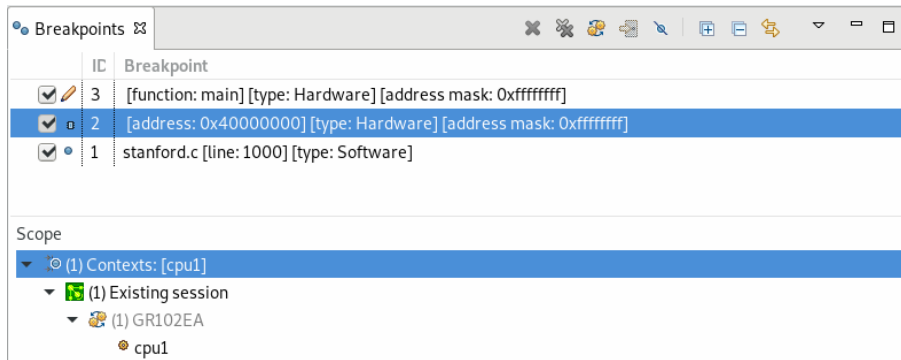











Figure 4.19. Breakpoints view

In Figure 4.19, “Breakpoints view” we see a soft breakpoint on line 1000 in stanford.c , a hard breakpoint at 0x40000000 , a write watchpoint on the function main . There are two other watchpoint types, read , and read/write . Selecting a breakpoint or watchpoint allows us to see its scope, which in the example above is cpu1.

A breakpoint/watchpoint can be removed by clicking . Clicking  removes all.

Clicking  disables all existing breakpoints/watchpoints, as well as any that are created, and they can not be enabled again until pushing the button again. This is useful if you temporarily want to skip all breakpoints/watchpoints.

Clicking  opens up a drop-down menu.

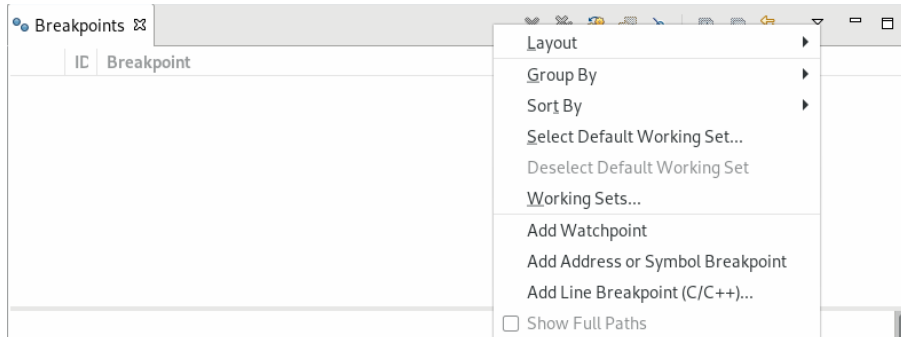


Figure 4.20. Drop down menu

Group By sorts the breakpoints in different ways, for instance by *Breakpoint Type* or *Scope*.

Add Address or Symbol Breakpoint opens up a breakpoint property page where the user can choose breakpoint type, either Regular (soft) or Hard, what address or symbol to place the breakpoint on and whether the breakpoint should be disabled or not. There is also an address mask which becomes available when choosing a Hard breakpoint.

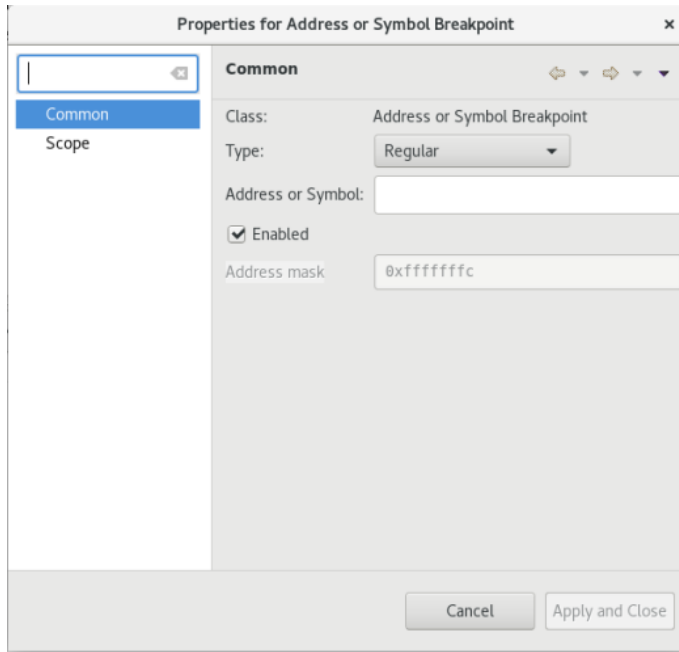


Figure 4.21. Breakpoint property page

Add Watchpoint opens up a watchpoint property page, which is similar to the breakpoint property page explained above, with the difference that you cannot choose the breakpoint type (watchpoints are always Hard type) and you can choose whether the watchpoint should be read, write or both.

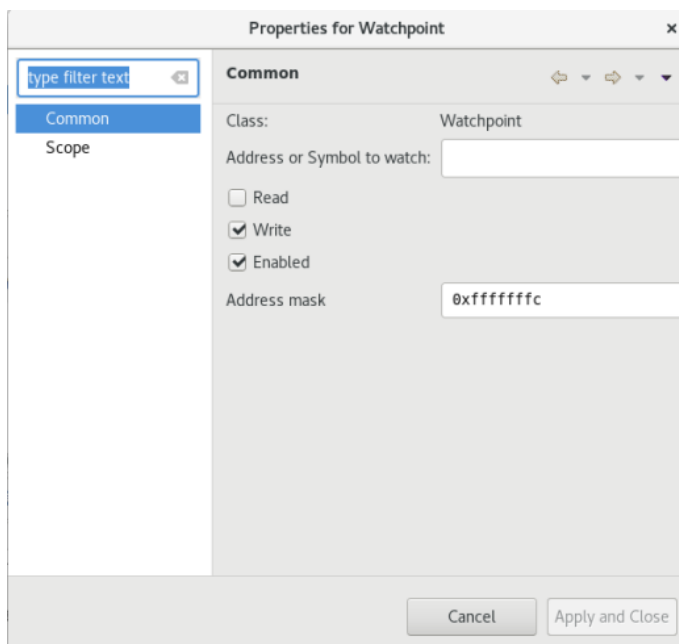


Figure 4.22. Watchpoint property page

Add Line breakpoint opens up a line breakpoint property page, where you choose breakpoint type, file and line number.

All breakpoints have a property called *Scope*, which describes which CPUs it is placed on. The default scope is based on the active context in the *Debug* view. If the system context is selected, the default scope will be on all CPUs except RTAs. If a CPU is selected, only that CPU will be included in the scope by default. It is equivalent to specifying a CPU when placing a breakpoint/watchpoint in the terminal, i.e. putting a watchpoint on main with

scope cpu0 has the same effect as typing "bp watch main cpu0" in the terminal. When creating a breakpoint, the scope can be changed via the breakpoint property page.

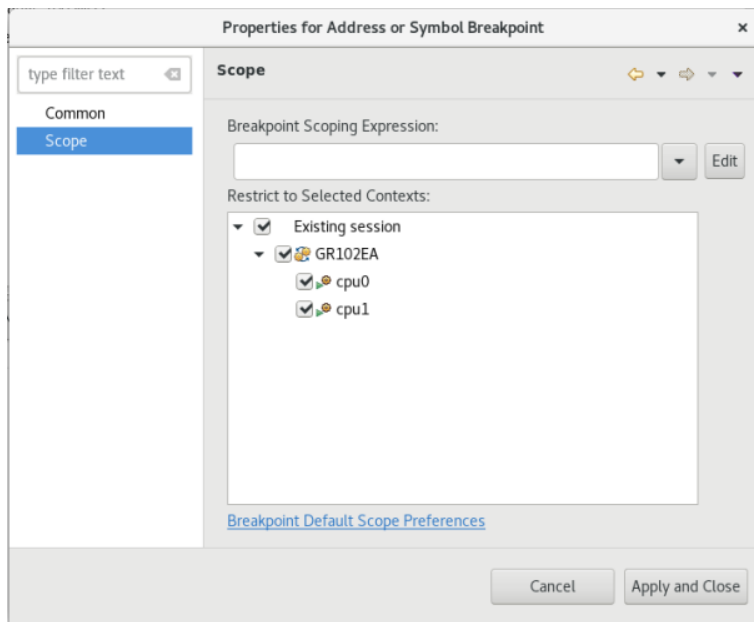


Figure 4.23. Scope page

Right-clicking in the Breakpoints view without selecting a breakpoint opens up a context menu.

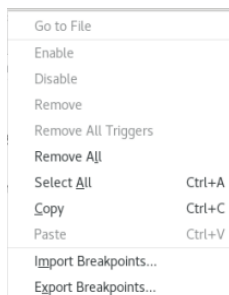


Figure 4.24. Breakpoints view context menu

Here the user can save any breakpoint to a *.bkpt* file by *Export Breakpoints* and import it at some other time with *Import Breakpoints*.

Right-clicking on a breakpoint gives the option to open up its address in the *Disassembly* view as well as in the *Memory* view.

4.7.11. Disassembly View

Disassembly View provides a convenient method to inspect the instruction memory of an application. This view has support for adding and removing breakpoints on a target address. It also provides, in combination with the *CPU Registers View*, a powerful method for inspecting how CPU state is updated on instruction level when stepping.

Disassembly View is available from the main menu under Window > Show View > Other > Debug > Disassembly.

The currently highlighted row represents the instruction which is to be issued when execution continues. Execution history is also highlighted in shaded color to represent instructions which have been recently executed. The right-most column of the *Disassembly View* describes the name of the function which the instruction belongs to, together with the offset from the function symbol.

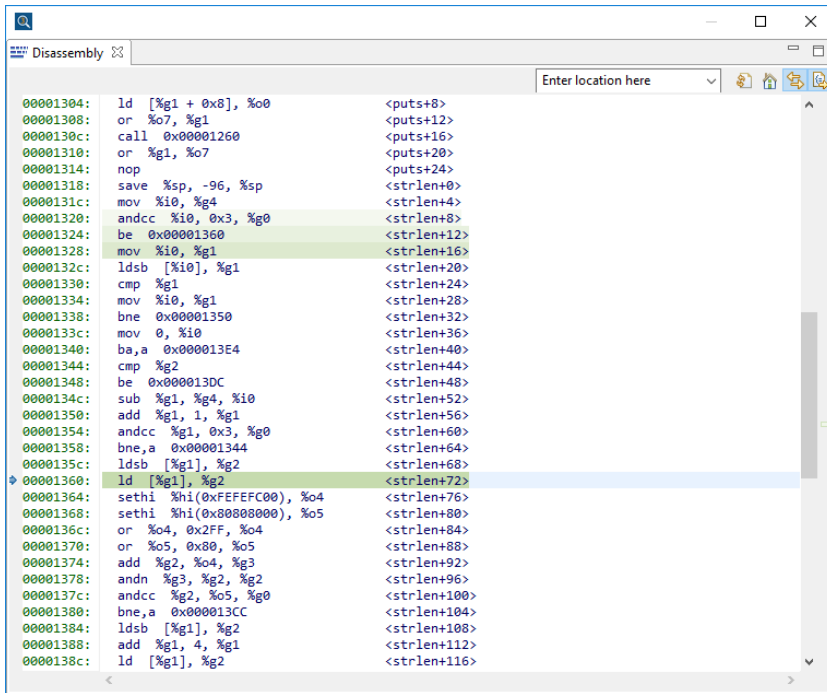


Figure 4.25. The disassembly view

Mixed mode can be toggled by pressing the *Show Source* . This mode shows both assembly instructions and the associated source code, if the source code is available. This mode is toggled on by default.

Disassembly View is updated each time execution stops, for example when stepping or a breakpoint. Methods for source stepping include:

- Step into the next line with the toolbar button named *Step Into*, also available via the **F5** short key. This stepping variant steps into functions.
- Step over the next line with the toolbar button named *Step Over*, also available via the **F6** short key. This stepping variant steps over functions.
- Step out to the previous function with the toolbar button named *Step Out*, also available via the **F7** short key.

Methods for instruction stepping include:

- Single-step using the **step** command in a Tcl terminal.
- Toggle instruction step mode on with *Instruction Stepping Mode* in the *Debug* view and click either the *Step Into* (**F5**), or *Step Over* tool button (**F6**). Note that *Step Out* is unchanged in instruction step mode.

A custom memory location can be disassembled by typing the address or symbol name in the box at the top of the *Disassembly View* labeled *Enter location here*. It can be used to disassemble instructions not related to the current CPU program counter.


```

Disassembly
Enter location here

Init:
40001278: save %sp, -128, %sp          <Init+0>
187      puts( "\n\n*** CLOCK TICK TEST ***" );
188
4000127c: sethi %hi(0x40024000), %o0   <Init+4>
40001280: call 0x40017dc4             <Init+8>
40001284: or %o0, 0x380, %o0         <Init+12>
189      time.year = 1988;
40001288: mov 0x7c4, %g1             <Init+16>
4000128c: st %g1, [%fp - 28]         <Init+20>
190      time.month = 12;
40001290: mov 0xc, %g1              <Init+24>
40001294: st %g1, [%fp - 24]         <Init+28>
191      time.day = 31;
40001298: mov 0x1f, %g1             <Init+32>
4000129c: st %g1, [%fp - 20]         <Init+36>
192      time.hour = 9;
400012a0: mov 0x9, %g1              <Init+40>
193      time.minute = 0;

```

Figure 4.26. Disassembly view mixed mode

The *Disassembly View* is context aware and will disassemble virtual memory as determined by the current CPU or thread context.

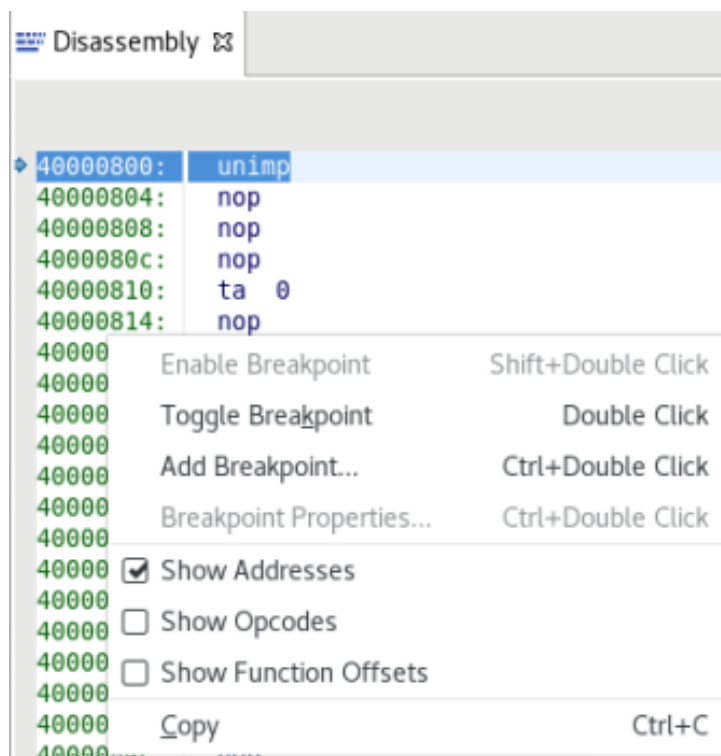


Figure 4.27. Disassembly view context menu

Right-clicking on an address opens a context menu, where the user can add a breakpoint. Choosing *Toggle Breakpoint* adds a soft breakpoint on the target address, this can also be achieved by double-clicking on the address. The breakpoint will be placed on the active context in the *Debug* view. If the system context is selected, the breakpoint will be placed on all its CPUs, except RTAs. Clicking instead on *Add Breakpoint*, or Ctrl+double-click opens a breakpoint property page with more breakpoint options, see Figure 4.21, “Breakpoint property page” in Breakpoints view. Existing breakpoints can be removed by double-clicking on them and enabling/disabling a breakpoint can be done by Shift+double-click or from the context menu.

Another useful feature in the context menu is *Show Opcodes*, which when clicked, creates a column to the right of the address column and displays the op code of the respective address.

In the context menu you have the option of not displaying the address column with *Show Addresses* and also the option to display the function offsets with *Show Function Offsets*. This will be displayed in a column to the right of the addresses, however this offset is already displayed in the column to the far right, regardless.

```

Disassembly
main:
0x40001250: 0x9de3bfa0 save %sp, -96, %
0x40001254: 0xb0102000 mov 0, %i0
0x40001258: 0x1110001a sethi %hi(0x4000
0x4000125c: 0x40000595 call 0x400028b0
0x40001260: 0x90122020 or %o0, 0x20, %r
0x40001264: 0x81c7e008 ret
0x40001268: 0x81e80000 restore
    
```

Figure 4.28. Breakpoints and op codes


In Figure 4.28, “Breakpoints and op codes” the op codes are enabled and can be seen in red. We also see two breakpoints: a hard breakpoint at 0x40001258 and a soft breakpoint at 0x40001264.



4.7.12. Messages View

The Messages view displays messages from the application that may be helpful for the user. The messages ranges from critical errors to less critical information of helpful nature. Messages can arise from many different situations, such when running a target image, or changing a setting.

Messages are sorted by the date they were created, showing newer messages on the top of the list.

To see the full message with all information, double-click the message in the view. This will open a dialog with all information available.

Copy messages to the clipboard by selecting one or many and press the Copy button  or press the copy command keyboard combination.

Remove one or more messages by selecting them and pressing the Remove button  or press delete on the keyboard. Remove all messages by clicking the Remove All button .

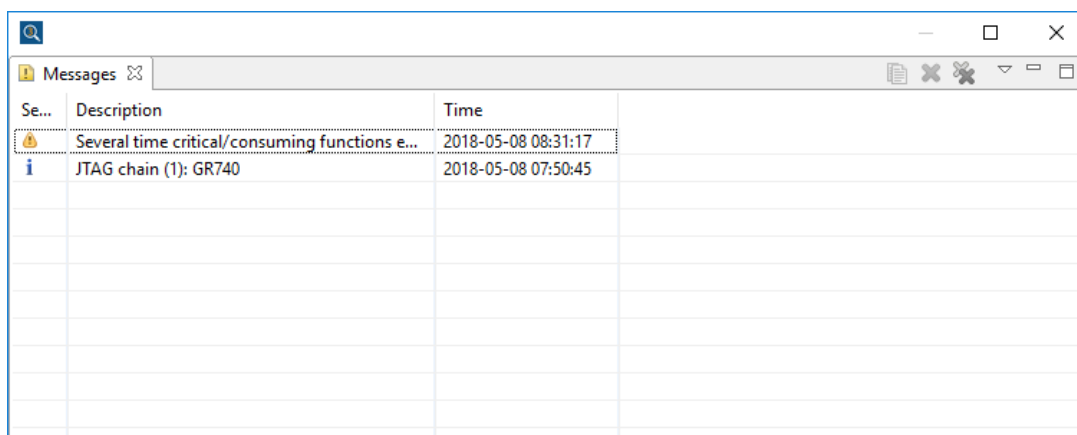


Figure 4.29. The message view

4.7.13. Executables View

Whenever an executable is loaded into GRMON, it will also appear in the *Executables View*, where the user can inspect its file contents. Double-clicking a file will open it in the *Source Editor*. It is also possible to add executables right into the *Executables View* without having to load them into GRMON.

Binaries built with DWARF5 are not supported.

Executables View can be opened from the main menu under Window > Show View > Executables.

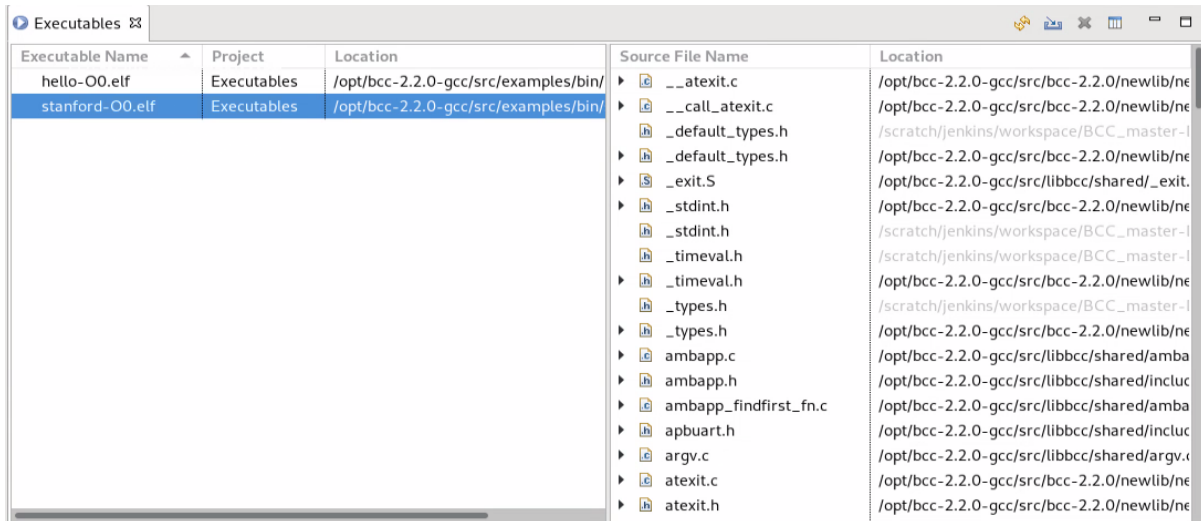


Figure 4.30. The executables view showing two loaded executables and one of their file contents.

4.7.14. Outline View

The *Outline View* shows all symbols of the currently active file in the *Source Editor*. Clicking a symbol will give the *Source Editor* focus to it.

Outline View can be opened from the main menu under Window > Show View > Outline View.

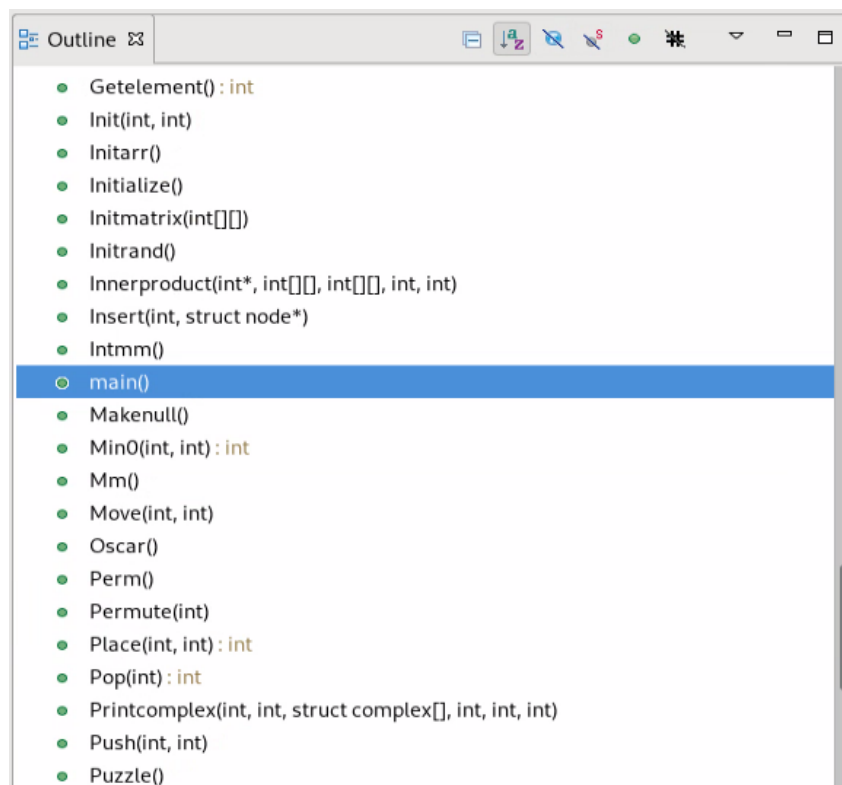


Figure 4.31. The outline view showing the symbols of the currently active file, stanford.c in this case.

4.7.15. Variables View

The *Variables View* shows local symbols of the currently active stack frame. The three different columns display name, type and value of the variable, respectively, and the value can be changed by the user. The currently selected variable will display size and also address or register, if applicable, in the lower part of the view.

Variables View can be opened from the main menu under Window > Show View > Variables View.

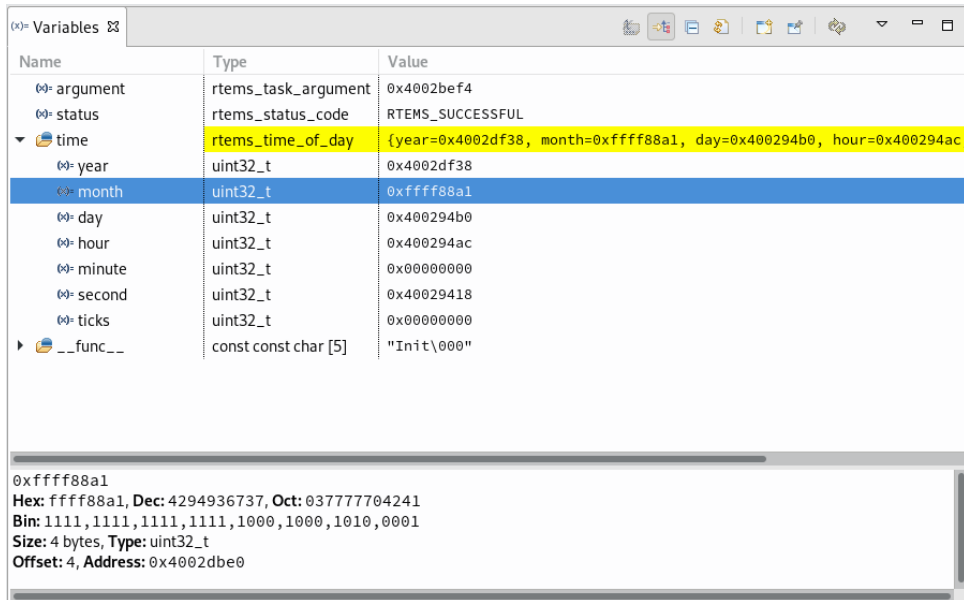


Figure 4.32. The Variables view showing the symbols of the currently active stack frame. Here, month, an uint32_t is selected, which belongs to the time variable, which is of rtems_time_of_day type. We see that month has value 0xffff88a1, is 4 bytes, has an offset of 4 from the parent type and resides at address 0x4002dbe0.

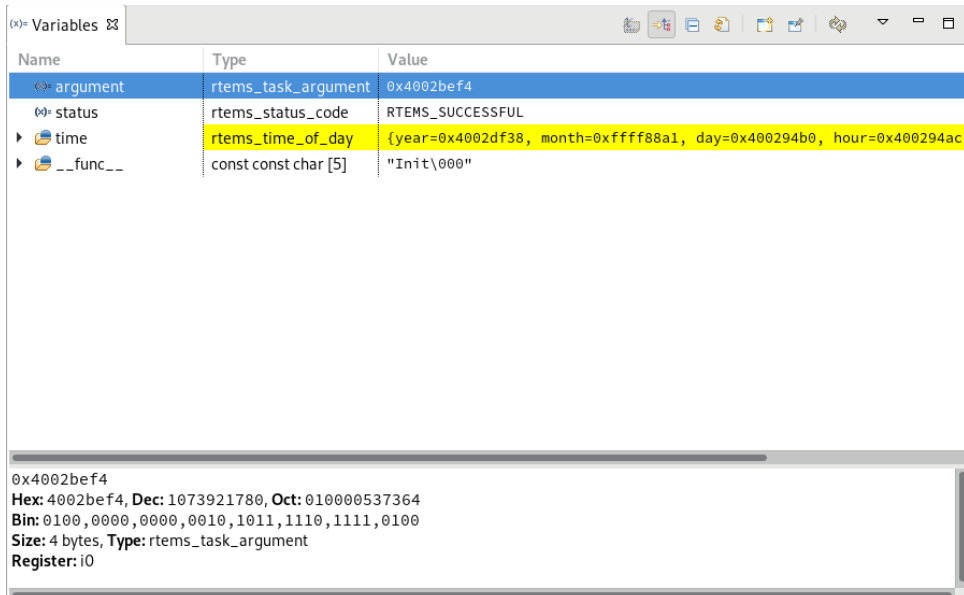


Figure 4.33. Same example as the previous figure. Now argument is selected, which is of rtems_task_argument type, has value 0x4002bef4, 4 bytes in size and resides in the i0 register.

4.7.16. Expressions View

In the *Expressions view*, the user can choose variables to keep track of (called Expressions), making it a complement to the *Variables view*, which only keeps track of local stack variables. The easiest way of creating an Expression is to find the target variable in the *Source Editor*, right-click and choose *Add Watch Expression*. It is

also possible to add Expressions with the *Create a new watch expression* button. The values of expressions can be changed by the user. Keep in mind that an expression might not be able to be resolved if it goes out of scope.

Expressions View can be opened from the main menu under Window > Show View > Expressions.

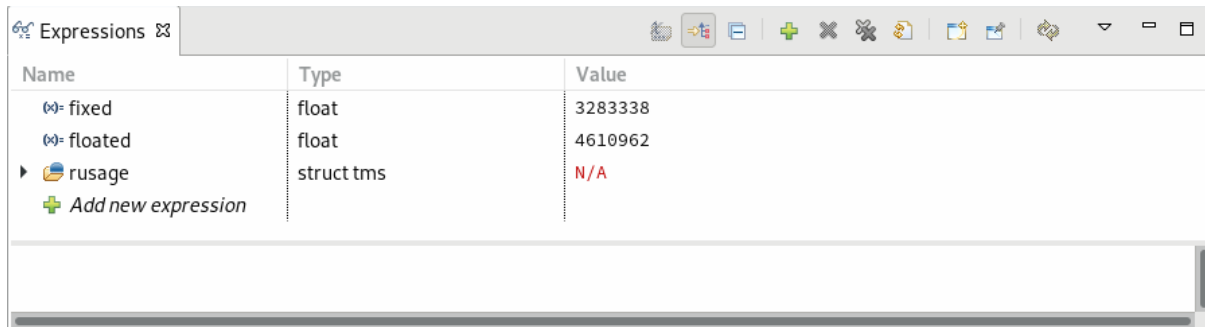


Figure 4.34. The expressions view showing three variables, of which one is out of scope.

4.8. Target communication

4.8.1. Memory view update

GRMON4 GUI normally updates the memory views when target execution stops, for example on a breakpoint or after single step. This can generate a large amount of traffic on a slow debug link in combination with large memory views. To save bandwidth on the debug link, there is an option named *Disable auto-updating memory view after execution*, available in the connection dialog *GUI* tab. The corresponding command-line option is `-tcf-nomemupdate`.

4.9. Limitations

This section describes limitation of GRMON4 GUI and areas of incompatibility with GRMON4 CLI.

C/C++ debugging

The C/C++ features works best with unoptimized (O0) code. With optimized code, symbols might be optimized away such that they can't be inspected.

Name mangling can occur for C++ code.

DWARF5 is not fully supported. The *Executables View* can not parse binaries built with DWARF5.

4.10. Troubleshooting the GUI

This section lists some useful tips for troubleshooting the GUI.

The GUI doesn't start

Check the log for errors. On Linux it is located in `~/.grmon-4.0/workspace/.metadata/.log` and in `$APPDATA$Frontgrade Gaisler/GRMON/4.0/workspace/.metadata/.log` for Windows.

Clear the workspace folder and try again. This folder is located in `~/.grmon-4.0/workspace/` for Linux and `$APPDATA$Frontgrade Gaisler/GRMON/4.0/workspace` for Windows.

Check your Java version. Java 11 (64-bit) or later is required.

The following error is displayed when trying to launch the GUI: "The configuration area at '/path-to-workspace/config' could not be created. Please choose a writable location using the '-configuration' command line option."

This error message is generated by the Eclipse launcher that launches the GUI, if a configuration folder can not be created. Please disregard the `-configuration` option; it is an option to the Eclipse launcher and not an option to GRMON.

To resolve the error, open up `share/grmon/gui<BITS>/configuration/config.ini` and change `'osgi.configuration.area'` into whatever folder you like.

5. Debug link

GRMON supports several different links to communicate with the target board. However all of the links may not be supported by the target board. Refer to the board user manual to see which links that are supported. There are also boards that have built-in adapters.

Refer to the board user manual to see which links that are supported.

The default communication link between GRMON and the target system is the host's serial port connected to a serial debug interface (AHBUART) of the target system. Connecting using any of the other supported link can be performed by using the switches listed below. More switches that may affect the connection are listed at each subsection.

<code>-altjtag</code>	Connect to the target system using Altera Blaster cable (USB or parallel).
<code>-eth</code>	Connect to the target system using Ethernet. Requires the EDCL core to be present in the target system.
<code>-digilent</code>	Connect to the target system Digilent HS1/HS2/HS3/SMT2/SMT3 cable.
<code>-ftdi</code>	Connect to the target system using a JTAG cable based on a FTDI chip.
<code>-gresb</code>	Connect to the target system through the GRESB bridge. The target needs a SpW core with RMAP.
<code>-jtag</code>	Connect to the target system the JTAG Debug Link using Xilinx Parallel Cable III or IV.
<code>-usb</code>	Connect to the target system using the USB debug link. Requires the GRUSB_DCL core to be present in the target.
<code>-xilusb</code>	Connect to the target system using a Xilinx Platform USB cable.
<code>-uart <device></code>	Connect to the target system using a serial cable.
<code>-user</code>	Connect to the target system using a custom user defined library.

8-/16-bit access to the target system is only supported by the JTAG debug links, all other interfaces access subwords using read-modify-write. All links supports 32-bit accesses. 8-bit access is generally not needed. An example of when it is needed is when programming a 8 or 16-bit flash memory on a target system *without* a LEON CPU available. Another example is when one is trying to access cores that have byte-registers, for example the CAN_OC core, but almost all GRLIB cores have word-registers and can be accessed by any debug link.

The speed of the debug links affects the performance of GRMON. It is most noticeable when loading large applications, for example Linux or VxWorks. Another case when the speed of the link is important is during profiling, a faster link will increase the number of samples. See Table 5.1 for a list of estimated speed of the debug links.

Table 5.1. Estimated debug link application download speed

Name	Estimated speed
UART	~100 kbit/s
JTAG (Parallel port)	~200 kbit/s
JTAG (USB)	~1 Mbit/s
GRESB	~25 Mbit/s
USB	~30 Mbit/s
Ethernet	~35 Mbit/s

5.1. UART debug link

To attach GRMON using the AHBUART debug link, first connect a cable between the UART connectors on target board and the host system. Then power-up and reset the target board and start GRMON with the `-uart` option. Use the `-uart <device>` option in case the target is not connected to the first UART port of your host. On some hosts, it might be necessary to lower the baud rate in order to achieve a stable connection to the target. In this case, use the `-baud` switch with the 57600 or 38400 options. Below is a list of start-up switches applicable for the AHBUART debug link.

Extra options for UART debug link:

`-uart <device>`

By default, GRMON communicates with the target using the first uart port of the host. This can be overridden by specifying an alternative device. Device names depend on the host operating system. On Linux systems serial devices are named as `/dev/tty##` and on Windows they are named `\\.com#`.

`-baud <baudrate>`

Use baud rate for the DSU serial link. By default, 115200 baud is used. Possible baud rates are 9600, 19200, 38400, 57600, 115200, 230400, 460800. Rates above 115200 need special uart hardware on both host and target.

When using an USB-to-Serial adapter based on FTDI chips, there is a latency timer that will be a bottleneck, especially when reading small amounts of data very often. For example when the I/O Forwarding is enabled or collecting profiling information. In Linux this timer will be adjusted automatically by GRMON, but in Windows it must be set manually. Open the Windows "Device Manager" and locate the serial port device. Right-click on the device and select properties. In the tab "Port Settings", push the "Advanced" button. Set the "Latency Timer" to lowest possible value and press the button "OK".

5.2. Ethernet debug link

If the target system includes a GRETH core with EDCL enabled then GRMON can connect to the system using Ethernet. The default network parameters can be set through additional switches.

Extra options for Ethernet:

`-eth [<ipnum>][:<port>]`

Use the Ethernet connection and optionally use `ipnum` for the target system IP number and/or `:port` to select which UDP port to use. Default IP address is 192.168.0.51 and the port number is random.

`-edclmem <kB>`

The EDCL hardware can be configured with different buffer size. Use this option to force the buffer size (in KB) used by GRMON during EDCL debug-link communication. By default the GRMON tries to autodetect the best value. Valid options are: 1, 2, 4, 8, 16, 32, 64.

`-edclus <us>`

Increase the EDCL timeout before resending a packet. Use this option if you have a large network delays.

The default IP address of the EDCL is normally determined at synthesis time. The IP address can be changed using the `edcl` command. If more than one core is present in the system, then select core by appending the name. The name of the core is listed in the output of `info sys`.

Note that if the target is reset using the reset signal (or power-cycled), the default IP address is restored. The `edcl` command can be given when GRMON is attached to the target with any interface (serial, JTAG, PCI ...), allowing to change the IP address to a value compatible with the network type, and then connect GRMON using the EDCL with the new IP number. If the `edcl` command is issued through the EDCL interface, GRMON must be restarted using the new IP address of the EDCL interface. The current IP address is also visible in the output from `info sys`.

```
grmon3> edcl
Device index: greth0
Edcl ip 192.168.0.51, buffer 2 kB

grmon3> edcl greth1
Device index: greth1
Edcl ip 192.168.0.52, buffer 2 kB

grmon3> edcl 192.168.0.53 greth1
Device index: greth1
Edcl ip 192.168.0.53, buffer 2 kB

grmon3> info sys greth0 greth1
greth0    Frontgrade Gaisler GR Ethernet MAC
          APB: FF940000 - FF980000
          IRQ: 24
          edcl ip 192.168.0.51, buffer 2 kbyte
greth1    Frontgrade Gaisler GR Ethernet MAC
          APB: FF980000 - FF9C0000
          IRQ: 25
          edcl ip 192.168.0.53, buffer 2 kbyte
```

5.3. JTAG debug link

The subsections below describe how to connect to a design that contains a JTAG AHB debug link (AHBJTAG). The following commandline options are common for all JTAG interfaces. If more than one cable of the same type is connected to the host, then you need to specify which one to use, by using a commandline option. Otherwise it will default to the first it finds.

Extra options common for all JTAG cables:

- jtaglist
List all available cables of the selected type and exit application.
- jtagcable <n>
Specify which cable to use if more than one is connected to the computer. If only one cable of the same type is connected to the host computer, then it will automatically be selected. It's also used to select parallel port.
- jtagserial <sn>
Specify which cable to use by serial number if more than one is connected to the computer.
- jtagdevice <n>
Specify which device in the chain to debug. Use if more than one is device in the chain is debuggable.
- jtagcomver <version>
Specify JTAG debug link version.
- jtagretry <num>
Set the number of retries.
- jtagcfg <filename>
Load a JTAG configuration file, defining unknown devices.

JTAG debug link version. The JTAG interface has in the past been unreliable in systems with very high bus loads, or extremely slow AMBA AHB slaves, that lead to GRMON reading out AHB read data before the access had actually completed on the AHB bus. Read failures have been seen in systems where the debug interface needed to wait hundreds of cycles for an AHB access to complete. With version 1 of the JTAG AHB debug link the reliability of the debug link has been improved. In order to be backward compatible with earlier versions of the debug link, GRMON cannot use all the features of AHBJTAG version 1 before the debug monitor has established that the design in fact contains a core with this version number. In order to do so, GRMON scans the plug and play area. However, in systems that have the characteristics described above, the scanning of the plug and play area may fail. For such systems the AHBJTAG version assumed by GRMON during plug and play scanning can be set with the switch `-jtagcomver<version>`. This will enable GRMON to keep reading data from the JTAG AHB debug interface until the AHB access completes and valid data is returned. Specifying the version in systems that have AHBJTAG version 0 has no benefit and may lead to erroneous behavior. The option `-jtagretry<num>` can be used to set the number of attempts before GRMON gives up.

JTAG chain devices. If more than one device in the JTAG chain are recognized as debuggable (FPGAs, ASICs etc), then the device to debug must be specified using the commandline option `-jtagdevice`. In addition, all devices in the chain must be recognized. GRMON automatically recognizes the most common FPGAs, CPLDs, proms etc. But unknown JTAG devices will cause GRMON JTAG chain initialization to fail. This can be solved by defining a JTAG configuration file. GRMON is started with `-jtagcfg` switch. An example of JTAG configuration file is shown below. If you report the device ID and corresponding JTAG instruction register length to support@gaisler.com, then the device will be supported in future releases of GRMON.

```
# JTAG Configuration file
# Name      Id          Mask          Ir length  Debug I/F  Instr. 1  Instr. 2
xc2v3000    0x01040093  0x0fffffff    6          1          0x2      0x3
xc18v04     0x05036093  0x0ffefffff   8          0          0        0
ETH         0x103cb0fd  0x0fffffff    16         0          0        0
```

Each line consists of device name, device id, device id mask, instruction register length, debug link and user instruction 1 and 2 fields, where:

Name	String with device name
Id	Device identification code
Mask	Device id mask is ANDed with the device id before comparing with the identification codes obtained from the JTAG chain. Device id mask allows user to define a range of identification codes on a single line, e.g. mask 0xffffffff will define all versions of a certain device.

Ir length	Length of the instruction register in bits
Debug I/F	Set debug link to 1 if the device implements JTAG Debug Link, otherwise set to 0.
Instr. 1	Code of the instruction used to access JTAG debug link address/command register (default is 0x2). Only used if debug link is set to 1.
Instr. 2	Code of the instruction used to access JTAG debug link data register (default is 0x3). Used only if debug link is set to 1.

The JTAG configuration file can not be used with Altera blaster cable (`-altjtag`).

5.3.1. Xilinx parallel cable III/IV

If target system has the JTAG AHB debug link, GRMON can connect to the system through Xilinx Parallel Cable III or IV. The cable should be connected to the host computers parallel port, and GRMON should be started with the `-jtag` switch. Use `-jtagcable` to select port. On Linux, you must have read and write permission, i.e. make sure that you are a member of the group 'lp'. I.a. on some systems the Linux module lp must be unloaded, since it uses the port.

Extra options for Xilinx parallel cable:

- `-jtag`
Connect to the target system using a Xilinx parallel cable III/IV cable

5.3.2. Xilinx Platform USB cable

JTAG debugging using the Xilinx USB Platform cable is supported on Linux and Windows 7 systems. The platform cable models DLC9G and DLC10 are supported. The legacy model DLC9 is not supported. GRMON should be started with `-xilusb` switch. Certain FPGA boards have a USB platform cable logic implemented directly on the board, using a Cypress USB device and a dedicated Xilinx CPLD. GRMON can also connect to these boards, using the `--xilusb` switch.

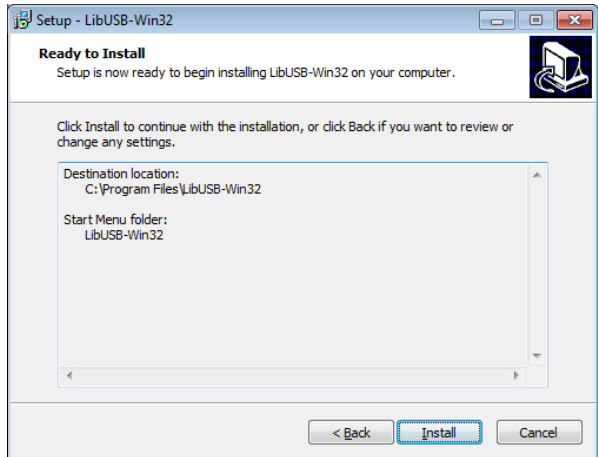
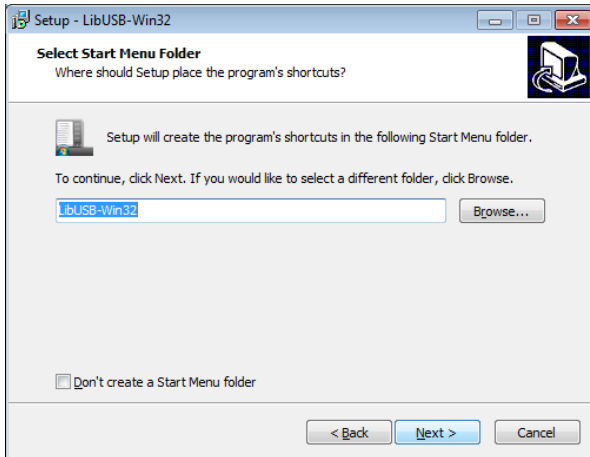
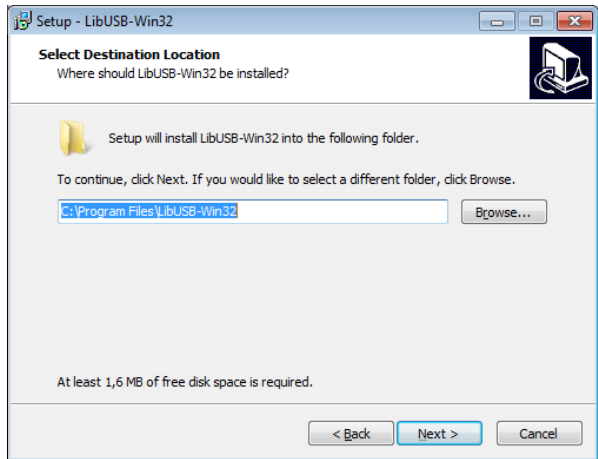
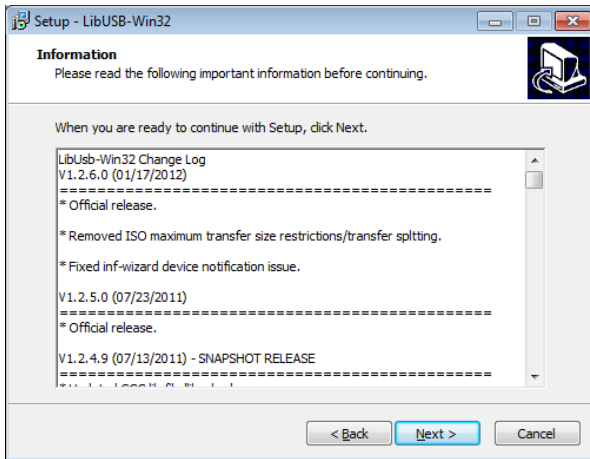
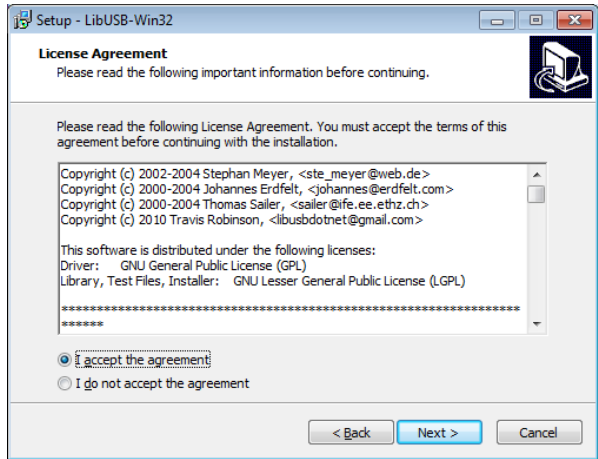
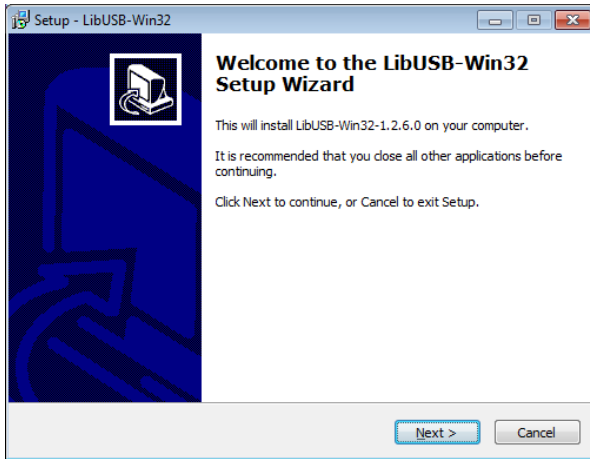
Extra options for Xilinx USB Platform cable:

- `-xilusb`
Connect to the target system using a Xilinx USB Platform cable.
- `-xilmhz [12/6/3/1.5/0.75]`
Set Xilinx Platform USB frequency. Valid values are 12, 6, 3, 1.5 or 0.75 MHz. Default is 3 MHz.

On Linux systems, the Xilinx USB drivers must be installed by executing `./setup_pcusb` in the ISE `bin/bin/lin` directory (see ISE documentation). I.a. the program `fxload` must be available in `/sbin` on the used host, and `libusb` must be installed.

On Windows hosts follow the instructions below. The USB cable drivers should be installed from Xilinx ISE, ISE Webpack or Vivado Lab Tools. Xilinx ISE 9.2i or later is required, or Xilinx Vivado Lab Tools 2017.4. Then install the *filter driver*, from the `libusb-win32` project [<http://libusb-win32.sourceforge.net>], by running `install-filter-win.exe` from the `libusb` package.

1. Install the ISE, ISE-Webpack, iMPACT or Vivado Lab Tools by following their instructions. This will install the drivers for the Xilinx Platform USB cable. Xilinx ISE 9.2i or later is required, or Vivado 2017.4. After the installation is complete, make sure that the Xilinx tools can find the Platform USB cable.
2. Then run `libusb-win32-devel-filter-1.2.6.0.exe`, which can be found in the folder '`<grmon-ver>/share/grmon/`', where `<grmon-ver>` is the path to the extracted win32 or win64 folder from the the GRMON archive. This will install the libusb filter driver tools. Step through the installer dialog boxes as seen in Figure 5.1 until the last dialog. The `libusb-win32-devel-filter-1.2.6.0.exe` installation is compatible with both 64-bit and 32-bit Windows.
3. Make sure that **Launch filter installer wizard** is checked, then press **Finish**. The wizard can also be launched from the start menu.



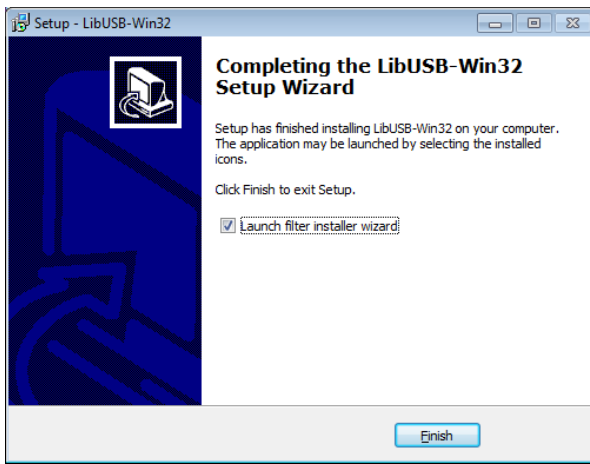


Figure 5.1.

4. At the first dialog, as seen in Figure 5.2, choose 'Install a device filter' and press **Next**.
5. In the second dialog, mark the Xilinx USB cable. You can identify it either by name Xilinx USB Cable in the 'Description' column or `vid:03fd` in the 'Hardware ID' column. Then press **Install** to continue.
6. Press **OK** to close the pop-up dialog and then **Cancel** to close the filter wizard. You should now be able to use the Xilinx Platform USB cable with both GRMON and iMPACT.

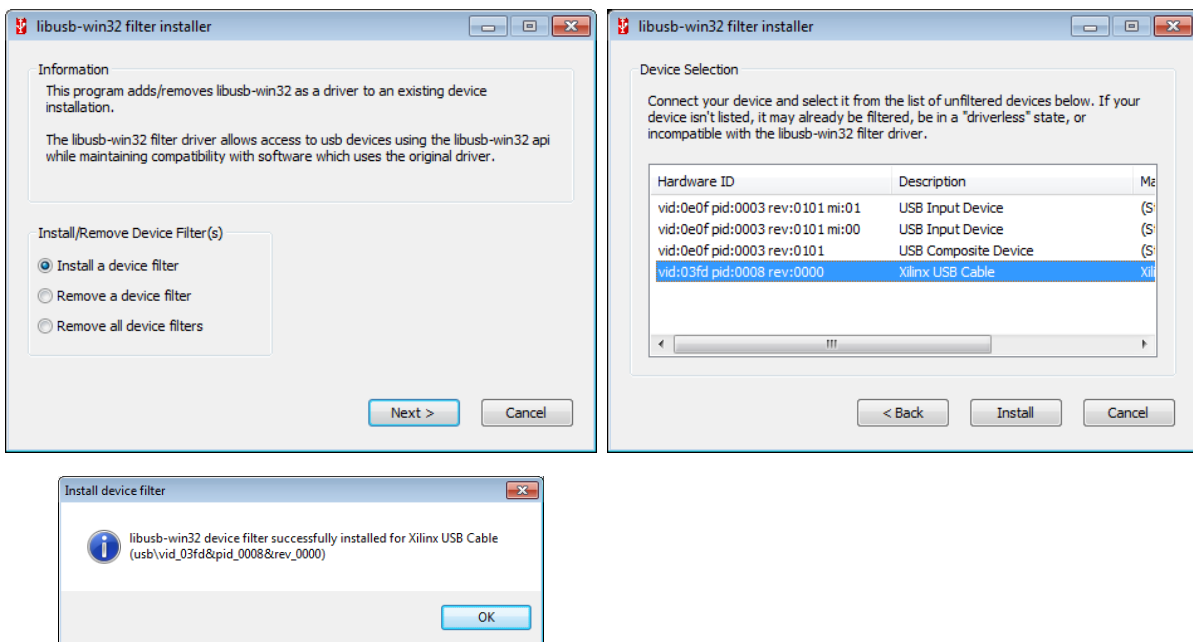


Figure 5.2.

The *libusb-win32 filter installer wizard* may have to be run again if the Xilinx Platform USB cable is connected to another USB port or through a USB hub.

5.3.3. Altera USB Blaster or Byte Blaster

For GRLIB systems implemented on Altera devices GRMON can use USB Blaster or Byte Blaster cable to connect to the system. GRMON is started with `-altjtag` switch. Drivers are included in the the Altera Quartus software, see Actel's documentation on how to install on your host computer.

The connection requires Altera Quartus version less then or equal to 13.

On Linux systems, the path to Quartus shared libraries has to be defined in the `LD_LIBRARY_PATH` environment variable, i.e.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/quartus/linux
$ grmon -altjtag

GRMON3 LEON debug monitor v3.0.0 32-bit professional version
...
```

On Windows, the path to the Quartus binary folder must be added to the environment variable `PATH`, see Appendix F, *Appending environment variables* in how to do this. The default installation path to the binary folder should be similar to `C:\altera\11.1sp2\quartus\bin`, where *11.1sp2* is the version of Quartus.

Extra options for Altera Blaster:

```
-altjtag
```

Connect to the target system using Altera Blaster cable (USB or parallel).

5.3.4. FTDI FT4232/FT2232

JTAG debugging using a FTDI FT2232/FT4232 chip in MPSSE-JTAG-emulation mode is supported in Linux and Windows. GRMON has support for two different back ends, one based on libftdi 0.20 and the other based on FTDI's official d2xx library.

When using Windows, GRMON will use the d2xx back end per default. FTDI's D2XX driver must be installed. Drivers and installation guides can be found at FTDI's website [<http://www.ftdichip.com>].

In Linux, the libftdi back end is used per default. The user must also have read and write permission to the device file. This can be achieved by creating a udev rules file, `/etc/udev/rules.d/51-ftdi.rules`, containing the lines below and then reconnect the USB cable.

```
ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6011", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6014", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6040", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6041", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6042", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6043", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6044", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6045", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="6048", MODE="666"
ATTR{idVendor}=="0403", ATTR{idProduct}=="cfe8", MODE="666"
```

Extra options for FTDI:

```
-ftdi [libftdi|d2xx]
```

Connect to the target system using a JTAG cable based on a FTDI chip. Optionally a back end can be specified. Defaults to libftdi on Linux and d2xx on Windows

```
-ftdidetach
```

On Linux, force the detachment of any kernel drivers attached to the USB device.

```
-ftdimhz <mhz>
```

Set FTDI frequency divisor. Values between 0.0 and 30.0 are allowed (values higher than 6.0 MHz are hardware dependent) The frequency will be rounded down to the closest supported frequency supported by the hardware. Default value of *mhz* is 1.0 MHz

```
-ftdivid <vid>
```

Set the vendor ID of the FTDI device you are trying to connect to. This can be used to add support for 3rd-party FTDI based cables.

```
-ftdipid <pid>
```

Set the product ID of the FTDI device you are trying to connect to. This can be used to add support for 3rd-party FTDI based cables.

```
-ftdigpio <val>
```

Set the GPIO signals of the FTDI device. The lower 16bits sets the level of the GPIO and the upper bits set the direction.

Bits 0-3	Reserved
Bits 4-7	GPIOL 0-3 level
Bits 8-15	GPIOH 0-7 level
Bits 16-19	Reserved
Bits 20-23	GPIOL 0-3 direction

5.3.5. Amontec JTAGkey

The Amontec JTAGkey is based on a FTDI device, therefore see Section 5.3.4, “FTDI FT4232/FT2232” about FTDI devices on how to connect. Note that the user does *not* need to specify VID/PID for the Amontec cable. The drivers and installation guide can be found at Amontec's website [<http://www.amontec.com>].

5.3.6. Actel FlashPro 3/3x/4/5

Support for Actel FlashPro 3/3x/4/5 is only supported by the professional version.

On Windows 32-bit, JTAG debugging using the Microsemi FlashPro 3/3x/4 is supported for GRLIB systems implemented on Microsemi devices. This also requires FlashPro 11.4 software or later to be installed on the host computer (to be downloaded from Microsemi's website). Windows support is detailed at the website. GRMON is started with the `-fpro` switch.

JTAG debugging using the Microsemi FlashPro 5 cable is supported on both Linux and Windows, for GRLIB systems implemented on Microsemi devices, using the FTDI debug link. See Section 5.3.4, “FTDI FT4232/FT2232” about FTDI devices on how to connect. Note that the user does *not* need to specify VID/PID for the FlashPro 5 cable. This also requires FlashPro 11.4 software or later to be installed on the host computer (to be downloaded from Microsemi's website).

Extra options for Actel FlashPro:

`-fpro`

Connect to the target system using the Actel FlashPro cable. (Windows)

5.3.7. Digilent HS1/HS2/HS3/SMT2/SMT3

JTAG debugging using a Digilent JTAG HS1/HS2/HS3/SMT2/SMT3 cable is supported on Linux and Windows systems. Start GRMON with the `-digilent` switch to use this interface.

On Windows hosts, the Digilent Adept System software must be installed on the host computer, which can be downloaded from Digilent's website.

On Linux systems, the Digilent Adept Runtime must be installed on the host computer, which can be downloaded from Digilent's website. The Adept v2.19.2 Runtime supports the Linux distributions listed below.

CentOS 6 / Red Hat Enterprise Linux 6

CentOS 7 / Red Hat Enterprise Linux 7

Ubuntu 14.04

Ubuntu 16.10

Ubuntu 18.04

Extra options for Digilent cables:

`-digilent`

Connect to the target system using the Digilent cable.

`-digifreq <hz>`

Set Digilent cable frequency in Hz. Default is 1 MHz.

5.4. USB debug link

GRMON can connect to targets equipped with the GRUSB_DCL core using the USB bus. To do so start GRMON with the `-usb` switch. Both USB 1.1 and 2.0 are supported. Several target systems can be connected to a single host at the same time. GRMON scans all the USB buses and claims the first free USB DCL interface. If the first target system encountered is already connected to another GRMON instance, the interface cannot be claimed and the bus scan continues.

On Linux the GRMON binary must have read and write permission. This can be achieved by creating a udev rules file, `/etc/udev/rules.d/51-gaisler.rules`, containing the line below and then reconnect the USB cable.

SUBSYSTEM=="usb", ATTR{idVendor}=="1781", ATTR{idProduct}=="0aa0", MODE="666"

On Windows a driver has to be installed. The first the time the device is plugged in it should be automatically detected as an unknown device, as seen in Figure 5.3. Follow the instructions below to install the driver.

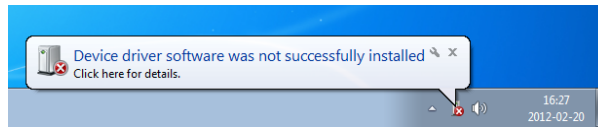


Figure 5.3.

1. Open the device manager by writing '`mmc devmgmt.msc`' in the run-field of the start menu.
2. In the device manager, find the unknown device. Right click on it to open the menu and choose '**Update Driver Software...**' as Figure 5.4 shows.

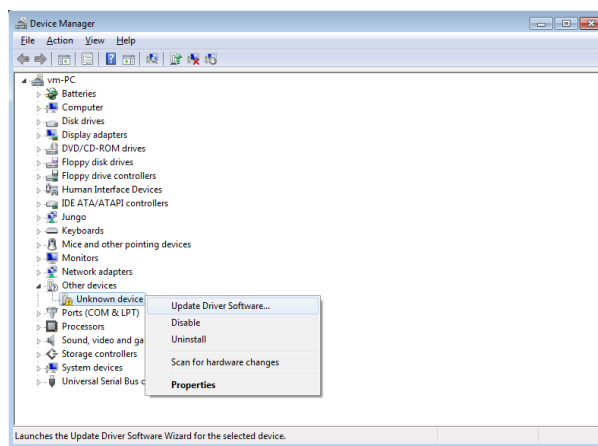
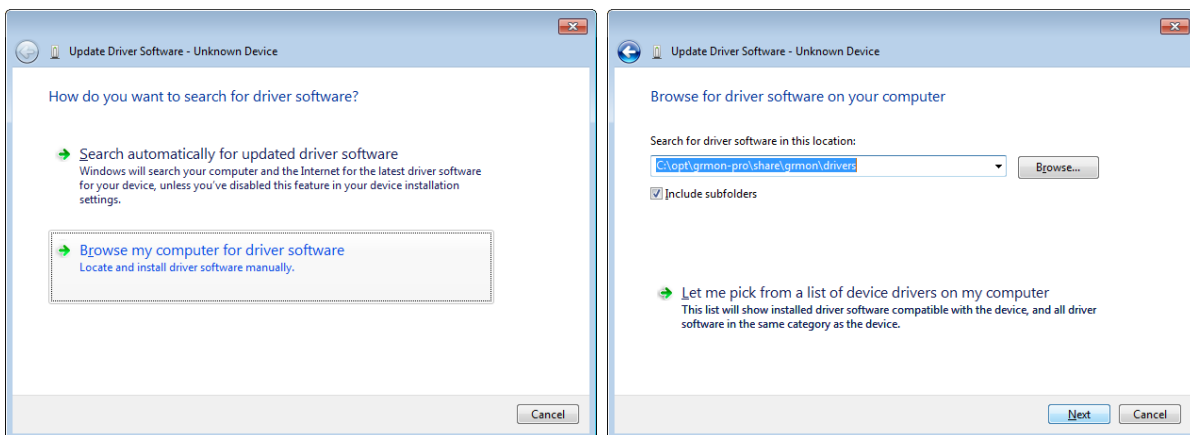


Figure 5.4.

3. In the dialog that open, the first image in Figure 5.5, choose '**Browse my computer for driver software**'.
4. In the next dialog, press the **Browse** button and locate the path to `<grmon-win32>/share/grmon/drivers`, where grmon-win32 is the path to the extracted win32 folder from the the GRMON archive. Press **Next** to continue.
5. A warning dialog might pop-up, like the third image in Figure 5.5. Press '**Install this driver software anyway**' if it shows up.
6. Press '**Close**' to exit the dialog. The USB DCL driver is now installed and GRMON should be able to connect to the target system using the USB DCL connection.



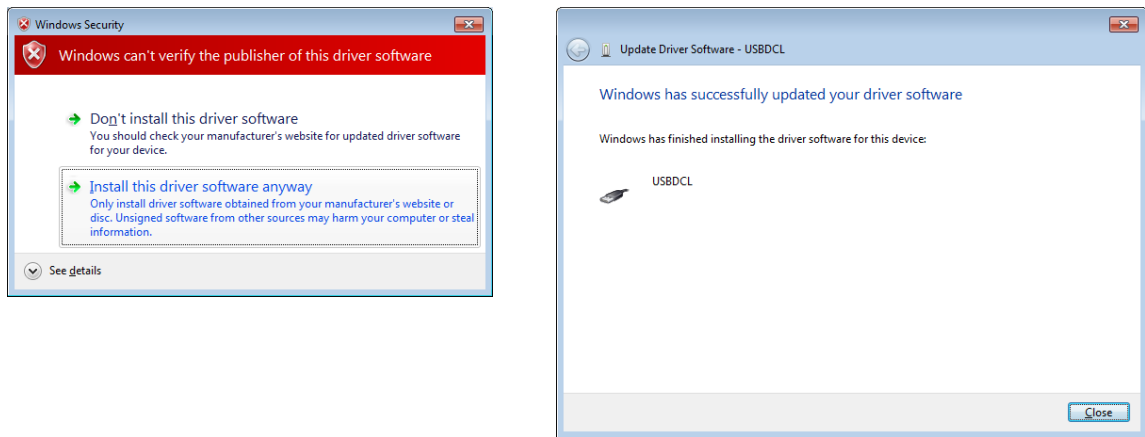


Figure 5.5.

In Windows 8.1 or later the USB device will malfunction after the drivers has been installed. This can be fixed by adding an entry into the Windows registry.

1. Click Start, click Run, type regedit in the Open box, and then click OK .
2. Locate and then click the following subkey in the registry: **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\UsbFlags**
3. Locate the subkey **17810AA00000** or create a new key if doesn't exist. To create a new key on the Edit menu, point to New, and then click Key. Type the name of the new key **17810AA00000**
4. On the Edit menu, point to New, and then click DWORD (32-bit) Value.
5. Type **SkipBOSDescriptorQuery** for the name of the DWORD Value, and then press ENTER.
6. Right-click **SkipBOSDescriptorQuery**, and then click Modify.
7. In the Value data box, type **1**, and then click OK.
8. Exit Registry Editor.
9. Unplug and re-plug the device for the workaround to take effect.

5.5. GRESB debug link

Targets equipped with a SpaceWire core with RMAP support can be debugged through the GRESB debug link using the GRESB Ethernet to SpaceWire bridge. To do so start GRMON with the `-gresb` switch and use the any of the switches below to set the needed parameters.

For further information about the GRESB bridge see the GRESB manual.

Extra options for the GRESB connection:

- `-gresb [<ipnum>]`
Use the GRESB connection and optionally use *ipnum* for the target system IP number. Default is 192.168.0.50.
- `-link <num>`
Use link *linknum* on the bridge. Defaults to 0.
- `-dna <dna>`
The destination node address of the target. Defaults to 0xfe.
- `-sna <sna>`
The SpW node address for the link used on the bridge. Defaults to 32.
- `-dpa <dpa1> [,<dpa2>, . . . ,<dpa12>]`
The destination path address. Comma separated list of addresses.
- `-spa <spa1> [,<spa2>, . . . ,<spa12>]`
The source path address. Comma separated list of addresses.
- `-dkey <key>`
The destination key used by the targets RMAP interface. Defaults to 0.
- `-clkdiv <div>`
Divide the TX bit rate by div. If not specified, the current setting is used.
- `-gresbtimeout <sec>`
Timeout period in seconds for RMAP replies. Defaults is 8.

`-gresbretry <n>`

Number of retries for each timeout. Defaults to 0.

5.5.1. AGGA4 SpaceWire debug link

It is possible to debug the AGGA4 via SpaceWire, using the GRESB Ethernet SpaceWire Bridge, by combining the commandline switches `'-gresb'` and `'-sys agga4'` when starting GRMON. In addition, the following options can also be added: `-link`, `-clkdiv`, `-gresbtimeout` and `-gresbretry`.

The AGGA4 SpaceWire debug link does not use a regular SpaceWire packet protocol, therefore the GRESB must be setup to tunnel all the packets as raw data. To achieve this the GRESB must be configured to use separate routing tables, this setting can only be enabled via the web interface.

The GRESB routing tables for the SpaceWire port and the TCP port that will be used must also be configured. The routing tables can be setup via the web interface or using the software distributed with the GRESB. All the node addresses in the routing table for the SpaceWire port must be configured to forward packets to the TCP port without any header deletion. The routing table for the TCP port must be setup in the same way but to forward the packets from all nodes to the SpaceWire port instead. A Linux bash script and a Windows bat-script is provided with GRMON professional distribution in folder `share/grmon/tools`, that can be used with the GRESB software to setup the routing tables. The scripts must be able to find the GRESB software, so either the PATH environment variable must be setup or execute the scripts from the GRESB software folder.

GRESB separate routing table mode shall be used when connecting to the AGGA4 SpaceWire debug link. This can be configured in the GRESB web interface: "Routing table configuration" -> "Set/view Mode" -> "Set Separate mode".

5.6. User defined debug link

In addition to the supported DSU communication interfaces (Serial, JTAG, ETH and PCI), it is possible for the user to add a custom interface using a loadable module. The custom DSU interface must provide functions to read and write data on the target system's AHB bus.

Extra options for the user defined connection:

`-dback <filename>`

Use the user defined debug link. The debug link should be implemented in a loadable module pointed out by the filename parameter.

`-dbackarg <arg>`

Set a custom argument to be passed to the user defined debug link during start-up.

5.6.1. API

The loadable module must export a pointer variable named `DsuUserBackend` that points to a `struct ioif`, as described below:

```
struct ioif {
    int (*wmem) (unsigned int addr, const unsigned int *data, int len);
    int (*gmem) (unsigned int addr, unsigned int *data, int len);
    int (*open) (char *device, int baudrate, int port);
    int (*close) ();
    int (*setbaud) (int baud, int pp);
    int (*init) (char* arg);
};

struct ioif my_io = {my_wmem, my_gmem, NULL, my_close, NULL, my_init};
struct ioif *DsuUserBackend = &my_io;
```

On the Linux platform, the loadable module should be compiled into a library and loaded into GRMON as follows:

```
> gcc -fPIC -c my_io.c
> gcc -shared my_io.o -o my_io.so
> grmon -dback my_io.so -dbackarg "my argument"
```

On the Windows platform, the loadable module should be compiled into a library and loaded into GRMON as follows:

```
> gcc -c my_io.c
> gcc -shared my_io.o -o my_io.dll
```

```
> grmon -dback my_io.dll -dbackarg "my argument"
```

The members of the struct `ioif` are defined as:

```
int (*wmem) (unsigned int addr, const unsigned int *data, int len);
```

A function that performs one or more 32-bit writes on the AHB bus. The parameters indicate the AHB (start) address, a pointer to the data to be written, and the number of words to be written. The data is in little-endian format (note that the AMBA bus on the target system is big-endian). If the `len` parameter is zero, no data should be written. The return value should be the number of words written.

```
int (*gmem) (unsigned int addr, unsigned int *data, int len);
```

A function that reads one or more 32-bit words from the AHB bus. The parameters indicate the AHB (start) address, a pointer to where the read data should be stored, and the number of words to be read. The returned data should be in little-endian format (note that the AMBA bus on the target system is big-endian). If the `len` parameter is zero, no data should be read. The return value should be the number of words read.

```
int (*open) (char *device, int baudrate, int port);
```

Not used, provided only for backwards compatibility. This function is replaced by the function `init`.

```
int (*close) ();
```

Called when disconnecting.

```
int (*setbaud) (int baud, int pp);
```

Not used, provided only for backwards compatibility.

```
int (*init) (char* arg);
```

Called when initiating a connection to the target system. The parameter `arg` is set using the GRMON start-up switch `-dbackarg <arg>`. This allows to send arbitrary parameters to the DSU interface during start-up.

An example module is provided with the professional version of GRMON located at `<grmon3>/share/grmon/src/dsu_user_backend`.

6. Debug drivers

This section describes GRMON debug commands available through the TCL GRMON shell.

6.1. AMBA AHB trace buffer driver

The **at** command and its subcommands are used to control the AHBTRACE buffer core. It is possible to record AHB transactions without interfering with the processor. With the commands it is possible to set up triggers formed by an address and an address mask indicating what bits in the address that must match to set the trigger off. When the triggering condition is matched the AHBTRACE stops the recording of the AHB bus and the log is available for inspection using the **at** command. The **at delay** command can be used to delay the stop of the trace recording after a triggering match.

Note that this is an stand alone AHB trace buffer it is not to be confused with the DSU AHB trace facility. When a break point is hit the processor will not stop its execution.

The **info sys** command displays the size of the trace buffer in number of lines.

```
ahbtrace0 Frontgrade Gaisler  AMBA Trace Buffer
AHB: FFF40000 - FFF60000
Trace buffer size: 512 lines
```

6.2. Clock gating

The GRCLKGATE debug driver provides an interface to interact with a GRCLKGATE clock gating unit. A command line switch can be specified to automatically reset and enable all clocks, controlled by clock gating units, during GRMON's system initialization.

The GRCLKGATE core is accessed using the command **grcg**, see command description in Appendix B, *Command syntax* for more information.

6.2.1. Switches

```
-cginit [<mask>,<mask>,...]
```

Reset and enable all clock-signals controlled by GRCLKGATE during initialization. If no mask is set then all clock-signals will be enabled. If a mask is specified, then clock-signals that will be enabled depends on the mask. One mask per GRCLKGATE core.

6.3. Debug support drivers

The DSU driver for the LEON processor(s), and the RVDM driver for NOEL-V, are a central part of GRMON. It handles most of the functions regarding application execution, debugging, processor register access, cache access and trace buffer handling. The most common interactions with the DSU/RVDM are explained in Chapter 3, *Operation*. Additional information about the configuration of the target processor and debugging support on the target system can be listed with the command **info sys**.

```
dsu0      Frontgrade Gaisler  LEON4 Debug Support Unit
AHB: D0000000 - E0000000
AHB trace: 64 lines, 32-bit bus
CPU0:    win 8, hwbp 2, itrace 64, V8 mul/div, srmmu, lddel 1, GRFPU-lite
         stack pointer 0x4fffffff0
         icache 2 * 8 kB, 32 B/line lrr
         dcache 2 * 4 kB, 32 B/line lrr
CPU1:    win 8, hwbp 2, itrace 64, V8 mul/div, srmmu, lddel 1, GRFPU-lite
         stack pointer 0x4fffffff0
         icache 2 * 8 kB, 32 B/line lrr
         dcache 2 * 4 kB, 32 B/line lrr

dm0      Frontgrade Gaisler  RISC-V Debug Module
AHB: fe000000 - ff000000
hart0:   DXLEN 64, MXLEN 64, SXLEN 64, UXLEN 64
         ISA A D F I M, Modes M S U
         Stack pointer 0x3fffffff0
         icache 4 * 4 kB, 32 B/line, rnd
         dcache 4 * 4 kB, 32 B/line, rnd
         3 triggers,
         itrace 64 lines
hart1:   DXLEN 64, MXLEN 64, SXLEN 64, UXLEN 64
```

```

ISA A D F I M, Modes M S U
Stack pointer 0x3fffffff0
icache 4 * 4 kB, 32 B/line, rnd
dcache 4 * 4 kB, 32 B/line, rnd
3 triggers,
itrace 64 lines

```

6.3.1. Switches

Below is a list of commandline switches that affects how the DSU/RVDM driver interacts with the hardware.

- nb
When the -nb flag is set, the CPUs will not go into debug mode when a error trap occurs. Instead the OS must handle the trap. (LEON only)
- nswb
When the -nswb flag is set, the CPUs will not go into debug mode when a software breakpoint occur. This option is required when a native software debugger like GDB is running on the target CPU.
- dsudelay <ms>
Set polling period when executing application on the target processor. Normally GRMON will poll the hardware as fast as possible.
- nic
Disable instruction cache
- ndc
Disable data cache
- stack <addr>
Set addr as stack pointer for applications, overriding the auto-detected value.
- mpgsz
Enable support for MMU page sizes larger then 4KiB. Must be supported by hardware. (DSU only)

6.3.2. Commands

The driver for the debug support unit provides the commands listed in Table 6.1.

Table 6.1. DSU commands

ahb	Print AHB transfer entries in the trace buffer
attach	Stop execution and attach GRMON to processor again
at	Print AHB transfer entries in the trace buffer
bp	Add, delete or list breakpoints
bt	Print backtrace
cctrl	Display or set cache control register
cont	Continue execution
cpu	Enable, disable CPU or select current active cpu
dcache	Show, enable or disable data cache
dccfg	Display or set data cache configuration register
detach	Resume execution with GRMON detached from processor
ei	Error injection
ep	Set entry point
float	Display FPU registers
forward	Control I/O forwarding
go	Start execution without any initialization
hist	Print AHB transfer or instruction entries in the trace buffer
icache	Show, enable or disable instruction cache
iccfg	Display or set instruction cache configuration register
inst	Print instruction entries in the trace buffer
leon	Print leon specific registers

mmu	Translate virtual addresses
perf	Measure performance
profile	Enable, disable or show simple profiling
reg	Show or set integer registers.
run	Reset and start execution
stack	Set or show the initial stack-pointer
step	Step one or more instructions
stop	Interrupts current CPU execution
tmode	Select tracing mode between none, processor-only, AHB only or both
va	Translate a virtual address
vmemb	AMBA bus 8-bit virtual memory read access, list a range of addresses
vmemd	AMBA bus 64-bit virtual memory read access, list a range of addresses
vmemh	AMBA bus 16-bit virtual memory read access, list a range of addresses
vmem	AMBA bus 32-bit virtual memory read access, list a range of addresses
vwmemb	AMBA bus 8-bit virtual memory write access
vwmemd	AMBA bus 64-bit virtual memory write access
vwmemh	AMBA bus 16-bit virtual memory write access
vwmems	Write a string to an AMBA bus virtual memory address
wmem	AMBA bus 32-bit virtual memory write access
walk	Translate a virtual address, print translation

6.3.3. Tcl variables

The DSU driver exports one Tcl variable per CPU (`cpuN`), they allow the user to access various registers of any CPU instead of using the standard **reg**, **float** and **cpu** commands. The variables are mostly intended for Tcl scripting. See Section 3.4.13, “Multi-processor support” for more information how the `cpu` variable can be used.

6.4. Ethernet controller

The GRETH debug driver provides commands to configure the GRETH 10/100/1000 Mbit/s Ethernet controller core. The driver also enables the user to read and write Ethernet PHY registers. The `info sys` command displays the core’s configuration settings:

```
greth0    Frontgrade Gaisler GR Ethernet MAC
          AHB Master 2
          APB: C0100100 - C0100200
          IRQ: 12
          edcl ip 192.168.0.201, buffer 2 kbyte
```

If more than one GRETH core exists in the system, it is possible to specify which core the internal commands should operate on. This is achieved by appending a device name parameter to the command. The device name is formatted as `greth#` where the `#` is the GRETH device index. If the device name is omitted, the command will operate on the first device. The device name is listed in the **info sys** information.

The IP address must have the numeric format when setting the EDCL IP address using the **edcl** command, i.e. `edcl 192.168.0.66`. See command description in Appendix B, *Command syntax* and Ethernet debug interface in Section 5.2, “Ethernet debug link” for more information.

6.4.1. Commands

The driver for the greth core provides the commands listed in Table 6.2.

Table 6.2. GRETH commands

edcl	Print or set the EDCL ip
mdio	Show PHY registers

phyaddr	Set the default PHY address
wmdio	Set PHY registers

6.5. GRPWM core

The GRPWM debug driver implements functions to report the available PWM modules and to query the waveform buffer. The **info sys** command will display the available PWM modules.

```
grpwm0  Frontgrade Gaisler  PWM generator
        APB: 80010000 - 80020000
        IRQ: 13
        cnt-pwm: 3
```

The GRPWM core is accessed using the command **grpwm**, see command description in Appendix B, *Command syntax* for more information.

6.6. USB Host Controller

The GRUSBHC host controller consists of two host controller types. GRMON provides a debug driver for each type. The info sys command displays the number of ports and the register setting for the enhanced host controller or the universal host controller:

```
usbhci0 Frontgrade Gaisler  USB Enhanced Host Controller
        AHB Master 4
        APB: C0100300 - C0100400
        IRQ: 6
        2 ports, byte swapped registers
usbhci0 Frontgrade Gaisler  USB Universal Host Controller
        AHB Master 5
        AHB: FFF00200 - FFF00300
        IRQ: 7
        2 ports, byte swapped registers
```

If more than one EHCI or UHCI core exists in the system, it is possible to specify which core the internal commands should operate on. This is achieved by appending a device name parameter to the command. The device name is formatted as *usbhci#*/*usbuhci#* where the # is the device index. If the device name is omitted, the command will operate on the first device. The device name is listed in the **info sys** information.

6.6.1. Switches

-nousbrst
Prevent GRMON from automatically resetting the USB host controller cores.

6.6.2. Commands

The drivers for the USB host controller cores provides the commands listed in Table 6.3.

Table 6.3. GRUSBHC commands

ehci	Control the USB host ECHI core
uhci	Controll the USB host UHCI core

6.7. I²C

The I²C-master debug driver initializes the core's prescaler register for operation in normal mode (100 kb/s). The driver supplies commands that allow read and write transactions on the I²C-bus. I.a. it automatically enables the core when a read or write command is issued.

The I2CMST core is accessed using the command **i2c**, see command description in Appendix B, *Command syntax* for more information.

6.8. I/O Memory Management Unit

The debug driver for GRIOMMU provides commands for configuring the core, reading core status information, diagnostic cache accesses and error injection to the core's internal cache (if implemented). The debug driver also

has support for building, modifying and decoding Access Protection Vectors and page table structures located in system memory.

The GRIOMMU core is accessed using the command **iommu**, see command description in Appendix B, *Command syntax* for more information.

The **info sys** command displays information about available protection modes and cache configuration.

```
iommu0  Frontgrade Gaisler IO Memory Management Unit
        AHB Master 4
        AHB: FF840000 - FF848000
        IRQ: 31
        Device index: 0
        Protection modes: APV and IOMMU
        msts: 9, grps: 8, accsz: 128 bits
        APV cache lines: 32, line size: 16 bytes
        cached area: 0x00000000 - 0x80000000
        IOMMU TLB entries: 32, entry size: 16 bytes
        translation mask: 0xff000000
        Core has multi-bus support
```

6.9. Multi-processor interrupt controller

The debug driver for IRQMP provides commands for forcing interrupts and reading core status information. The debug driver also supports ASMP and other extension provided in the IRQ(A)MP core. The IRQMP and IRQAMP cores are accessed using the command **irq**, see command description in Appendix B, *Command syntax* for more information.

The **info sys** command displays information on the cores memory map. I.a. if extended interrupts are enabled it shows the extended interrupt number.

```
irqmp0  Frontgrade Gaisler Multi-processor Interrupt Ctrl.
        APB: FF904000 - FF908000
        EIRQ: 10
```

6.10. L2-Cache Controller

The debug driver for L2C is accessed using the command **l2cache**, see command description in Appendix B, *Command syntax* for more information. It provides commands for showing status, data and hit-rate. It also provides commands for enabling/disabling options and flushing or invalidating the cache lines.

If the L2C core has been configured with memory protection, then the **l2cache error** subcommand can be used to inject check bit errors and to read out error detection information.

L2-Cache is enabled by default when GRMON resets the system. This behavior can be disabled by giving the **-nl2c** command line option which instead disables the cache. L2-Cache can be enabled/disabled later by the user or by software in either case. If **-ni** is given, then L2-Cache state is not altered when GRMON starts.

When GRMON is started without **-ni** and **-nl2c**, the L2-Cache controller will be configured with EDAC disabled, LRU replacement policy, no locked ways, copy-back replacement policy and not using *HPROT* to determine cacheability. Pending EDAC error injection is also removed.

When connecting without **-ni**, if the L2-Cache is disabled, the L2-Cache contents will be invalidated to make sure that any random power-up values will not affect execution. If the L2-Cache was already enabled, it is assumed that the contents are valid and L2-Cache is flushed to backing memory and then invalidated.

When enabling L2-Cache, the subcommand **l2cache disable flushinvalidate** can be used to atomically invalidate and write back dirty lines. The inverse operation is **l2cache invalidate** followed by **l2cache enable**. For debugging the state of L2-Cache itself, it may be more appropriate to use **l2cache disable** as it does not have any side effects on cache tags.

The **info sys** command displays the cache configuration.

```
l2cache0 Frontgrade Gaisler L2-Cache Controller
```

```

AHB Master 0
AHB: 00000000 - 80000000
AHB: F0000000 - F0400000
AHB: FFE00000 - FFF00000
IRQ: 28
L2C: 4-ways, cachesize: 128 kbytes, mtrr: 16

```

6.10.1. Switches

```

-nl2c
  Disable L2-Cache on start-up.
-nl2csplit
  Disable L2 cache split support.

```

6.10.2. Errata

Workarounds for the issues described in GRLIB TN 0021 [<https://www.gaisler.com/doc/antn/GR-LIB-TN-0021.pdf>] have been implemented when split is enabled.

Issues 1, 2 and 3 are most often not applicable since GRMON uses a single master on the bus and other masters are inactive. If any other bus masters are active (e.g. cores with DMA capabilities, incoming SpaceWire RMAP packets or running TCL scripts while the CPUs are executing) then it is up to the user to avoid these issues.

Issue 4 is resolved by writing `L2CERR.COMP = 0`.

Issue 5 is resolved by setting `L2CACCC.DBPF = 0`, `L2CACCC.I28WF = 1` and trigger a line fetch. This issue can also be triggered after reset by the Plug and play scanning performed by GRMON during startup. It can be avoided by using a fixed target XML description of the system (see Appendix D, *Fixed target configuration file format*). GRMON will automatically use an internal fixed target XML description when connecting to the GR740.

Issues 6 and 7 do not apply to GRMON since no errors are propagated to the host computer.

6.11. Statistics Unit

The debug driver for L4STAT provides commands for reading and configuring the counters available in a L4STAT core. The L4STAT core can be implemented with two APB interfaces. GRMON treats a core with dual interfaces the same way as it would treat a system with multiple instances of L4STAT cores. If several L4STAT APB interfaces are found the `l4stat` command must be followed by an interface index reported by `info sys`. The `info sys` command displays also displays information about the number of counters available and the number of processor cores supported.

```

l4stat0 Frontgrade Gaisler LEON4 Statistics Unit
APB: E4000100 - E4000200
cpus: 2, counters: 4, i/f index: 0

l4stat1 Frontgrade Gaisler LEON4 Statistics Unit
APB: FFA05000 - FFA05100
cpus: 2, counters: 4, i/f index: 1

```

The L4STAT core is accessed using the command `l4stat`, see command description in Appendix B, *Command syntax* for more information.

If the core is connected to the DSU it is possible to count several different AHB events. In addition it is possible to apply filter to the signals connected to the L4STAT (if the DSU supports filter), see command `ahb filter performance` in Appendix B, *Command syntax*.

The `l4stat set` command is used to set up counting for a specific event. All allowed values for the event parameters are listed with `l4stat events`. The number and types of events may vary between systems. Example 6.1 shows how to set counter zero to count data cache misses on processor one and counter one to count instruction cache misses on processor zero.

Example 6.1.

```

grmon3> l4stat 1 events
icmiss - icache miss
itmiss - icache tlb miss

```

```

ichold    - icache hold
ithold    - icache mmu hold
dcmiss    - dcache miss
... more events are listed ...

grmon3> l4stat 1 set 0 1 dcmiss
cnt0: Enabling dcache miss on cpu/AHB 1

grmon3> l4stat 1 set 1 0 icmiss
cnt1: Enabling icache miss on cpu/AHB 0

grmon3> l4stat 1 status
CPU  DESCRIPTION  VALUE
0:  cpu1  dcache miss    0000000000
1:  cpu0  icache miss    0000000000
2:  cpu0  icache miss    0000000000 (disabled)
3:  cpu0  icache miss    0000000000 (disabled)

```

Some of the L4STAT events 0x40-0x7F can be counted either per AHB master or independent of master. The **l4stat** command will only count events generated by the AHB master specified in the **l4stat set** command.

The L4STAT debug driver provides two modes that are used to continuously sample L4STAT counters. The driver will print out the latest read value(s) together with total accumulated amount(s) of events while polling. A poll operation can either be started directly or be deferred until the run command is issued. In both cases, counters should first be configured with the type of event to count. When this is done, one of the two following commands can be issued: **l4stat poll st sp int hold** or **l4stat runpoll st sp int**

The behavior of the first command, **l4stat poll**, depends on the hold argument. If hold is 0 or not specified, the specified counter(s) (st - sp) will be enabled and configured to be cleared on read. These counters will then be polled with an interval of int seconds. After each read, the core will print out the current and accumulated values for all counters. If the hold argument is 1, GRMON will not initialize the counters. Instead the first specified counter (st) will be polled. When counter st is found to be enabled the polling operating will begin. This functionality can be used to, for instance, let software signal when measurements should take place.

Polling ends when at least one of the following is true: User pressed CTRL+C (SIGINT) or counter st becomes disabled. When polling stops, the debug driver will disable the selected counter(s) and also disable the automatic clear feature.

The second command, **l4stat runpoll**, is used to couple the poll operation with the run command. When **l4stat runpoll st sp int** has been issued, counters st - sp will be polled after the run command is given. The interval argument in this case does not specify the poll interval seconds but rather in terms of iterations when GRMON polls the Debug Support Unit to monitor execution. A suitable value for the int argument in this case depends on the speed of the host computer, debug link and target system.

Example 6.2 is a transcript from a GRMON session where a vxWorks image is loaded and statistics are collected while it runs.

Example 6.2.

```

grmon3> l4stat 1 set 0 0 icmiss 0
cnt0: Configuring icache miss on cpu/AHB 0

grmon3> l4stat 1 set 1 0 dcmiss 0
cnt1: Configuring dcache miss on cpu/AHB 0

grmon3> l4stat 1 set 2 0 load 0
cnt2: Configuring load instructions on cpu/AHB 0

grmon3> l4stat 1 set 3 0 store 0
cnt3: Configuring store instructions on cpu/AHB

grmon3> l4stat 1 status
CPU  DESCRIPTION  VALUE
0:  cpu0  icache miss    0000000000 (disabled)
1:  cpu0  dcache miss    0000000000 (disabled)
2:  cpu0  load instructions 0000000000 (disabled)
3:  cpu0  store instructions 0000000000 (disabled)

grmon3> l4stat 1 runpoll 0 3 5000
Setting up callbacks so that polling will be performed during 'run'

grmon3> load vxWorks

```

```

00003000 .text                1.5MB / 1.5MB  [=====] 100%
0018F7A8 .init$00             12B           [=====] 100%
0018F7B4 .init$99             8B            [=====] 100%
0018F7BC .fini$00            12B           [=====] 100%
0018F7C8 .fini$99             8B            [=====] 100%
0018F7E0 .data                177.5kB / 177.5kB [=====] 100%
Total size: 1.72MB (2.03Mbit/s)
Entry point 0x3000
Image vxWorks loaded

grmon3> run
TIME      COUNTER  CURRENT READ  CURRENT RATE  TOTAL READ  TOTAL RATE
5.88      0        1973061       335783        1973061     335783
5.88      1        7174279       1220946       7174279     1220946
5.88      2        22943354      3904587       22943354    3904587
5.88      3        491916        83716         491916      83716
11.16     0         0              0              0            176718
11.16     1        11014132     2082460       18188411    1629056
11.16     2        33072417     6253057       56015771    5017087
11.16     3        15751         2978          507667      45470
... output removed ...
51.35     0         0              0              1973061     38425
51.35     1        12113004     2079486       101754132   1981657
51.35     2        36365101     6242936       306891414   5976697
51.35     3        17273         2965          627067      12212

```

And alternative to coupling polling to the run command is to break execution, issue **detach** and then use the **l4stat poll** command. There are a few items that may be worth considering when using poll and runpoll.

- All counters are not read in the same clock cycle. Depending on the debug link used there may be a significant delay between the read of the first and the last counter.
- Measurements are timed on the host computer and reads experience jitter from several sources.
- A counter may overflow after 2^{32} target clock cycles. The poll period (interval) should take this into account so that counters are read (and thereby cleared) before an overflow can occur.
- Counters are disabled when polling stops
- **l4stat runpoll** is only supported for uninterrupted run. Commands like **bp** and **cont** may disrupt measurements.
- If the L4STAT core has two APB interfaces, initialize it via the interface to which traffic causes the least disturbance to other system bus traffic.

6.12. LEON2 support

A LEON2 system has a fixed set of IP cores and address mapping. GRMON will use an internal plug and play table that describes this configuration. The plug and play table used for LEON2 is fixed, and no automatic detection of present cores is attempted. Only those cores that need to be initialized by GRMON are included in the table, so the listing might not correspond to the actual target.

By default, GRMON will enable the UART receivers and transmitters for the AT697E/F by setting the corresponding bits in the IODIR register to output. This can be disabled by providing the commandline switch `-at697-nouart`, GRMON will then reset the IODIR to inputs on all bits.

6.12.1. Switches

- sys at697
Disable plug and play scanning and configure GRMON for an AT697E system
- sys at697e
Disable plug and play scanning and configure GRMON for an AT697E system
- sys at697f
Disable plug and play scanning and configure GRMON for an AT697F system
- at697-nouart
Disable GPIO alternate UART function. When this is set, GRMON will reset the GPIO dir register bits to input. By default GRMON will setup the GPIO dir register to enable both UARTs for the AT697E/F.
- sys agga4
Disable plug and play scanning and configure GRMON for an AGGA4 system
- agga4-nognss
Disable the built-in support for the GNSS core to make sure that GRMON never makes any accesses to the core. This flag should be used if no clock is provided to the GNSS core.
- sys leon2
Disable plug and play scanning and configure GRMON for a LEON2 system

6.13. On-chip logic analyzer driver

The LOGAN debug driver contains commands to control the LOGAN on-chip logic analyzer core. It allows to set various triggering conditions and to generate VCD waveform files from trace buffer data.

The LOGAN core is accessed using the command **la**, see command description in Appendix B, *Command syntax* for more information.

The LOGAN driver can create a VCD waveform file using the **la dump** command. The file `setup.logan` is used to define which part of the trace buffer belong to which signal. The file is read by the debug driver before a VCD file is generated. An entry in the file consists of a signal name followed by its size in bits separated by white-space. Rows not having these two entries as well as rows beginning with an # are ignored. GRMON will look for the file in the current directory. I.e. either start GRMON from the directory where `setup.logan` is located or use the Tcl command **cd**, in GRMON, to change directory.

Example 6.3.

```
#Name      Size
clk        1
seq        14
edclstate  4
txdstate   5
dataout0   32
dataout1   32
dataout2   32
dataout3   32
writem     1
writel     1
nak        1
lock       1
```

The Example 6.3 has a total of 128 traced bits, divided into twelve signals of various widths. The first signal in the configuration file maps to the most significant bits of the vector with the traced bits. The created VCD file can be opened by waveform viewers such as GTKWave or Dinotrace.

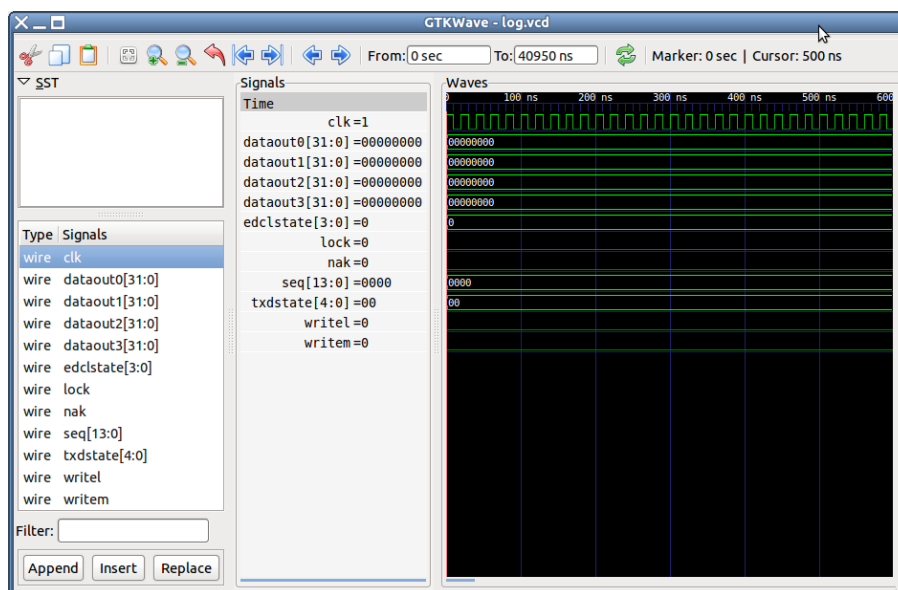


Figure 6.1. GTKWave

6.14. Memory controllers

SRAM/SDRAM/PROM/IO memory controllers. Most of the memory controller debug drivers provides switches for timing, wait state control and sizes. They also probes the memory during GRMON's initialization. In addition they also enables some commands. The **mcfg#** sets the reset value ¹ of the registers. The **info sys** shows the timing and amount of detected memory of each type. Supported cores: MCTRL, SRCTRL, SSRCTRL, FTMCTRL, FTSRCTRL, FTSRCTRL8

¹ The memory register reset value will be written when GRMON's resets the drivers, for example when **run** or **load** is called.

```

mctrl0 European Space Agency LEON2 Memory Controller
AHB: 00000000 - 20000000
AHB: 20000000 - 40000000
AHB: 40000000 - 80000000
APB: 80000000 - 80000100
8-bit prom @ 0x00000000
32-bit sdram: 1 * 64 Mbyte @ 0x40000000
col 9, cas 2, ref 7.8 us

```

PC133 SDRAM Controller . PC133 SDRAM debug drivers provides switches for timing. It also probes the memory during GRMON's initialization. In addition it also enables the **sdcfg1** affects, that sets the reset value ¹ of the register. Supported cores: SDCTRL, FTSDCTRL

DDR memory controller. The DDR memory controller debug drivers provides switches for timing. It also performs the DDR initialization sequence and probes the memory during GRMON's initialization. It does not enable any commands. The **info sys** shows the DDR timing and amount of detected memory. Supported cores: DDRSPA

DDR2 memory controller. The DDR2 memory controller debug driver provides switches for timing. It also performs the DDR2 initialization sequence and probes the memory during GRMON's initialization. In addition it also enables some commands. The **ddr2cfg#** only affect the DDR2SPA, that sets the reset value ¹ of the register. The commands **ddr2skew** and **ddr2delay** can be used to adjust the timing. The **info sys** shows the DDR timing and amount of detected memory Supported cores: DDR2SPA

```

ddr2spa0 Frontgrade Gaisler Single-port DDR2 controller
AHB: 40000000 - 80000000
AHB: FFE00100 - FFE00200
32-bit DDR2 : 1 * 256 MB @ 0x40000000, 8 internal banks
200 MHz, col 10, ref 7.8 us, trfc 135 ns

```

Dual-port AHB(/CPU) On-Chip RAM. The On-Chip RAM debug drivers probes the memory during GRMON's initialization. The scrubber is not enabled. The **info sys** the amount of detected memory. Supported cores: LRAM

```

ilram0 Frontgrade Gaisler Dual-port AHB(/CPU) On-Chip RAM
AHB: 31000000 - 31100000
APB: 8000b000 - 8000b100
IRQ: 63
32-bit static ram: 128 kB @ 0x31000000
dlram0 Frontgrade Gaisler Dual-port AHB(/CPU) On-Chip RAM
AHB: 30000000 - 30100000
APB: 80001000 - 80001100
IRQ: 63
32-bit static ram: 64 kB @ 0x30000000

```

SPI memory controller. The SPI memory controller debug driver is affected by the common memory commands, but provides commands **spim** to perform basic communication with the core. The driver also provides functionality to read the CSD register from SD Card and a command to reinitialize SD Cards. The debug driver has bindings to the SPI memory device layer. These commands are accessed via **spim flash**. Please see Section 3.11.4, "SPI memory device" for more information. Supported cores: SPIMCTRL

6.14.1. Switches

- edac
Enable EDAC operation (FTMCTRL, FTSRCTRL, FTSRCTRL8, LRAM)
- edac8[4/5]
Overrides the auto-probed EDAC area size for 8-bit RAM. Valid values are 4 if the EDAC uses a quarter of the memory, or 5 if the EDAC uses a fifth. (FTMCTRL)
- rsedac
Enable Reed-Solomon EDAC operation (FTMCTRL)
- mcfg1 <val>
Set the reset value for memory configuration register 1 (MCTRL, FTMCTRL, SSRCTRL)
- mcfg2 <valn>
Set the reset value for memory configuration register 2 (MCTRL, FTMCTRL)
- mcfg3 <val>
Set the reset value for memory configuration register 3 (MCTRL, FTMCTRL, SSRCTRL)
- pageb
Enable SDRAM page burst (FTMCTRL)

- normw
 Disables read-modify-write cycles for sub-word writes to 16- bit 32-bit areas with common write strobe (no byte write strobe). (MCTRL, FTMCTRL)

ROM switches:

- romwidth [*8/16/32*]
 Set the rom bit width. Valid values are 8, 16 or 32. (MCTRL, FTMCTRL, SRCTRL, FTSRCTRL, FT-SRCTRL8)
- romrws *<n>*
 Set *n* number of wait-states for rom reads. (MCTRL, FTMCTRL, SSRCTRL)
- romwws *<n>*
 Set *n* number of wait-states for rom writes. (MCTRL, FTMCTRL, SSRCTRL)
- romws *<n>*
 Set *n* number of wait-states for rom reads and writes. (MCTRL, FTMCTRL, SSRCTRL)

SRAM switches:

- nosram
 Disable SRAM and map SDRAM to the whole plug and play bar. (MCTRL, FTMCTRL, SSRCTRL)
- nosram5
 Disable SRAM bank 5 detection. (MCTRL, FTMCTRL)
- ram *<kB>*
 Overrides the auto-probed amount of static ram bank size. Bank size is given in kilobytes. (MCTRL, FTM-CTRL)
- rambanks *<n>*
 Overrides the auto-probed number of populated ram banks. (MCTRL, FTMCTRL)
- ramwidth [*8/16/32*]
 Overrides the auto-probed ram bit width. Valid values are 8, 16 or 32. (MCTRL, FTMCTRL)
- ramrws *<n>*
 Set *n* number of wait-states for ram reads. (MCTRL, FTMCTRL)
- ramwws *<n>*
 Set *n* number of wait-states for ram writes. (MCTRL, FTMCTRL)
- ramws *<n>*
 Set *n* number of wait-states for rom reads and writes. (MCTRL, FTMCTRL)

SDRAM switches:

- cas *<cycles>*
 Programs SDRAM to either 2 or 3 cycles CAS latency and RAS/CAS delay. Default is 2. (MCTRL, FTM-CTRL, SDCTRL, FTSDCTRL)
- ddr2cal
 Run delay calibration routine on start-up before probing memory (see **ddr2delay scan** com-mand).(DDR2SPA) (DDR2SPA)
- nosdram
 Disable SDRAM. (MCTRL, FTMCTRL)
- ref *<us>*
 Set the refresh reload value, default is 7.8us (64ms, 8,192-cycle refresh). (MCTRL, FTMCTRL, SDCTRL, FTSDCTRL)
- regmem
 Enable registered memory. (DDR2SPA)
- trcd *<cycles>*
 Programs SDRAM to either 2 or 3 cycles RAS/CAS delay. Default is 2. (DDRSPA, DDR2SPA)
- trfc *<ns>*
 Programs the SDRAM trfc to the specified timing. (MCTRL, FTMCTRL, DDRSPA, DDR2SPA, SDC-TRL, FTSDCTRL)
- trp3
 Programs the SDRAM trp timing to 3. Default is 2. (MCTRL, FTMCTRL, DDRSPA, DDR2SPA, SDC-TRL, FTSDCTRL)
- twr
 Programs the SDRAM twr to the specified timing. (DDR2SPA)

- sddcs <value>
Enable double chip select mode. (GR740 SDCTRL)
- sddel <value>
Set the SDCLK value. (MCTRL, FTMCTRL)
- sd2tdis
Disable SDRAM 2T signaling. By default 2T is enabled on GR740 during GRMON initialization. (GR740 SDCTRL)
- sdfreq <mhz>
Set SDRAM frequency in MHz. Default is the system frequency (except for GR740 which defaults to 50MHz). (MCTRL, FTMCTRL, SDCTRL, FTSDCTRL)
- gr712rc_hybrid
Enable GR712RC Hybrid mode where the memory is configured to CAS 2 and the GR712RC to CAS 3. This can allow SDRAM to work at higher frequencies in a desktop environment. (FTMCTRL)

6.14.2. Commands

The driver for the Debug support unit provides the commands listed in Table 6.4.

Table 6.4. MEMCTRL commands

ddr2cfg1	Show or set the reset value of the memory register
ddr2cfg2	Show or set the reset value of the memory register
ddr2cfg3	Show or set the reset value of the memory register
ddr2cfg4	Show or set the reset value of the memory register
ddr2cfg5	Show or set the reset value of the memory register
ddr2delay	Change read data input delay.
ddr2skew	Change read skew
mcfg1	Show or set reset value of the memory controller register 1
mcfg2	Show or set reset value of the memory controller register 2
mcfg3	Show or set reset value of the memory controller register 3
sdcfg1	Show or set reset value of SDRAM controller register 1
sddel	Show or set the SDCLK delay
spim	Commands for the SPI memory controller

6.15. Memory scrubber

The MEMSCRUB core is accessed using the command **scrub**, see command description in Appendix B, *Command syntax* for more information. It provides commands for reading the core's status, and performing some basic operations such as clearing memory.

The **info sys** command displays information on the configured burst length of the scrubber.

```
memscrub0 Frontgrade Gaisler AHB Memory Scrubber
          AHB Master 1
          AHB: FFE01000 - FFE01100
          IRQ: 28
          burst length: 32 bytes
```

6.16. MIL-STD-1553B Interface

The **info sys** command displays the enabled parts of the core, and the configured codec clock frequency. The GR1553B core is accessed using the command **mil**, see command description in Appendix B, *Command syntax* for more information.

```
gr1553b0 Frontgrade Gaisler MIL-STD-1553B Interface
          APB: FFA02000 - FFA02100
          IRQ: 26
          features: BC RT BM, codec clock: 20 MHz
```

Device index: 0

Examining data structures. The `mil bcx` and `mil bmx` commands prints the contents of memory interpreted as BC descriptors or BM entries, in human readable form, as seen in Example 6.4.

Example 6.4.

```
grmon3> mil bcx 0x40000080
Address      TType      RTAddr:SA  WC Bus  Tries  SlTime TO  Options Result  vStat BufPtr
=====
0x40000080  BC-RT      05:30     1  B  01:Same  0 14   s  NoRes 1 0000 40000000
0x40000090  RT-BC      05:30     1  B  01:Same  0 14   s  [Not written] 40000040
0x400000a0  BC-RT      05:30     2  B  01:Same  0 14   s  [Not written] 40000000
0x400000b0  RT-BC      05:30     2  B  01:Same  0 14   s  [Not written] 40000040
0x400000c0  BC-RT      05:30     3  B  01:Same  0 14   s  [Not written] 40000000
0x400000d0  RT-BC      05:30     3  B  01:Same  0 14   s  [Not written] 40000040
0x400000e0  BC-RT      05:30     4  B  01:Same  0 14   s  [Not written] 40000000
```

Data transfers. If the GR1553B core is BC capable, you can perform data transfers directly from the GRMON command line. The commands exist in two variants: `mil get` and `mil put` that specify data directly on the command line and through the terminal, and `mil getm` and `mil putm` that sends/receives data to an address in RAM.

In order to perform BC data transfers, you must have a temporary buffer in memory to store descriptors and data, this is set up with the `mil buf` command.

The data transfer commands use the asynchronous scheduling feature of the core, which means that the command can be performed even if a regular BC schedule is running in parallel. The core will perform the transfer while the primary schedule is idle and will not affect the schedule. It can even be run with BC software active in the background, as long as the software does not make use of asynchronous transfer lists.

If the primary schedule blocks the asynchronous transfer for more than two seconds, the transfer will be aborted and an error message is printed. This can happen if the running schedule does not have any slack, or if it is stuck in suspended state or waiting for a sync pulse with no previous slot time left. In this case, you need to stop the ordinary processing (see `mil halt`) and retry the transfer.

Temporary data buffer. Many of the `mil` subcommands need a temporary data buffer in order to do their work. The address of this buffer is set using the `mil buf` command and defaults to the start of RAM. By default the driver will read out the existing contents and write it back after the transfer is done, this can be changed using the `mil bufmode` command.

If the core is on a different bus where the RAM is at another address range, the scratch area address in the core's address space should be given as an additional `coreaddr` argument to the `mil buf` command.

Halting and resuming. The `mil halt` command will stop and disable the RT,BC and BM parts of the core, preventing them from creating further DMA and 1553 bus traffic during debugging. Before this is done, the current enable state is stored, which allows it to later be restored using `mil resume`. The core is halted gracefully and the command will wait for current ongoing transfers to finish.

The state preserved between `mil halt` and `mil resume` are:

- BC schedules' (both primary and async) states and next positions. If schedule is not stopped, the last transfer status is also preserved (as explained below)
- BC IRQ ring position
- RT address, enable status, subaddress table location, mode code control register, event log size and position
- BM enable status, filter settings, ring buffer pointers, time tag setup

State that is not preserved is:

- IRQ set/clear status
- BC schedule time register and current slot time left.
- RT bus words and sync register
- RT and BM timer values
- Descriptors and other memory contents

For the BC, some extra handling is necessary as the last transfer status is not accessible via the register interface. In some cases, the BC must be probed for the last transfer status by running a schedule with conditional suspends and checking which ones are taken. This requires the temporary data buffer to be setup (see `mil buf`).

Loop-back test. The debug driver contains a loop-back test command **mil lbtest** for testing 1553 transmission on both buses between two devices. In this test, one of the devices is configured as RT with a loop-back subaddress 30. The other device is configured as BC, sends and receives back data with increasing transfer size up to the maximum of 32 words.

The **mil lbtest** command needs a 16K RAM scratch area, which is either given as extra argument or selected using the **mil buf** command as described in the previous section.

Before performing the loop-back test, the routine performs a test of the core's internal time base, by reading out the timer value at a time interval, and displays the result. This is to quickly identify if the clock provided to the core has the wrong frequency.

In the RT case, the command first configures the RT to the address given and enables subaddress 30 in loop-back mode with logging. The RT event log is then polled and events arriving are printed out to the console. The command exits after 60 seconds of inactivity.

In the BC case, the command sets up a descriptor list with alternating BC-to-RT and RT-to-BC transfers of increasing size. After running through the list, the received and transmitted data are compared. This is looped twice, for each bus.

6.17. PCI

The debug driver for the PCI cores are mainly useful for PCI host systems. It provides a command that initializes the host. The initialization sets AHB to PCI memory address translation to 1:1, AHB to PCI I/O address translation to 1:1, points BAR1 to 0x40000000 and enables PCI memory space and bus mastering, but it will not configure target bars. To configure the target bars on the pci bus, call **pci conf** after the core has been initialized. Commands for scanning the bus, disabling byte twisting and displaying information are also provided.

The PCI cores are accessed using the command **pci**, see command description in Appendix B, *Command syntax* for more information. Supported cores are GRPCI, GRPCI2 and PCIF.

The PCI commands have been split up into several sub commands in order for the user to have full control over what is modified. The **init** command initializes the host controller, which may not be wanted when the LEON target software has set up the PCI bus. The typical two different use cases are, GRMON configures PCI or GRMON scan PCI to viewing the current configuration. In the former case GRMON can be used to debug PCI hardware and the setup, it enables the user to set up PCI so that the CPU or GRMON can access PCI boards over I/O, Memory and/or Configuration space and the PCI board can do DMA to the 0x40000000 AMBA address. The latter case is often used when debugging LEON PCI software, the developer may for example want to see how Linux has configured PCI but not to alter anything that would require Linux to reboot. Below are command sequences of the two typical use cases on the ML510 board:

```
grmon3> pci init
grmon3> pci conf

PCI devices found:

Bus 0 Slot 1 function: 0 [0x8]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5451 (M5451 PCI AC-Link Controller Audio Device)
IRQ INTA# LINE: 0
BAR 0: 1201 [256B]
BAR 1: 82206000 [4kB]

Bus 0 Slot 2 function: 0 [0x10]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x1533 (M1533/M1535/M1543 PCI to ISA Bridge [Aladdin IV/V/V+])

Bus 0 Slot 3 function: 0 [0x18]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5457 (M5457 AC'97 Modem Controller)
IRQ INTA# LINE: 0
BAR 0: 82205000 [4kB]
BAR 1: 1101 [256B]

Bus 0 Slot 6 function: 0 [0x30] (BRIDGE)
Vendor id: 0x3388 (Hint Corp)
Device id: 0x21 (HB6 Universal PCI-PCI bridge (non-transparent mode))
Primary: 0 Secondary: 1 Subordinate: 1
I/O: BASE: 0x0000f000, LIMIT: 0x0000fff (DISABLED)
```

```
MEMIO: BASE: 0x82800000, LIMIT: 0x830fffff (ENABLED)
MEM:   BASE: 0x80000000, LIMIT: 0x820fffff (ENABLED)

Bus 0 Slot 9 function: 0 [0x48] (BRIDGE)
Vendor id: 0x104c (Texas Instruments)
Device id: 0xac23 (PCI2250 PCI-to-PCI Bridge)
Primary: 0 Secondary: 2 Subordinate: 2
I/O:   BASE: 0x00001000, LIMIT: 0x00001fff (ENABLED)
MEMIO: BASE: 0x82200000, LIMIT: 0x822fffff (ENABLED)
MEM:   BASE: 0x82100000, LIMIT: 0x821fffff (ENABLED)

Bus 0 Slot c function: 0 [0x60]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x7101 (M7101 Power Management Controller [PMU])

Bus 0 Slot f function: 0 [0x78]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTA# LINE: 0
BAR 0: 82204000 [4kB]

Bus 1 Slot 0 function: 0 [0x100]
Vendor id: 0x102b (Matrox Electronics Systems Ltd.)
Device id: 0x525 (MGA G400/G450)
IRQ INTA# LINE: 0
BAR 0: 80000008 [32MB]
BAR 1: 83000000 [16kB]
BAR 2: 82800000 [8MB]
ROM:  82000001 [128kB] (ENABLED)

Bus 2 Slot 2 function: 0 [0x210]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTB# LINE: 0
BAR 0: 82202000 [4kB]

Bus 2 Slot 2 function: 1 [0x211]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTC# LINE: 0
BAR 0: 82201000 [4kB]

Bus 2 Slot 2 function: 2 [0x212]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTD# LINE: 0
BAR 0: 82200000 [4kB]

Bus 2 Slot 2 function: 3 [0x213]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5239 (USB 2.0 Controller)
IRQ INTA# LINE: 0
BAR 0: 82203200 [256B]

Bus 2 Slot 3 function: 0 [0x218]
Vendor id: 0x1186 (D-Link System Inc)
Device id: 0x4000 (DL2000-based Gigabit Ethernet)
IRQ INTA# LINE: 0
BAR 0: 1001 [256B]
BAR 1: 82203000 [512B]
ROM:  82100001 [64kB] (ENABLED)
```

When analyzing the system, the sub commands *info* and *scan* can be called without altering the hardware configuration:

```
grmon3> pci info

GRPCI initiator/target (in system slot):

  Bus master:   yes
  Mem. space en: yes
  Latency timer: 0x0
  Byte twisting: disabled

  MMAP:         0x8
  IOMAP:        0xffff2

  BAR0:         0x00000000
  PAGE0:        0x40000001
  BAR1:         0x40000000
  PAGE1:        0x40000000

grmon3> pci scan
Warning: PCI driver has not been initialized
```

Warning: PCI driver has not been initialized

PCI devices found:

```

Bus 0 Slot 1 function: 0 [0x8]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5451 (M5451 PCI AC-Link Controller Audio Device)
IRQ INTA# LINE: 0
BAR 0: 1201 [256B]
BAR 1: 82206000 [4kB]

Bus 0 Slot 2 function: 0 [0x10]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x1533 (M1533/M1535/M1543 PCI to ISA Bridge [Aladdin IV/V/V+])

Bus 0 Slot 3 function: 0 [0x18]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5457 (M5457 AC'97 Modem Controller)
IRQ INTA# LINE: 0
BAR 0: 82205000 [4kB]
BAR 1: 1101 [256B]

Bus 0 Slot 6 function: 0 [0x30] (BRIDGE)
Vendor id: 0x3388 (Hint Corp)
Device id: 0x21 (HB6 Universal PCI-PCI bridge (non-transparent mode))
Primary: 0 Secondary: 1 Subordinate: 1
I/O: BASE: 0x0000f000, LIMIT: 0x0000ffff (DISABLED)
MEMIO: BASE: 0x82800000, LIMIT: 0x830fffff (ENABLED)
MEM: BASE: 0x80000000, LIMIT: 0x820fffff (ENABLED)

Bus 0 Slot 9 function: 0 [0x48] (BRIDGE)
Vendor id: 0x104c (Texas Instruments)
Device id: 0xac23 (PCI2250 PCI-to-PCI Bridge)
Primary: 0 Secondary: 2 Subordinate: 2
I/O: BASE: 0x00001000, LIMIT: 0x00001fff (ENABLED)
MEMIO: BASE: 0x82200000, LIMIT: 0x822fffff (ENABLED)
MEM: BASE: 0x82100000, LIMIT: 0x821fffff (ENABLED)

Bus 0 Slot c function: 0 [0x60]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x7101 (M7101 Power Management Controller [PMU])

Bus 0 Slot f function: 0 [0x78]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTA# LINE: 0
BAR 0: 82204000 [4kB]

Bus 1 Slot 0 function: 0 [0x100]
Vendor id: 0x102b (Matrox Electronics Systems Ltd.)
Device id: 0x525 (MGA G400/G450)
IRQ INTA# LINE: 0
BAR 0: 80000008 [32MB]
BAR 1: 83000000 [16kB]
BAR 2: 82800000 [8MB]
ROM: 82000001 [128kB] (ENABLED)

Bus 2 Slot 2 function: 0 [0x210]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTB# LINE: 0
BAR 0: 82202000 [4kB]

Bus 2 Slot 2 function: 1 [0x211]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTC# LINE: 0
BAR 0: 82201000 [4kB]

Bus 2 Slot 2 function: 2 [0x212]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5237 (USB 1.1 Controller)
IRQ INTD# LINE: 0
BAR 0: 82200000 [4kB]

Bus 2 Slot 2 function: 3 [0x213]
Vendor id: 0x10b9 (ULi Electronics Inc.)
Device id: 0x5239 (USB 2.0 Controller)
IRQ INTA# LINE: 0
BAR 0: 82203200 [256B]

Bus 2 Slot 3 function: 0 [0x218]
Vendor id: 0x1186 (D-Link System Inc)
Device id: 0x4000 (DL2000-based Gigabit Ethernet)
IRQ INTA# LINE: 0

```

```
BAR 0: 1001 [256B]
BAR 1: 82203000 [512B]
ROM: 82100001 [64kB] (ENABLED)
```

```
grmon3> pci bus reg
```

```
grmon3> info sys pdev0 pdev5 pdev10
pdev0      Bus 00 Slot 01 Func 00 [0:1:0]
           vendor: 0x10b9 ULi Electronics Inc.
           device: 0x5451 M5451 PCI AC-Link Controller Audio Device
           class: 040100 (MULTIMEDIA)
           BAR1: 00001200 - 00001300 I/O-32 [256B]
           BAR2: 82206000 - 82207000 MEMIO [4kB]
           IRQ INTA# -> IRQX
pdev5      Bus 00 Slot 09 Func 00 [0:9:0]
           vendor: 0x104c Texas Instruments
           device: 0xac23 PCI2250 PCI-to-PCI Bridge
           class: 060400 (PCI-PCI BRIDGE)
           Primary: 0 Secondary: 2 Subordinate: 2
           I/O Window: 00001000 - 00002000
           MEMIO Window: 82200000 - 82300000
           MEM Window: 82100000 - 82200000
pdev10     Bus 02 Slot 03 Func 00 [2:3:0]
           vendor: 0x1186 D-Link System Inc
           device: 0x4000 DL2000-based Gigabit Ethernet
           class: 020000 (ETHERNET)
           subvendor: 0x1186, subdevice: 0x4004
           BAR1: 00001000 - 00001100 I/O-32 [256B]
           BAR2: 82203000 - 82203200 MEMIO [512B]
           ROM: 82100000 - 82110000 MEM [64kB]
           IRQ INTA# -> IRQW
```

A configured PCI system can be registered into the GRMON device handling system similar to the on-chip AMBA bus devices, controlled using the **pci bus** commands. GRMON will hold a copy of the PCI configuration in memory until a new **pci conf**, **pci bus unreg** or **pci scan** is issued. The user is responsible for updating GRMON's PCI configuration if the configuration is updated in hardware. The devices can be inspected from **info sys** and Tcl variables making read and writing PCI devices configuration space easier. The Tcl variables are named in a similar fashion to AMBA devices, for example **puts \$pdev0::status** prints the STATUS register of PCI device0. See **pci bus** reference description and Appendix C, *Tcl API*.

Only the **pci info** command has any effect on non-host systems.

Also note that the **pci conf** command can fail to configure all found devices if the PCI address space addressable by the PCI Host controller is smaller than the amount of memory needed by the devices.

The **pci scan** command may fail if the PCI buses (PCI-PCI bridges) haven't been enumerated correctly in a multi-bus PCI system.

After registering the PCI bus into GRMON's device handling system commands may access device information and Tcl may access variables (PCI configuration space registers). Accessing bad PCI regions may lead to target deadlock where the debug-link may disconnect/hang. It is the user's responsibility to make sure that GRMON's PCI information is correct. The PCI bus may need to be re-scanned/unregistered when changes to the PCI configuration has been made by the target OS running on the LEON.

6.17.1. PCI Trace

The **pci trace** commands are supported by the cores PCITRACE, GRPCI2 and GRPCI2_TB. The commands can be used to control the trace and viewing trace data. With the commands it is possible to set up trigger conditions that must match to set the trigger off. When the triggering condition is matched the AHBTRACE stops the recording of the PCI bus and the log is available for inspection using the **pci trace log** command. The **pci trace tdelay** command can be used to delay the stop of the trace recording after a triggering match.

The **info sys** command displays the size of the trace buffer in number of lines.

```
pcitrace0 Frontgrade Gaisler 32-bit PCI Trace Buffer
          APB: C0101000 - C0200000
          Trace buffer size: 128 lines
pci0     Frontgrade Gaisler GRPCI2 PCI/AHB bridge
          AHB Master 5
          AHB: C0000000 - D0000000
```



```

AHB: FFF00000 - FFF40000
APB: 80000600 - 80000700
IRQ: 6
Trace buffer size: 1024 lines
pcitrace1 Frontgrade Gaisler GRPCI2 Trace buffer
APB: 80040000 - 80080000
Trace buffer size: 1024 lines

```

6.18. GR716B Real-Time Accelerator

The GR716B Real-Time Accelerator (RTA) consist of a small sub-system that includes a LEON3 core. The RTAs must have their clocks enabled in the clock gating unit before they can be used. When connecting to a system where they have been enabled, GRMON will reset the RTA cores and detach them (i.e. put them in a power-down free running mode). GRMON can take control of the cores using the command **attach**.

When attached to GRMON the RTA cores can be controlled like normal multi-core system CPUs. See Section 6.3.2, “Commands” for a list of cpu related commands can also be used to display information and control the execution.

```

grmon3> cpu
cpu 0: enabled active
cpu 1: detached
cpu 2: detached

grmon3> attach cpu2
CPU 2: Power down mode

grmon3> cpu
cpu 0: enabled active
cpu 1: detached
cpu 2: enabled

grmon3> reg cpu2
      INS      LOCALS      OUTS      GLOBALS
0: 00000000  00000000  00000000  00000000
1: 00000000  00000000  00000000  00000000
2: 00000000  00000000  00000000  00000000
3: 00000000  00000000  00000000  00000000
4: 00000000  00000000  00000000  00000000
5: 00000000  00000000  00000000  00000000
6: 00000000  00000000  70003ff0  00000000
7: 00000000  00000000  00000000  00000000

psr: f30000e0  wim: 00000002  tbr: 70000000  y: 00000000

pc: 70000000  unimp
npc: 70000004  unimp

```

Some commands, e.g. **bp** or **load**, will only associated it's data with the main cpu by default. To associate them with an RTA core it must be specified when issuing the command.

```

grmon3> load /opt/sparc-bcc-2.3.0-gcc/src/libdrv/examples/rta/main_cpu.elf
31000000 .text          16.9kB / 16.9kB  [=====>] 100%
30000000 .rodata        704B           [=====>] 100%
300002c0 .data           352B           [=====>] 100%
Total size: 17.91kB (575.25kbit/s)
Entry point 0x31000038
Image /opt/sparc-bcc-2.3.0-gcc/src/libdrv/examples/rta/main_cpu.elf loaded

grmon3> load /opt/sparc-bcc-2.3.0-gcc/src/libdrv/examples/rta/rtal.elf cpu2
71000000 .text          12.8kB / 12.8kB  [=====>] 100%
70000000 .rodata        464B           [=====>] 100%
700001d0 .data           352B           [=====>] 100%
Total size: 13.55kB (563.33kbit/s)
Entry point 0x71000038
Image /opt/sparc-bcc-2.3.0-gcc/src/libdrv/examples/rta/rtal.elf loaded

grmon3> bp main
Software breakpoint 1 at <main>

grmon3> bp main cpu2
Software breakpoint 2 at <main>

grmon3> bp
NUM  ADDRESS      MASK      TYPE      EN CPU      SYMBOL FILE
  1  : 0x310003c8      (soft)    Y    0      main+0
  2  : 0x710003dc      (soft)    Y    2      main+0

```

It is possible to run and debug an application on a RTA only. Easiest way to do this is to connect to a RTA subsystem (see startup switches below), but it is also possible when connected to the whole system. To achieve it the RTA core must be attached and the main cpu must be disabled so GRMON will keep it in debug mode. The command **run** cannot be used to start the RTA core, since it will put the RTA core in power-down before starting the core. Instead the **reset** should be called, followed by a command to wake up the RTA using the interrupt controller and then calling **go** to start it. The RTA core can be woken up by using the **irq boot** command or by writing one of the the IRQMP core registers called mpstat or boot.

```
grmon3> attach cpul
CPU 1: Power down mode

grmon3> cpu active 1

grmon3> cpu disable 0

grmon3> load /opt/sparc-bcc-2.3.0-gcc/src/libdrv/examples/rta/rta0.elf cpul
61000000 .text          12.8kB / 12.8kB [=====] 100%
60000000 .rodata        464B          [=====] 100%
600001d0 .data          352B          [=====] 100%
Total size: 13.55kB (563.33kbit/s)
Entry point 0x61000038
Image /opt/sparc-bcc-2.3.0-gcc/src/libdrv/examples/rta/rta0.elf loaded

grmon3> bp main cpul
Software breakpoint 1 at <main>

grmon3> reset

grmon3> irq boot irqmpl

grmon3> go
CPU 1: Breakpoint 1 hit
0x610003dc: f83ba030 std %i4, [%sp + 0x30] <main+0>
```

6.18.1. Switches

Below is a list of commandline switches that affects how the RTA driver interacts with the hardware.

```
-sys gr716b-rta0
-sys gr716b-rta1
```

Connect to the RTA0 or RTA1 sub-system only. GRMON will only discover and initialize the RTA sub-system core. The core will be attached to GRMON during startup and it will not enter power-down mode, (i.e. it will behave like a normal single core system). Accesses to whole system can still be made using the **mem** or **wmem**.

```
-gr716b-norta
```

Disable GRMON support for GR716B RTA subsystem and analog parts. The cores will be detached from GRMON during startup and nothing will be initialized. The intention of this option is to create an environment where the cores run in the background without interference from GRMON. The command **system** can be used to attach and initialize the cores.

6.19. SPI

The SPICTRL debug driver provides commands to configure the SPI controller core. The driver also enables the user to perform simple data transfers. The **info sys** command displays the core's FIFO depth and the number of available slave select signals.

```
spi0      Frontgrade Gaisler SPI Controller
APB: C0100000 - C0100100
IRQ: 23
FIFO depth: 8, 2 slave select signals
Maximum word length: 32 bits
Supports automated transfers
Supports automatic slave select
Controller index for use in GRMON: 0
```

The SPICTRL core is accessed using the command **spi**, see command description in Appendix B, *Command syntax* for more information.

The debug driver has bindings to the SPI memory device layer. These commands are accessed via **spi flash**. Please see Section 3.11.4, "SPI memory device" for more information.

For information about the SPI memory controller (SPIMCTRL), see Section 6.14, “Memory controllers”.

6.20. SpaceWire router

The SPWROUTER core is accessed using the command **spwrtr**, see command description in Appendix B, *Command syntax* for more information. It provides commands to display the core’s registers. The command can also be used to display or setup the routing table.

The **info reg** command only displays a subset of all the registers available. Add **-all** to the **info reg** command to print all registers, or specify one or more register to print a subset. Add **-l** to **info reg** to list all the register names.

```
grmon3> info reg -all -l spwrtr0
GRSPW Router
0xff880004 rtpmap_1          Port 1 routing table map
0xff880008 rtpmap_2          Port 2 routing table map
0xff88000c rtpmap_3          Port 3 routing table map
...

grmon3> info reg spwrtr0::pctrl_2 spwrtr0::rtpmap_2 spwrtr0::rtpmap_64
GRSPW Router
0xff880808 Port 2 control          0x1300002c
GRSPW Router
0xff880008 Port 2 routing table map 0x00000021
GRSPW Router
0xff880100 Logical addr. 64 routing table map 0x00001c38
```

In addition, all registers and register fields are available as variables, see Tcl API more information.

The **info sys** command displays how many ports are implemented in the router.

```
spwrtr0 Frontgrade Gaisler GRSPW Router
AHB: FF880000 - FF881000
Instance id: 67
SpW ports: 8 AMBA ports: 4 FIFO ports: 0
```

6.21. SVGA frame buffer

The SVGACTRL debug driver implements functions to report the available video clocks in the SVGA frame buffer, and to display screen patters for testing. The **info sys** command will display the available video clocks.

```
svga0 Frontgrade Gaisler SVGA frame buffer
AHB Master 2
APB: C0800000 - C0800100
clk0: 25.00 MHz clk1: 25.00 MHz clk2: 40.00 MHz clk3: 65.00 MHz
```

The SVGACTRL core is accessed using the command **svga**, see command description in Appendix B, *Command syntax* for more information.

The **svga draw test_screen** command will show a simple grid in the resolution specified via the format selection. The color depth can be either 16 or 32 bits.

The **svga draw file** command will determine the resolution of the specified picture and select an appropriate format (resolution and refresh rate) based on the video clocks available to the core. The required file format is ASCII PPM which must have a suitable amount of pixels. For instance, to draw a screen with resolution 640x480, a PPM file which is 640 pixels wide and 480 pixels high must be used. ASCII PPM files can be created with, for instance, the GNU Image Manipulation Program (The GIMP).

The **svga custom period horizontal-active-video horizontal-front-porch horizontal-sync horizontal-back-porch vertical-active-video vertical-front-porch vertical-sync vertical-back-porch** command can be used to specify a custom format. The custom format will have precedence when using the **svga draw** command.

7. Support

For support contact the support team at support@gaisler.com.

When contacting support, please identify yourself in full, including company affiliation and site name and address. Please identify exactly what product that is used, specifying if it is an IP core (with full name of the library distribution archive file), component, software version, compiler version, operating system version, debug tool version, simulator tool version, board version, etc.

The support service is only for paying customers with a support contract.

Appendix A. Command index

This section lists all documented commands available in GRMON4.

Table A.1. GRMON command overview

Command Name	Description
about	Show information about GRMON
ahb	Print AHB transfer entries in the trace buffer
amem	Asynchronous bus read
attach	Stop execution and attach GRMON to processor again
at	Print AHB transfer entries in the trace buffer
batch	Execute batch script
bdump	Dump memory to a file
bload	Load a binary file
bp	Add, delete or list breakpoints
bt	Print backtrace
cctrl	Display or set cache control register
cont	Continue execution
cpu	Enable, disable CPU or select current active cpu
dcache	Show, enable or disable data cache
dccfg	Display or set data cache configuration register
dcom	Print or clear debug link statistics
ddr2cfg1	Show or set the reset value of the memory register
ddr2cfg2	Show or set the reset value of the memory register
ddr2cfg3	Show or set the reset value of the memory register
ddr2cfg4	Show or set the reset value of the memory register
ddr2cfg5	Show or set the reset value of the memory register
ddr2delay	Change read data input delay.
ddr2skew	Change read skew
detach	Resume execution with GRMON detached from processor
disassemble	Disassemble memory
dtb	Setup a DTB to be uploaded or print filenames of DTB files
dump	Dump memory to a file
edcl	Print or set the EDCL ip
eeload	Load a file into an EEPROM
ehci	Control the USB host ECHI core
ei	Error injection
ep	Set entry point
execsh	Run commands in the execution shell
exit	Exit GRMON
flash	Write, erase or show information about the flash
float	Display FPU registers
forward	Control I/O forwarding

Command Name	Description
fpgaload	Upload bitstream to GR740-MINI board FPGA
gdb	Control the built-in GDB remote server
go	Start execution without any initialization
gr1553b	MIL-STD-1553B Interface commands
grcg	Control clockgating
grpwm	Control the GRPWM core
grtmx	Control GRTM devices
gui	Control the graphical user interface
help	Print all commands or detailed help for a specific command
hist	Print AHB transfer or instruction entries in the trace buffer
i2c	Commands for the I2C masters
icache	Show, enable or disable instruction cache
iccfg	Display or set instruction cache configuration register
info	Show information
inst	Print instruction entries in the trace buffer
iommu	Control IO memory management unit
irq	Force interrupts or read IRQ(A)MP status information
l2cache	L2 cache control
l3stat	Control Leon3 statistics unit
l4stat	Control Leon4 statistics unit
la	Control the LOGAN core
leon	Print leon specific registers
load	Load a file or print filenames of uploaded files
mcfg1	Show or set reset value of the memory controller register 1
mcfg2	Show or set reset value of the memory controller register 2
mcfg3	Show or set reset value of the memory controller register 3
mdio	Show PHY registers
memb	AMBA bus 8-bit memory read access, list a range of addresses
memd	AMBA bus 64-bit memory read access, list a range of addresses
memh	AMBA bus 16-bit memory read access, list a range of addresses
mem	AMBA bus 32-bit memory read access, list a range of addresses
mil	MIL-STD-1553B Interface commands
mmu	Translate virtual addresses
nolog	Suppress stdout of a command
pci	Control the PCI bus master
perf	Measure performance
phyaddr	Set the default PHY address
profile	Enable, disable or show simple profiling
quit	Quit the GRMON console
reg	Show or set integer registers.
reset	Reset drivers

Command Name	Description
rtg4fddr	Print initialization sequence
rtg4serdes	Print initialization sequence
run	Reset and start execution
rviommu	Control RISC-V IOMMU
scrub	Control memory scrubber
sdefg1	Show or set reset value of SDRAM controller register 1
sddel	Show or set the SDCLK delay
sf2mddr	Print initialization sequence
sf2serdes	Print initialization sequence
shell	Execute shell process
silent	Suppress stdout of a command
spim	Commands for the SPI memory controller
spi	Commands for the SPI controller
spwrtr	Spacewire router information
stack	Set or show the initial stack-pointer
step	Step one or more instructions
stop	Interrupts current CPU execution
svga	Commands for the SVGA controller
symbols	Load, print or lookup symbols
system	Attach or detach devices
thread	Show OS-threads information or backtrace
timer	Show information about the timer devices
tmode	Select tracing mode between none, processor-only, AHB only or both
tps	Control the TPS service
uhci	Control the USB host UHCI core
usrsh	Run commands in threaded user shell
va	Translate a virtual address
verify	Verify that a file has been uploaded correctly
vmemb	AMBA bus 8-bit virtual memory read access, list a range of addresses
vmemd	AMBA bus 64-bit virtual memory read access, list a range of addresses
vmemh	AMBA bus 16-bit virtual memory read access, list a range of addresses
vmem	AMBA bus 32-bit virtual memory read access, list a range of addresses
vwmemb	AMBA bus 8-bit virtual memory write access
vwmemd	AMBA bus 64-bit virtual memory write access
vwmemh	AMBA bus 16-bit virtual memory write access
vwmems	Write a string to an AMBA bus virtual memory address
vwmem	AMBA bus 32-bit virtual memory write access
walk	Translate a virtual address, print translation
wash	Clear or set memory areas
wmdio	Set PHY registers
wmemb	AMBA bus 8-bit memory write access

Command Name	Description
wnemd	AMBA bus 64-bit memory write access
wmemh	AMBA bus 16-bit memory write access
wmems	Write a string to an AMBA bus memory address
wmem	AMBA bus 32-bit memory write access

Appendix B. Command syntax

This section lists the syntax of all documented commands available in GRMON4

1. about - syntax

NAME

about - Show information about GRMON

SYNOPSIS

about

DESCRIPTION

about

Show information about GRMON

2. ahb - syntax

NAME

ahb - Print AHB transfer entries in the trace buffer

SYNOPSIS

ahb *?length?*

ahb subcommand *?args...?*

DESCRIPTION

ahb *?length?*

Print the AHB trace buffer. The *?length?* entries will be printed, default is 10.

ahb break *boolean*

Enable or disable if the AHB trace buffer should break the CPU into debug mode. If disabled it will freeze the buffer and the CPU will continue to execute. Default value of the boolean is true.

ahb force *?boolean?*

Enable or disable the AHB trace buffer even when the processor is in debug mode. Default value of the boolean is true.

ahb performance *?boolean?*

Enable or disable the filter on the signals connected to the performance counters, see “LEON3 Statistics Unit (L3STAT)” and “LEON4 Statistics Unit (L4STAT)”. Only available for DSU3 version 2 and above, and DSU4.

ahb timer *?boolean?*

Enable the timetag counter when in debug mode. Default value of the boolean is true. Only available for DSU3 version 2 and above, and DSU4.

ahb delay *cnt*

If *cnt* is non-zero, the CPU will enter debug-mode after delay trace entries after an AHB watchpoint was hit.

ahb filter reads *?boolean?*

ahb filter writes *?boolean?*

ahb filter addresses *?boolean? ?address mask?*

Enable or disable filtering options if supported by the DSU core. When enabling the addresses filter, the second AHB breakpoint register will be used to define the range of the filter. Default value of the boolean is true. If left out, then the address and mask will be ignored. They can also be set with the command **ahb filter range**. (Not available in all implementations)

ahb filter range *address mask*

Set the base *address* and *mask* that the AHB trace buffer will include if the address filtering is enabled. (Only available in some DSU4 implementations).

ahb filter bwmask *mask*

ahb filter dwmask *mask*

Set which AHB bus/data watchpoints that the filter will affect.

ahb filter mmask *mask*

ahb filter smask *mask*

Set which AHB masters or slaves connected to the bus to exclude. (Only available in some DSU4 implementations)

ahb status

Print AHB trace buffer settings.

RETURN VALUE

Upon successful completion, **ahb** returns a list of trace buffer entries. Each entry is a sublist on the format format: {AHB *time addr data rw trans size master lock resp bp*}. The data field is a sublist of 1,2 or 4 words with MSB first, depending on the size of AMBA bus. Detailed description about the different fields can be found in the DSU core documentation in document grip.pdf. [<http://gaisler.com/products/grlib/grip.pdf>]

The other subcommands have no return value.

EXAMPLE

Print 10 rows

```
grmon3> ahb
TIME      ADDRESS D[127:96] D[95:64] D[63:32] D[31:0]  TYPE ...
266718 FF900004 00000084 00000084 00000084 00000084 read ...
266727 FF900000 0000000D 0000000D 0000000D 0000000D write ...
266760 000085C0 C2042054 80A06000 02800003 01000000 read ...
266781 000085D0 C2260000 81C7E008 91E80008 9DE3BF98 read ...
266812 0000B440 00000000 00000000 00000000 00000000 read ...
266833 0000B450 00000000 00000000 00000000 00000000 read ...
266899 00002640 02800005 01000000 C216600C 82106040 read ...
266920 00002650 C236600C 40001CBD 90100011 1080062E read ...
266986 00000800 91D02000 01000000 01000000 01000000 read ...
267007 00000810 91D02000 01000000 01000000 01000000 read ...
```

TCL returns:

```
{AHB 266718 0xFF900004 {0x00000084 0x00000084 0x00000084 0x00000084} R 0 2 2
0 0 0 0} {AHB 266727 0xFF900000 {0x0000000D 0x0000000D 0x0000000D 0x0000000D}
W 0 2 2 0 0 0 0} {AHB 266760 0x000085C0 {0xC2042054 0x80A06000 0x02800003
0x01000000} R 0 2 4 1 0 0 0} {AHB 266781 0x000085D0 ...
```

Print 2 rows

```
grmon3> ahb 2
TIME      ADDRESS D[127:96] D[95:64] D[63:32] D[31:0]  TYPE ...
266986 00000800 91D02000 01000000 01000000 01000000 read ...
267007 00000810 91D02000 01000000 01000000 01000000 read ...
```

TCL returns:

```
{AHB 266986 0x00000800 {0x91D02000 0x01000000 0x01000000 0x01000000} R 0 2 4
1 0 0 0} {AHB 267007 0x00000810 {0x91D02000 0x01000000 0x01000000 0x01000000}
R 0 3 4 1 0 0 0}
```

SEE ALSO

Section 3.4.9, “Using the trace buffer”

tmode

3. amem - syntax

NAME

amem - Asynchronous bus read

SYNOPSIS

amem

amem *list*

amem *subcommand ?arg?*

DESCRIPTION

The **amem** command is used to schedule bus read transfers for later retrieval of the result. Each transfer is associated with a handle that has to be created before starting a transfer. Multiple concurrent transfers are supported by using separate handles per transfer.

amem

amem list

List all amem handles and their states. An amem state is one of IDLE, RUN or DONE.

amem add *name*

Create a new amem handle named *name*. The name is used as an identifier for the handle when using other **amem** commands.

amem delete *name*

Delete the amem handle named *name*.

amem eval *name address length*

Schedule a bus read access for the handle *name* to read *length* bytes, starting at *address*. If a transfer is already in progress, then the command will fail with the error code set to EBUSY.

amem wait *name*

Wait for an access to finish. The command returns when handle *name* is no longer in the RUN state.

amem result *name*

Return the result of a previous read access if finished, or raise an error if not finished.

amem prio *name ?value?*

Display or set debug link priority for a handle. 0 is the highest priority and 4 is the lowest.

amem state *name*

Display and return the current state of a handle.

RETURN VALUE

amem list returns a list of amem handle entries. Each entry is a sublist of the format: {*name state*}.

amem result returns the read data.

amem prio returns the priority.

amem state returns one of the strings IDLE, RUN or DONE.

EXAMPLE

Create a handle named `myhandle` and schedule a read of 1 MiB from address 0 in the background.

```
grmon3> amem add myhandle
Added amem handle: myhandle

grmon3> amem eval myhandle 0 0x100000
```

```
grmon3> set myresult [amem result myhandle]
```

List handles

```
grmon3> amem list
```

```
grmon3> amem list
```

NAME	STATE	ADDRESS	LENGTH	PRIO	NREQ	BYTES	ERRORS
myhandle	IDLE	-	-	4	1	1048576	0
test0	DONE	0x00000004	0x00000064	4	1	100	0

SEE ALSO

mem

Section 3.4.7, “Displaying memory contents”

4. attach - syntax

attach - Stop execution and attach GRMON to processor again

SYNOPSIS

attach

DESCRIPTION

attach

This command will stop the execution on all CPUs that was started by the command **detach** and attach GRMON again.

RETURN VALUE

Command **attach** has no return value.

5. at - syntax

NAME

at - Print AHB transfer entries in the trace buffer

SYNOPSIS

at *?length?*

at subcommand *?args...?*

DESCRIPTION

at *?length? ?devname?*

Print the AHB trace buffer. The *?length?* entries will be printed, default is 10.

at bp1 *?options? ?address mask? ?devname?*

at bp2 *?options? ?address mask? ?devname?*

Sets AHB trace buffer breakpoint to address and mask. The AHB trace buffer will stop recording when triggered. Available options are `-read` or `-write`.

at bsel *?bus? ?devname?*

Selects bus to trace (not available in all implementations)

at delay *?cnt? ?devname?*

Delay the stops the trace buffer recording after match.

at disable *?devname?*

Stops the trace buffer recording

at enable *?devname?*

Arms the trace buffer and starts recording.

at filter reads *?boolean? ?devname?*

at filter writes *?boolean? ?devname?*

at filter addresses *?boolean? ?address mask? ?devname?*

Enable or disable filtering options if supported by the core. When enabling the addresses filter, the second AHB breakpoint register will be used to define the range of the filter. Default value of the boolean is true. If left out, then the address and mask will be ignored. They can also be set with the command **at filter range**.

at filter range *?address mask? ?devname?*

Set the base *address* and *mask* that the AHB trace buffer will include if the address filtering is enabled.

at filter mmask *mask ?devname?*

at filter smask *mask ?devname?*

Set which AHB masters or slaves connected to the bus to exclude. (Only available in some DSU4 implementations)

at log *?devname?*

Print the whole AHB trace buffer.

at status *?devname?*

Print AHB trace buffer settings.

RETURN VALUE

Upon successful completion, **at** returns a list of trace buffer entries, on the same format as the command **ahb**. Each entry is a sublist on the format format: `{AHB time addr data rw trans size master lock resp irq bp}`. The data field is a sublist of 1,2 or 4 words with MSB first, depending on the size of AMBA bus. Detailed description about the different fields can be found in the DSU core documentation in document grip.pdf. [<http://gaisler.com/products/grlib/grip.pdf>]

The other sub commands have no return value.

EXAMPLE

Print 10 rows

```
grmon3> at
TIME    ADDRESS  D[127:96] D[95:64] D[63:32] D[31:0]  TYPE    ...
266718  FF900004  00000084 00000084 00000084 00000084 read    ...
266727  FF900000  0000000D 0000000D 0000000D 0000000D write   ...
266760  000085C0  C2042054 80A06000 02800003 01000000 read    ...
266781  000085D0  C2260000 81C7E008 91E80008 9DE3BF98 read    ...
266812  0000B440  00000000 00000000 00000000 00000000 read    ...
266833  0000B450  00000000 00000000 00000000 00000000 read    ...
266899  00002640  02800005 01000000 C216600C 82106040 read    ...
266920  00002650  C236600C 40001CBD 90100011 1080062E read    ...
266986  00000800  91D02000 01000000 01000000 01000000 read    ...
267007  00000810  91D02000 01000000 01000000 01000000 read    ...
```

TCL returns:

```
{AHB 266718 0xFF900004 {0x00000084 0x00000084 0x00000084 0x00000084} R 0 2 2 0
0 0 0 0} {AHB 266727 0xFF900000 {0x0000000D 0x0000000D 0x0000000D 0x0000000D}
W 0 2 2 0 0 0 0 0} {AHB 266760 0x000085C0 {0xC2042054 0x80A06000 0x02800003
0x01000000} R 0 2 4 1 0 0 0 0} {AHB 266781 0x000085D0 ...
```

Print 2 rows

```
grmon3> at 2
TIME    ADDRESS  D[127:96] D[95:64] D[63:32] D[31:0]  TYPE    ...
266986  00000800  91D02000 01000000 01000000 01000000 read    ...
267007  00000810  91D02000 01000000 01000000 01000000 read    ...
```

TCL returns:

```
{AHB 266986 0x00000800 {0x91D02000 0x01000000 0x01000000 0x01000000} R 0 2 4 1
0 0 0 0} {at 267007 0x00000810 {0x91D02000 0x01000000 0x01000000 0x01000000}
R 0 3 4 1 0 0 0 0}
```

SEE ALSO

Section 3.4.9, “Using the trace buffer”

tmode

6. batch - syntax

NAME

batch - Execute a batch script

SYNOPSIS

batch *?options? filename ?args...?*

DESCRIPTION

batch

Execute a TCL script. The **batch** is similar to the TCL command source, except that the batch command sets up the variables argv0, argv and argc in the global namespace. While executing the scrip, argv0 will contain the script filename, argv will contain a list of all the arguments that appear after the filename and argc will be the length of argv.

OPTIONS

-echo

Echo all commands/procedures that the TCL interpreter calls.

-prefix *?string?*

Print a prefix on each row when echoing commands. Has no effect unless -echo is also set.

RETURN VALUE

Command **batch** has no return value.

7. bdump - syntax

NAME

`bdump` - Dump memory to a file.

SYNOPSIS

```
bdump address length ?filename?
```

DESCRIPTION

The `bdump` command may be used to store memory contents a binary file. It's an alias for 'dump -binary'.

```
bdump address length ?filename?
```

Dumps *length* bytes, starting at *address*, to a file in binary format. The default name of the file is "grmon-dump.bin"

RETURN VALUE

Command `bdump` has no return value.

EXAMPLE

```
Dump 32KiB of data from address 0x40000000  
grmon3> bdump 0x40000000 32768
```

8. blood - syntax

NAME

blood - Load a binary file

SYNOPSIS

```
blood ?options...? filename ?address? ?cpu#?
```

DESCRIPTION

The blood command may be used to upload a binary file to the system. It's an alias for 'load -binary'. When a file is loaded, GRMON will reset the memory controllers registers first.

```
blood ?options...? filename ?address? ?cpu#?
```

The load command may be used to upload the file specified by *filename*. If the *address* argument is present, then binary files will be stored at this address, if left out then they will be placed at the base address of the detected RAM. The *cpu#* argument can be used to specify which CPU it belongs to. The options is specified below.

OPTIONS

-delay ms

The -delay option can be used to specify a delay between each word written. If the delay is non-zero then the maximum block size is 4 bytes.

-bsize bytes

The -bsize option may be used to specify the size blocks of data in bytes that will be written. Sizes that are not even words may require a JTAG based debug link to work properly. See Chapter 5, *Debug link* for more information.

-wprot

If the -wprot option is given then write protection on the core will be disabled

RETURN VALUE

Command **blood** returns a guessed entry point.

EXAMPLE

Load and then verify a binary data file at a 16MiB offset into the main memory starting at 0x40000000.

```
grmon3> blood release/ramfs.cpio.gz 0x41000000  
grmon3> verify release/ramfs.cpio.gz 0x41000000
```

SEE ALSO

Section 3.4.2, "Uploading application and data to target memory"

9. bp - syntax

NAME

bp - Add, delete or list breakpoints

SYNOPSIS

```

bp ?address? ?cpu#?
bp ?filename:linenum? ?cpu#?
bp type ?options? address ?mask? ?cpu#?
bp type ?options? filename:linenum ?mask? ?cpu#?
bp delete ?index?
bp enable ?index?
bp disable ?index?
bp map
bp map vaddr paddr
bp map clear

```

DESCRIPTION

The bp command may be used to list, add or delete all kinds of breakpoints. The *address* parameter that is specified when creating a breakpoint can either be an address or symbol. This parameter can be replaced with the filename and line number of a source file, if debug information is available, a so called line breakpoint. If there is no instruction corresponding to that particular line number, the breakpoint will instead be planted on the next line number that do have one. This happens, for instance, if you try to plant a breakpoint on a line consisting of a comment or a declaration. *mask* parameter can be used to break on a range of addresses. If omitted, the default value is 0xffffffff (i.e. a single address).

Software breakpoints are inserted by replacing an instruction in the memory with a breakpoint instruction. I.e. any CPU in a multi-core system that encounters this breakpoint will break.

Hardware breakpoints/watchpoints will be set to a single CPU core.

When adding a breakpoint a *cpu#* may optionally be specified to associate the breakpoint with a CPU. The CPU index will be used to lookup symbols, MMU translations and for hardware breakpoints/watchpoints.

```
bp ?address? ?cpu#?
```

When omitting the address parameter this command will list breakpoints. If the address parameter is specified, it will create a software breakpoint.

```
bp ?filename:linenum? ?cpu#?
```

When omitting the filename, semi-colon and linenum parameters this command will list breakpoints. If the parameters are specified, it will create a software breakpoint.

```
bp soft address ?cpu#?
```

Create a software breakpoint.

```
bp soft filename:linenum ?cpu#?
```

Create a software breakpoint.

```
bp hard address ?mask? ?cpu#?
```

Create a hardware breakpoint.

```
bp hard filename:linenum ?mask? ?cpu#?
```

Create a hardware breakpoint.

```
bp watch ?options? address ?mask? ?cpu#?
```

Create a hardware watchpoint. The options *-read/-write* can be used to make it watch only reads or writes, by default it will watch both reads and writes.

bp bus *?options? address ?mask? ?cpu#?*

Create an AMBA-bus watchpoint. The options `-read`/`-write` can be used to make it watch only reads or writes, by default it will watch both reads and writes.

bp data *?options? value ?mask? ?cpu#?*

Create an AMBA data watchpoint. The *value* and *mask* parameters may be up to 128 bits, but number of bits used depends on width of the bus on the system. Valid options are `-addr` and `-invert`. If `-addr` is specified, then also `-read` or `-write` are valid. See below for a description of the options.

bp delete *?index..?*

When omitting the index all breakpoints will be deleted. If one or more indexes are specified, then those breakpoints will be deleted. Listing all breakpoints will show the indexes of the breakpoints.

bp enable *?index..?*

When omitting the index all breakpoints will be enabled. If one or more indexes are specified, then those breakpoints will be enabled. Listing all breakpoints will show the indexes of the breakpoints.

bp disable *?index..?*

When omitting the index all breakpoints will be disabled. If one or more indexes are specified, then those breakpoints will be disabled. Listing all breakpoints will show the indexes of the breakpoints.

bp map

List the memory mapping used for soft breakpoints when the MMU is not yet available.

bp map *vaddr paddr*

Setup memory mapping from virtual to physical addresses to be used with soft breakpoints when the MMU is not yet available.

bp map clear

Clears the memory mapping used for soft breakpoints when the MMU is not yet available.

OPTIONS

`-read`

This option will enable a watchpoint to only watch loads at the specified address. The `-read` and `-write` are mutual exclusive.

`-write`

This option will enable a watchpoint to only watch stores at the specified address. The `-read` and `-write` are mutual exclusive.

`-addr address mask`

This option will combine an AMBA data watchpoint with a a bus watchpoint so it will only trigger if a value is read accessed from a certain address range.

`-invert`

The AMBA data watchpoint will trigger if value is NOT set.

`--`

End of options. This might be needed to set if value the first parameter after the options is negative.

RETURN VALUE

Command **bp** returns an breakpoint id when adding a new breakpoint.

When printing all breakpoints, a list will be returned containing one element per breakpoint. Each element has the format: {ID ADDR MASK TYPE ENABLED CPU SYMBOL FILE {DATA INV DATAMASK}}. AMBA watchpoints and AMBA data watchpoints will only have associated CPUs if has a symbol. The last subelement is only valid for AMBA data watchpoints. The file subelement will show filename and line number of the breakpoint and is only valid for line breakpoints. The line number of a line breakpoint is adjusted if there is no corresponding instruction on that line, but the line number shown by this command is the requested one.

EXAMPLE

Create a software breakpoint at the symbol main:

```
grmon3> bp soft main
```

Create an AMBA bus watchpoint that watches loads in the address range of 0x40000000 to 0x400000FF:

```
grmon3> bp bus -read 0x40000000 0xFFFFFFFF00
```

Create a breakpoint at line 1000 in the file stanford.c:

```
grmon3> bp stanford.c:1000
```

Listing breakpoints:

```
grmon3> bp
NUM  ADDRESS      MASK          TYPE          EN CPU  SYMBOL          FILE
 1 : 0x40000000  0xffffffff00 (bus r)       Y  0  1  __bcc_entry_point+0
 2 : 0x4000498c                (soft)        Y  0          Fft+804          stanford.c:1000
 3 : 0x40004c18                (soft)        Y  0  1  main+4
 4 : 0x40001958  0xffffffff    (hard)        Y  0  1  tower+4
```

SEE ALSO

Section 3.4.4, “Inserting breakpoints and watchpoints”

10. bt - syntax

NAME

bt - Print backtrace

SYNOPSIS

bt *?cpu#?*

DESCRIPTION

bt *?cpu#?*

Print backtrace on current active CPU, optionally specify which CPU to show.

RETURN VALUE

Upon successful completion **bt** returns a list of tuples, where each tuple consist of a PC- and SP-register values.

EXAMPLE

Show backtrace on current active CPU
grmon3> bt

TCL returns:

```
{1073746404 1342177032} {1073746020 1342177136} {1073781172 1342177200}
```

Show backtrace on CPU 1
grmon3> bt cpu1

TCL returns:

```
{1073746404 1342177032} {1073746020 1342177136} {1073781172 1342177200}
```

SEE ALSO

Section 1.6, “NOEL-V Support”
Section 3.4.6, “Backtracing function calls”

11. **cctrl** - syntax

NAME

cctrl - Display or set cache control register

SYNOPSIS

cctrl *?options? ?value? ?cpu#?*

cctrl flush *?cpu#?*

DESCRIPTION

cctrl *?options? ?value? ?cpu#?*

Display or set cache control register

cctrl flush *?cpu#?*

Flushes both instruction and data cache

OPTIONS

-v

-x

If option *-v* is specified, then GRMON will print the field names and values

If option *-x* is specified, then GRMON will interpret the specified *value*, and print its field information, without writing the the value to the register. This option requires the *value* argument.

RETURN VALUE

Upon successful completion **cctrl** will return the value of the cache control register.

SEE ALSO

-nic and *-ndc* switches described in Section 6.3.1, “Switches”

SEE ALSO

Section 3.4.16, “CPU cache support”

12. cont - syntax

NAME

cont - Continue execution

SYNOPSIS

`cont` *?options?* *?count?*

DESCRIPTION

`cont` *?options?* *?count?*

This command will continue the execution of instructions on the active CPU at the current location.

If the *count* parameter is set then the CPU will run the specified number of instructions. Note that the *count* parameter is only supported by the DSU4.

OPTIONS

-noret

Do not evaluate the return value. Then this options is set, no return value will be set.

RETURN VALUE

Upon successful completion `cont` returns a list of signals, one per CPU. Possible signal values are SIGBUS, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM or SIGTRAP. If a CPU is disabled, then a empty string will be returned instead of a signal value.

EXAMPLE

Continue execution from current PC
grmon3> cont

SEE ALSO

Section 3.4.3, "Running applications"

13. cpu - syntax

cpu - Enable, disable CPU or select current active CPU

SYNOPSIS

cpu

cpu active *cpuid*

cpu count

cpu enable *cpuid*

cpu disable *cpuid*

DESCRIPTION

Control processors in LEON3 multi-processor (MP) systems.

cpu

Without parameters, the **cpu** command prints the processor status.

cpu active *cpuid*

Set current active CPU

cpu count *cpuid*

Return the number of CPUs

cpu enable *cpuid*

cpu disable *cpuid*

Enable/disable the specified CPU.

RETURN VALUE

Upon successful completion **cpu** returns the active CPU and a list of booleans, one per CPU, describing if they are enabled or disabled.

The command **cpu count** returns the number of CPUs

The other sub commands has no return value.

EXAMPLE

Set current active to CPU 1

```
grmon3> cpu active 1
```

Print processor status in a two-processor system when CPU 1 is active and disabled.

```
grmon3> cpu
```

TCL returns:

```
1 {1 0}
```

SEE ALSO

Section 3.4.13, “Multi-processor support”

14. dcache - syntax

NAME

dcache - Show, enable or disable data cache

SYNOPSIS

```

dcache ?boolean? ?cpu#?
dcache diag ?windex? ?lindex? ?cpu#?
dcache flush ?cpu#?
dcache way windex ?lindex? ?word value? ?cpu#?
dcache tag windex lindex ?value? ?tbmask? ?cpu#?
dcache stag windex lindex ?value? ?tbmask? ?cpu#?

```

DESCRIPTION

In all forms of the **dcache** command, the optional parameter *?cpu#?* specifies which CPU to operate on. The active CPU will be used if parameter is omitted.

dcache *?boolean? ?cpu#?*

If *?boolean?* is not given then show the content of all ways. If *?boolean?* is present, then enable or disable the data cache.

dcache diag *?windex? ?lindex? ?cpu#?*

Check if the data cache is consistent with the memory. Optionally a specific way or line can be checked.

dcache flush *?cpu#?*

Flushes the data cache

dcache way *windex ?lindex? ?word value? ?cpu#?*

Show the contents of specified way *windex* or optionally a specific line *?lindex?*. If *word* and *value* is set then it will write a single 32-bit word into position specified by word.

dcache tag *windex lindex ?value? ?tbmask? ?cpu#?*

Read or write a raw data cache tag value. Way and line is selected with *windex* and *lindex*. The parameter *value*, if given, is written to the tag. The optional parameter *tbmask* is xor-ed with the test check bits generated by the cache controller during the write.

dcache stag *windex lindex ?value? ?tbmask? ?cpu#?*

Read or write a raw data cache snoop tag value. Way and line is selected with *windex* and *lindex*. The parameter *value*, if given, is written to the snoop tag. The optional parameter *tbmask* is xor-ed with the test check bits generated by the cache controller during the write.

RETURN VALUE

Command **dcache diag** returns a list of all inconsistent entries. Each element of the list contains CPU id, way id, line id, word id, physical address, cached data and the data from the memory.

Command **dcache tag** returns the tag value on read.

The other **dcache** commands have no return value.

SEE ALSO

Section 3.4.16, "CPU cache support"

icache

15. dccfg - syntax

NAME

dccfg - Display or set data cache configuration register

SYNOPSIS

dccfg *?value?* *?cpu#?*

DESCRIPTION

dccfg *?value?* *?cpu#?*

Display or set data cache configuration register for the active CPU. GRMON will not keep track of this register value and will not reinitialize the register when starting or resuming software execution.

RETURN VALUE

Upon successful completion **dccfg** will return the value of the data cache configuration register.

SEE ALSO

-nic and -ndc switches described in Section 6.3.1, “Switches”

SEE ALSO

Section 3.4.16, “CPU cache support”

16. dcom - syntax

NAME

dcom - Print or clear debug link statistics

SYNOPSIS

dcom

dcom clear

DESCRIPTION

dcom

dcom clear

Print debug link statistics.

Clear debug link statistics.

RETURN VALUE

Upon successful completion **dcom** has no return value.

17. ddr2cfg1 - syntax

ddr2cfg1 - Show or set the reset value of the memory register

SYNOPSIS

ddr2cfg1 *?value?*

DESCRIPTION

ddr2cfg1 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

RETURN VALUE

Upon successful completion **ddr2cfg1** returns a the value of the register.

SEE ALSO

Section 6.14, “Memory controllers ”

18. ddr2cfg2 - syntax

ddr2cfg2 - Show or set the reset value of the memory register

SYNOPSIS

ddr2cfg2 *?value?*

DESCRIPTION

ddr2cfg2 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

RETURN VALUE

Upon successful completion **ddrcfg2** returns a the value of the register.

SEE ALSO

Section 6.14, “Memory controllers ”

19. ddr2cfg3 - syntax

ddr2cfg3 - Show or set the reset value of the memory register

SYNOPSIS

ddr2cfg3 *?value?*

DESCRIPTION

ddr2cfg3 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

RETURN VALUE

Upon successful completion **ddrcfg3** returns a the value of the register.

SEE ALSO

Section 6.14, “Memory controllers ”

20. ddr2cfg4 - syntax

ddr2cfg4 - Show or set the reset value of the memory register

SYNOPSIS

ddr2cfg4 *?value?*

DESCRIPTION

ddr2cfg4 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

RETURN VALUE

Upon successful completion **ddr2cfg4** returns a the value of the register.

SEE ALSO

Section 6.14, “Memory controllers ”

21. ddr2cfg5 - syntax

ddr2cfg5 - Show or set the reset value of the memory register

SYNOPSIS

ddr2cfg5 *?value?*

DESCRIPTION

ddr2cfg5 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

RETURN VALUE

Upon successful completion **ddr2cfg5** returns a the value of the register.

SEE ALSO

Section 6.14, “Memory controllers ”

22. ddr2delay - syntax

ddr2delay - Change read data input delay

SYNOPSIS

ddr2delay ?subcommand? ?args...?

DESCRIPTION

ddr2delay inc ?steps?

ddr2delay dec ?steps?

ddr2delay ?value?

Use **inc** to increment the delay with one tap-delay for all data bytes. Use **dec** to decrement all delays. A *value* can be specified to calibrate each data byte separately. The *value* is written to the 16 LSB of the DDR2 control register 3.

ddr2delay reset

Set the delay to the default value.

ddr2delay scan

The scan subcommand will run a calibration routine that searches over all tap delays and read delay values to find working settings. Supports only Xilinx Virtex currently

The scan may overwrite beginning of memory.

RETURN VALUE

Command **ddr2delay** has no return value.

SEE ALSO

Section 6.14, “Memory controllers ”

23. ddr2skew - syntax

ddr2skew - Change read skew

SYNOPSIS

ddr2skew *?subcommand?* *?args...?*

DESCRIPTION

ddr2skew inc *?steps?*

ddr2skew dec *?steps?*

Increment/decrement the delay with one step. Commands **inc** and **dec** can optionally be given the number of steps to increment/decrement as an argument.

ddr2skew reset

Set the skew to the default value.

RETURN VALUE

Command **ddr2skew** has no return value.

SEE ALSO

Section 6.14, “Memory controllers ”

24. detach - syntax

detach - Resume execution with GRMON detached from processor

SYNOPSIS

detach

DESCRIPTION

detach

This command will detach GRMON and resume execution on enabled CPUs.

RETURN VALUE

Command **detach** has no return value.

25. disassemble - syntax

disassemble - Disassemble memory

SYNOPSIS

disassemble *?options? ?address? ?length? ?cpu#?*

DESCRIPTION

disassemble *?options? ?address? ?length? ?cpu#?*

Disassemble memory. If *length* is left out it defaults to 16 and the address defaults to current PC value. Symbols may be used as address.

OPTIONS

-p

Interpret addresses as physical addresses.

-r start stop

Disassemble a range of instructions between address *start* and up to *stop* (excluding *stop*). The arguments *address* and *length* will be ignored

RETURN VALUE

Command **disassemble** has no return value.

SEE ALSO

Section 3.4.7, “Displaying memory contents”

26. dtb - syntax

NAME

dtb - Setup a DTB to be uploaded or print filenames of DTB files.

SYNOPSIS

dtb *?options...? filename ?cpu#...?*

dtb *subcommand ?arg?*

DESCRIPTION

The dtb command may be used to setup a DTB file that will be upload to the system when GRMON resets the system. It can also be used to list all DTBs that will be loaded.

dtb *?options...? filename ?cpu#...?*

The dtb command may be used to setup a DRTB file, specified by *filename*, that will be uploaded when GRMON resets the system. The DTBs will be placed at the top of the stack or the address specified by the *-addr*. GRMON will also write the address of the DTB to input registers of the CPU:s so the application can find it. On RISC-V architectures the address will stored in register a1. One or more *cpu#* arguments can be used to specify which CPUs it belongs to or all CPUs if omitted

dtb clear *?cpu#...?*

This command will clear the information about the DTBs that will be loaded to the CPU:s. If one or more *cpu#* arguments is specified, then only those CPUs will be listed.

dtb load *?cpu#...?*

This command will load the DTBs and write the address to CPU:s register. It can be used to manually initialize the system. If one or more *cpu#* arguments are specified, then only those CPUs will be initialized.

dtb show *?cpu#...?*

This command will list which DTBs that will be loaded to the CPU:s. If one or more *cpu#* arguments are specified, then only those CPUs will be listed.

dtb print *?cpu#...?*

This command will decompile and print the current DTBs that will be loaded to the CPU:s. If one or more *cpu#* arguments are specified, then only those CPUs will be listed.

dtb str path *?prop? ?values...? ?cpu#...?*

Append string to the property on the node. If the property is left out, then then it will only try to create an empty node. If the values is left out, then then it will only try to create an empty property.

dtb u64 path prop *?values...? ?cpu#...?*

Append 64-bit unsigned integer to the property on the node. If the values is left out, then then it will only try to create an empty property.

dtb u32 path prop *?values...? ?cpu#...?*

Append 32-bit unsigned integer to the property on the node. If the values is left out, then then it will only try to create an empty property.

dtb bytes path prop *?values...? ?cpu#...?*

Append bytes to the property on the node. If the values is left out, then then it will only try to create an empty property.

OPTIONS

-bsize bytes

The **-bsize** option may be used to specify the size blocks of data in bytes that will be written. Sizes that are not even words may require a JTAG based debug link to work properly. See Chapter 5, *Debug link* more information.

-delay ms

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 4 bytes, but can be changed using the `-bsize` option.

-wprot

If the `-wprot` option is given then write protection on the core will be disabled

-addr

Specify address where the DTB will be stored.

-rom

If the DTB is already present in the memory and only the input registers needs to be setup. and `-addr` is required.

-dts

Filename specifies a DTS that will be compiled into a DTB.

RETURN VALUE

Command **dtb show** returns the a list. Each entry is a sublist on the format format: `{filename address {cpu-ids}}`. The filename will be the keyword ROM if `-rom` was used. The address will be the keyword "Stack" if it located on the stack.

27. dump - syntax

NAME

dump - Dump memory to a file.

SYNOPSIS

```
dump ?options...? address length ?filename?
```

DESCRIPTION

```
dump ?options...? address length ?filename?
```

Dumps *length* bytes, starting at *address*, to a file in Motorola SREC format. The default name of the file is "grmon-dump.srec"

OPTIONS

-binary

The -binary option can be used to store data to a binary file

-bsize

The -bsize option may be used to specify the size blocks of data in bytes that will be read. Sizes that are not even words may require a JTAG based debug link to work properly. See Chapter 5, *Debug link* more information.

-append

Set the -append option to append the dumped data to the end of the file. The default is to truncate the file to zero length before storing the data into the file.

RETURN VALUE

Command **dump** has no return value.

EXAMPLE

Dump 32KiB of data from address 0x40000000
grmon3> dump 0x40000000 32768

28. edcl - syntax

NAME

edcl - Print or set the EDCL ip

SYNOPSIS

```
edcl ?ip? ?greth#?
```

DESCRIPTION

```
edcl ?ip? ?greth#?
```

If an ip-address is supplied then it will be set, otherwise the command will print the current EDCL ip. The EDCL will be disabled if the ip-address is set to zero and enabled if set to a normal address. If more than one device exists in the system, the *dev#* can be used to select device, default is dev0.

RETURN VALUE

Command **edcl** has no return value.

EXAMPLE

```
Set ip-address 192.168.0.123  
grmon3> edcl 192.168.0.123
```

SEE ALSO

Section 6.4, "Ethernet controller"

29. eeload - syntax

NAME

eeload - Load a file into an EEPROM

SYNOPSIS

```
eeload ?options...?filename ?cpu#?
```

DESCRIPTION

The eeload command may be used to upload a file to a EEPROM. It's an alias for 'load -delay 1 -bsize 4 -wprot'. When a file is loaded, GRMON will reset the memory controllers registers first.

```
eeload ?options...?filename ?address? ?cpu#?
```

The load command may be used to upload the file specified by *filename*. It will also try to disable write protection on the memory core. If the *address* argument is present, then binary files will be stored at this address, if left out then they will be placed at the base address of the detected RAM. The *cpu#* argument can be used to specify which CPU it belongs to. The options is specified below.

OPTIONS

-binary

The -binary option can be used to force GRMON to interpret the file as a binary file.

-bsize bytes

The -bsize option may be used to specify the size blocks of data in bytes that will be written. Valid value are 1, 2 or 4. Sizes 1 and 2 may require a JTAG based debug link to work properly See Chapter 5, *Debug link* more information.

RETURN VALUE

Command **eeload** returns the entry point.

EXAMPLE

Load and then verify a hello_world application

```
grmon3> eeload ../hello_world/hello_world  
grmon3> verify ../hello_world/hello_world
```

SEE ALSO

Section 3.4.2, "Uploading application and data to target memory"

30. ehci - syntax

NAME

ehci - Control the USB host's EHCI core

SYNOPSIS

ehci subcommand *?args...?*

DESCRIPTION

ehci endian *?devname?*

Displays the endian conversion setting

ehci capregs *?devname?*

Displays contents of the capability registers

ehci opregs *?devname?*

Displays contents of the operational registers

ehci reset *?devname?*

Performs a Host Controller Reset

RETURN VALUE

Upon successful completion, **ehci** have no return value.

SEE ALSO

Section 6.6, “USB Host Controller”

31. ei - syntax

NAME

ei - Inject errors in CPU cache and register files

SYNOPSIS

ei *subcommand ?args...?*

DESCRIPTION

Errors will be injected according to the CPU configuration. Injection of errors in ITAG, IDATA, DTAG, DDATA, STAG, IU register file and FP register file is supported.

ei un *?nr t?*

Enable error injection, uniform error distribution mode. *nr* errors are inserted during the time period of *t* minutes. Errors are uniformly distributed over the time period.

ei av *?r?*

Enable error injection, average error rate mode. Errors will be inserted during the whole program execution. Average error rate is *r* errors per second.

ei disable

Disable error injection.

ei log *?filename?*

ei log *disable*

Enable/disable error injection log. The error injection log is saved in file *log_file*.

ei stat

ei stat *?enable?*

ei stat *?disable?*

Show, enable or disable error injection statistics. When enabled, the SEU correction counters are modified. This option should not be used with software which itself monitors SEU error counters.

ei prob

ei prob *itag dtag idata ddata stag iurf fprf ?cpu#?*

Show or set probability of each error injection target. Each injection target has an associated probability value from 0.0 to 1.0. The value 0.0 means that no errors will be injected in the target. A value higher than 0.0 means that the error will be injected with the specified probability.

When no parameter is given to **ei prob**, then the currently configured values are listed. The second form configures the probabilities from user supplied decimal numbers. Target CPU is selected with the *cpu#* parameter. If no CPU parameter is given, then the current CPU is used.

RETURN VALUE

Command **ei** has no return value.

EXAMPLE

Configure **ei** to inject errors only in the data cache tags and instruction cache tags (DTAG and ITAG) of *cpu0*:

```
grmon3> ei prob 1.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 cpu0
```

```
grmon3> ei prob 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 cpu1
```

List the currently configured target probabilities:

```
grmon3> ei prob
```

SEE ALSO

Section 3.10.2, “LEON3-FT error injection”

icache

dcache

32. ep - syntax

NAME

ep - Set entry point

SYNOPSIS

```
ep ?cpu#?  
ep ?--? value ?cpu#?  
ep disable ?cpu#?
```

DESCRIPTION

ep ?cpu#?

Show current active CPUs entry point, or the CPU specified by `cpu#`.

ep ?--? value ?cpu#?

Set the current active CPUs entry point, or the CPU specified by `cpu#`. The only option available is '--' and it marks the end of options. It should be used if a symbol name is in conflict with a subcommand (i.e. a symbol called "disable").

ep disable ?cpu#?

Remove the entry point from the current active CPU or the the CPU specified by `cpu#`.

RETURN VALUE

Upon successful completion **ep** returns a list of entry points, one for each CPU. If `cpu#` is specified, then only the entry point for that CPU will be returned.

EXAMPLE

```
Set current active CPUs entry point to 0x40000000  
grmon3> ep 0x40000000
```

SEE ALSO

Section 3.4.13, "Multi-processor support"

33. grmon::execsh - syntax

NAME

grmon::execsh - Run commands in the execution shell

SYNOPSIS

grmon::execsh
grmon::execsh *subcommand ?arg?*

DESCRIPTION

The `grmon::execsh` command is used to execute scripts in the execution shell. This command should be used to manage execution hooks.

grmon::execsh eval *?options? arg ?arg ...?*

Evaluate command *arg* in the execution shell. If a script is running, then the command will fail with the error result set to EBUSY.

grmon::execsh interrupt *name*

Send an interrupt to the execution shell.

OPTIONS

-u

The `-u` option may be added to use the I/O forwarding settings of the calling shell.

RETURN VALUE

grmon::execsh eval will return the result from the script.

EXAMPLE

Install an execution hook

```
grmon::execsh eval {  
  proc myhook1 {} {puts "Hello World"}  
  lappend ::hooks::preexec ::myhook1  
}
```

SEE ALSO

Section 3.5, “Tcl integration”
Appendix C, *Tcl API*

34. exit - syntax

NAME

exit - Shut down the GRMON application

SYNOPSIS

exit *?code?*

DESCRIPTION

exit *?code?*

Shut down the GRMON application. If an code is set the GRMON will set it as an exit code. Values between 0 and 255 are valid, other values will be saturated to 255. If no code is specified, then GRMON will set 0 as exit code if no internal error has occurred or a value greater or equal to 1 to indicate an internal error

RETURN VALUE

Command **exit** has no return value.

EXAMPLE

Shut down GRMON application with exit code 1.
grmon3> exit 1

35. flash - syntax

NAME

flash - Write, erase or show information about the flash

SYNOPSIS

flash

flash blank all

flash blank *start ?stop?*

flash burst *?boolean?*

flash erase all

flash erase *start ?stop?*

flash load *?options...? filename ?address? ?cpu#?*

flash verify *?options...? filename ?address?*

flash lock all

flash lock *start ?stop?*

flash lockdown all

flash lockdown *start ?stop?*

flash query

flash scan *?addr?*

flash status

flash unlock all

flash unlock *start ?stop?*

flash wbuf *length*

flash write *address data*

DESCRIPTION

GRMON supports programming of CFI compatible flash PROM attached to the external memory bus of LEON2 and LEON3 systems. Flash programming is only supported if the target system contains one of the following memory controllers MCTRL, FTMCTRL, FTSRCTRL or SSRCTRL. The PROM bus width can be 8-, 16- or 32-bit. It is imperative that the prom width in the MCFG1 register correctly reflects the width of the external prom. To program 8-bit and 16-bit PROMs, the target system must also have at least one working SRAM or SDRAM bank.

When one of the flash commands are issued GRMON will probe for a CFI compatible memory at the beginning of the PROM area. GRMON will only control one flash memory at the time. If there are multiple CFI compatible flash memories connected to the PROM area, then it is possible to switch device using the command **flash scan** *addr*. If the PROM width or bank size is changed in the memory controller registers are changed, then GRMON will discard any probed CFI information, and a new **flash scan** command have to be issued.

There are many different suppliers of CFI devices, and some implements their own command set. The command set is specified by the CFI query register 14 (MSB) and 13 (LSB). The value for these register can in most cases be found in the datasheet of the CFI device. GRMON supports the command sets that are listed in Table 3.4, "Supported CFI command set" in section Section 3.11.3, "CFI compatible Flash PROM".

The sub commands erase, lock, lockdown and unlock works on memory blocks (the subcommand blank have the same parameters, but operates on addresses). These commands operate on the block that the *start* address belong. If the *stop* parameter is also given the commands will operate on all the blocks between and including the blocks that the *start* and *stop* belongs to. I.a the keyword 'all' can be given instead of the start address, then the command will operate on the whole memory.

flash

Print the flash memory configuration.

flash blank all

flash blank *start ?stop?*

Check that the flash memory is blank, i.e. can be re-programmed. See description above about the parameters.

flash burst *?boolean?*

Enable or disable flash burst write. Disabling the burst will decrease performance and requires either that a CPU is available in the system or that a JTAG debug link is used. This feature is only has effect when a 8-bit or 16-bit Intel style flash memory that is connected to a memory controller that supports bursting.

flash erase all

flash erase *start ?stop?*

Erase a flash block. See description above about the parameters.

flash load *?options...? filename ?address? ?cpu#?*

Program the flash memory with the contents file. The load command may be used to upload the file specified by *filename*. If the *address* argument is present, then binary files will be stored at this address, if left out then they will be placed at the base address of the detected ROM. The *cpu#* argument can be used to specify which CPU it belongs to.

The `-binary` option can be used to force GRMON to interpret the file as a binary file.

The `-erase` option to automatically erase the flash before writing. It will only erase the blocks where data will be written.

The `-nolock` option can be used to prevent GRMON from checking the protection bits to see if the block is locked before trying to load data to the block.

flash verify *?options...? filename ?address?*

Verify that the file *filename* has been uploaded correctly, if EDAC is enabled then the checkbits will be verified as well. If the *address* argument is present, then binary files will be compared against data at this address, if left out then they will be compared to data at the base address of the detected RAM.

The `-binary` option can be used to force GRMON to interpret the file as a binary file.

The `-max` option can be used to force GRMON to stop verifying when num errors have been found.

flash lock all

flash lock *start ?stop?*

Lock a flash block. See description above about the parameters.

flash lockdown all

flash lockdown *start ?stop?*

Lockdown a flash block. Work only on Intel-style devices which supports lock-down. See description above about the parameters.

flash query

Print the flash query registers

flash scan *?addr?*

Probe the address for a CFI flash. If the *addr* parameter is set, then GRMON will probe for a new memory at the address. If the *addr* parameter is unset, GRMON will probe for a new memory at the beginning of the PROM area. If the *addr* parameter is unset, and a memory has already been probed, then GRMON will only return the address of the last probed memory.

flash status

Print the flash lock status register

flash unlock all

flash unlock *start ?stop?*

Unlock a flash block. See description above about the parameters.

flash wbuf *length*

Limit the CFI auto-detected write buffer length. Zero disables the write buffer command and will perform single-word access only. -1 will reset to auto-detected value.

flash write *address data*

Write a 32-bit data word to the flash at address *addr*.

RETURN VALUE

Command **flash scan** returns the base address of the CFI compatible memory.

The other **flash** commands has no return value.

EXAMPLE

A typical command sequence to erase and re-program a flash memory could be:

```
grmon3> flash unlock all
grmon3> flash erase all
grmon3> flash load file.prom
grmon3> flash lock all
```

SEE ALSO

Section 3.11.3, “CFI compatible Flash PROM”

36. float - syntax

NAME

float - Display FPU registers

SYNOPSIS

float

DESCRIPTION

float

Display FPU registers

RETURN VALUE

Upon successful completion **float** returns 2 lists. The first list contains the values when the registers represents floats, and the second list contain the double-values.

SEE ALSO

Section 3.4.5, “Displaying processor registers”

37. forward - syntax

NAME

forward - Control I/O forwarding

SYNOPSIS

forward

forward list

forward enable *devname* *?channel?*

forward disable *devname*

forward mode *devname value*

forward start

DESCRIPTION

forward

forward list

List all enabled devices in the current shell.

forward enable *devname* *?channel?*

Enable I/O forwarding for a device. If a custom channel is not specified, then the default channel for the shell will be enabled. The I/O forwarding configuration is stored per shell.

forward disable *devname*

Disable I/O forwarding for a device.

forward mode *devname value*

Set forwarding mode. Valid values are "loopback", "debug" or "none".

forward start

Start forwarding I/O in the current shell. When executing an application using GDB, this can be used to redirect I/O to the command line shell instead of to GDB. Issue an interrupt (Ctrl-C) to return to the GRMON prompt.

The forwarding can be started when the GRMON has detached from the system as well. It will poll the UART regularly, use `-poll us` to set how often GRMON should poll.

RETURN VALUE

Upon successful completion **forward** has no return value.

EXAMPLE

Enable I/O forwarding

```
grmon3> forward enable uart0
```

Enable I/O forwarding to a file

```
grmon3> forward enable uart0 [open "grmon3.out" w]
```

38. fpgalload - syntax

NAME

fpgalload - Upload bitstream to GR740-MINI board FPGA.

SYNOPSIS

```
fpgalload ?options...?filename
```

DESCRIPTION

The fpgalload command may be used to upload a bitfile to the Lattice FPGA on the GR740-MINI board. See GR740-MINI board documentation for additional information about usage and requirements.

```
fpgalload ?options...?filename
```

The fpgalload command may be used to upload the file specified by *filename*. The options are specified below.

OPTIONS

-v

Produce verbose output

-f

Write bitstream in flash

RETURN VALUE

No return value

39. gdb - syntax

NAME

`gdb` - Control the built-in GDB remote server

SYNOPSIS

```
gdb ?port?  
gdb eval arg ?arg ...?  
gdb postload  
gdb reset  
gdb stop  
gdb status
```

DESCRIPTION

gdb *?port?*

Start the built-in GDB remote server, optionally listen to the specified port. Default port is 2222.

gdb eval *arg ?arg ...?*

Evaluate command *arg* in the GDB shell.

gdb postload

Called from the GDB prompt using a monitor command after a GDB load command has been issued to do some CPU additional initialization.

gdb reset

Called from the GDB prompt using a monitor command to reset the system.

gdb stop

Stop the built-in GDB remote server.

gdb status

Print status

RETURN VALUE

Only the command '**gdb status**' has a return value. Upon successful completion **gdb status** returns a tuple, where the first value represents the status (0 stopped, 1 connected, 2 waiting for connection) and the second value is the port number.

SEE ALSO

Section 3.7, "GDB interface"

Section 3.2, "Starting GRMON"

40. go - syntax

go - Start execution without any initialization

SYNOPSIS

```
go ?options? ?address? ?count?
```

DESCRIPTION

```
go ?options? ?address? ?count?
```

This command will start the execution of instructions on the active CPU.

When omitting the address parameter this command will start execution at the entry point from the last loaded application.

The command will clear the error mode of the CPUs, but not reset any drivers, unlike the **run** command.

If the *count* parameter is set then the CPU will run the specified number of instructions. Note that the *count* parameter is only supported by the DSU4.

OPTIONS

-noret

Do not evaluate the return value. Then this options is set, no return value will be set.

RETURN VALUE

Upon successful completion **go** returns a list of signals, one per CPU. Possible signal values are SIGBUS, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM or SIGTRAP. If a CPU is disabled, then a empty string will be returned instead of a signal value.

EXAMPLE

Execute instructions starting at 0x40000000.
grmon3> go 0x40000000

SEE ALSO

Section 3.4.3, "Running applications"

41. gr1553b - syntax

gr1553b - MIL-STD-1553B Interface commands

SYNOPSIS

gr1553b ?subcommand? ?args...?

DESCRIPTION

The **gr1553b** command is an alias for the **mil>** command. See help of command **mil>** for more information.

42. grcg - syntax

NAME

grcg - Control clock gating

SYNOPSIS

grcg *subcommand* *?args?* *?grcg#?*

DESCRIPTION

This command provides functions to control the GRCLKGATE core. If more than one core exists in the system, then the name of the core to control should be specified as the last command option (after the subcommand). The 'info sys' command lists the controller names.

grcg clkinfo *?grcg#?*

Show register values.

grcg enable *number* *?grcg#?*

grcg disable *number* *?grcg#?*

Enable or disable a clock gate. Argument *number* may be replaced by the keyword `all`.

RETURN VALUE

Upon successful completion **grcg clkinfo** returns three masks, where each bit of the masks represents a clock gate. The first mask shows unlock-bits, the second enabled-bits and the third reset-bits.

The other sub commands has no return value.

EXAMPLE

Enable all clock gates

```
grmon3> grcg enable all
```

Clock enable function 7 on the GRCLKGATE core with index 1.

```
grmon3> grcg enable 7 grcg1
```


43. grpwm - syntax

NAME

grpwm - Control GRPWM core

SYNOPSIS

grpwm subcommand *?args...?*

DESCRIPTION

grpwm info *?devname?*

Displays information about the GRPWM core

grpwm wave *?devname?*

Displays the waveform table

RETURN VALUE

Command **grpwm wave** returns a list of wave data.

The other **grpwm** commands have no return value.

44. grtmx - syntax

grtmx - Control GRTM devices

SYNOPSIS

grtmx ?subcommand? ?args...?

DESCRIPTION

grtmx

Display status

grtmx reset

Reset DMA and TM encoder

grtmx release

Release TM encoder

grtmx rate *rate*

Set rate register

grtmx len *nbytes*

Set frame length (actual number of bytes)

grtmx limit *nbytes*

Set limit length (actual number of bytes)

grtmx on

grtmx off

Enable/disable the TM encoder

grtmx reg

List register contents

grtmx conf

List design options

RETURN VALUE

Command **grtmx** has no return value.

45. gui - syntax

NAME

gui - Control the graphical user interface

SYNOPSIS

gui
gui status

DESCRIPTION

gui

When GRMON has been started in CLI mode this command can be used to start the graphical user interface. This command has no effect if the GUI has already been started.

gui status

Print status for the GUI connection.

RETURN VALUE

Only the command '**gui status**' has a return value. Upon successful completion **gui status** returns a tuple, where the first value represents the status (0 stopped, 1 started) and the second value is a reserved number.

SEE ALSO

Chapter 4, *Graphical user interface*
Section 4.2, "Starting GRMON GUI"

46. help - syntax

NAME

help - Print all GRMON commands or detailed help for a specific command

SYNOPSIS

help *?command?*

DESCRIPTION

help *?command?*

When omitting the command parameter this command will list commands. If the command parameter is specified, it will print a long detailed description of the command.

RETURN VALUE

Command **help** has no return value.

EXAMPLE

List all commands:

```
grmon3> help
```

Show detailed help of command 'mem':

```
grmon3> help mem
```

47. hist - syntax

NAME

hist - Print AHB transfers or instruction entries in the trace buffer

SYNOPSIS

hist *?length?* *?cpu#?*

DESCRIPTION

hist *?length?*

Print the hist trace buffer. The *?length?* entries will be printed, default is 10. Use *cpu#* to select CPU.

RETURN VALUE

Upon successful completion, **hist** returns a list of mixed AHB and instruction trace buffer entries, sorted after time. The first value in each entry is either the literal string AHB or INST indicating the type of entry. For more information about the entry values, see return values described for commands **ahb** and **inst**.

EXAMPLE

Print 10 rows

```
grmon3> hist
TIME      ADDRESS    INSTRUCTIONS/AHB SIGNALS    RESULT/DATA
266951    000021D4   restore %o0, %o0             [0000000D]
266954    000019E4   mov 0, %g1                    [00000000]
266955    000019E8   mov %g1, %i0                  [00000000]
266956    000019EC   ret                            [000019EC]
266957    000019F0   restore                        [00000000]
266960    0000106C   call 0x00009904               [0000106C]
266961    00001070   nop                            [00000000]
266962    00009904   mov 1, %g1                    [00000001]
266963    00009908   ta 0x0                        [ TRAP ]
266986    00000800   AHB read  mst=0  size=4       [91D02000 01000000 01000000 0100]
```

TCL returns:

```
{INST 266951 0x000021D4 0x91E80008 0x0000000D 0 0 0} {INST 266954 0x000019E4
0x82102000 0x00000000 0 0 0} {INST 266955 0x000019E8 0xB0100001 0x00000000
0 0 0} {INST 266956 0x000019EC ...
```

Print 2 rows

```
grmon3> hist 2
TIME      ADDRESS    INSTRUCTIONS/AHB SIGNALS    RESULT/DATA
266963    00009908   ta 0x0                        [ TRAP ]
266986    00000800   AHB read  mst=0  size=4       [91D02000 01000000 01000000 0100]
```

TCL returns:

```
{INST 266963 0x00009908 0x91D02000 0x00000000 0 1 0} {AHB 266986 0x00000800
{0x91D02000 0x01000000 0x01000000 0x01000000} R 0 2 4 1 0 0 0}
```

SEE ALSO

Section 3.4.9, “Using the trace buffer”

48. i2c - syntax

NAME

`i2c` - Commands for the I2C masters

SYNOPSIS

`i2c subcommand ?args...?`

`i2c index subcommand ?args...?`

DESCRIPTION

This command provides functions to control the SPICTRL core. If more than one core exists in the system, then the index of the core to control should be specified after the `i2c` command (before the subcommand). The 'info sys' command lists the device indexes.

i2c bitrate *rate*

Initializes the prescaler register. Valid keywords for the parameter *rate* are `normal`, `fast` or `highspeed`.

i2c disable

i2c enable

Enable/Disable the core

i2c read *?options? i2caddr ?addr? ?cnt?*

Performs *cnt* sequential reads starting at memory location *addr* from slave with *i2caddr*. Default value of *cnt* is 1. If only *i2caddr* is specified, then a simple read will be performed.

Available options are `-d8`, `-d16` and `-d32` to control how many bits there are in each data word that is read. Default is 8 bits.

i2c scan

Scans the bus for devices.

i2c status

Displays some status information about the core and the bus.

i2c write *?options? i2caddr ?addr? data*

Writes *data* to memory location *addr* on slave with address *i2caddr*. If only *i2caddr* and *data* is specified, then a simple write will be performed.

Available options are `-d8`, `-d16` and `-d32` to control how many bits there are in each data word that is written. Default is 8 bits.

Commands to interact with DVI transmitters:

i2c dvi devices

List supported devices.

i2c dvi delay *direction*

Change delay applied to clock before latching data. Valid keywords for *direction* are `inc` or `dec`.

i2c dvi init_14itx_dvi *?idf?*

i2c dvi init_14itx_vga *?idf?*

Initializes Chrontel CH7301C DVI transmitter with values that are appropriate for the GR-LEON4-ITX board with DVI/VGA output. The optional *idf* value selects the multiplexed data input format, default is IDF 2.

i2c dvi init_ml50x_dvi *?idf?*

i2c dvi init_ml50x_vga *?idf?*

Initializes Chrontel CH7301C DVI transmitter with values that are appropriate for a ML50x board with a standard LEON/GRLIB template design for DVI/VGA output. The optional *idf* value selects the multiplexed data input format, default is IDF 2.

i2c dvi setdev *devnr*

Set DVI transmitter type. See command **i2c dvi devices** to list valid values of the parameter *devnr*.

i2c dvi showreg

Show DVI transmitter registers

RETURN VALUE

Upon successful completion **i2c read** returns a list of values read. The **i2c dvi showreg** return a list of tuples, where the first element is the register address and the second element is the value.

The other sub commands has no return value.

49. icache - syntax

NAME

icache - Show, enable or disable instruction cache

SYNOPSIS

```
icache ?boolean? ?cpu#?
icache diag ?windex? ?lindex? ?cpu#?
icache flush ?cpu#?
icache way windex ?lindex? ?word value? ?cpu#?
icache tag windex lindex ?value? ?tbmask? ?cpu#?
```

DESCRIPTION

In all forms of the **icache** command, the optional parameter *?cpu#?* specifies which CPU to operate on. The active CPU will be used if parameter is omitted.

icache *?boolean? ?cpu#?*

If *?boolean?* is not given then show the content of all ways. If *?boolean?* is present, then enable or disable the instruction cache.

icache diag *?windex? ?lindex? ?cpu#?*

Check if the instruction cache is consistent with the memory. Optionally a specific way or line can be checked.

icache flush *?cpu#?*

Flushes the instruction cache

icache way *windex ?lindex? ?word value? ?cpu#?*

Show the contents of specified way *windex* or optionally a specific line *?lindex?*. If *word* and *value* is set then it will write a single 32-bit word into position specified by word.

icache tag *windex lindex ?value? ?tbmask? ?cpu#?*

Read or write a raw instruction cache tag value. Way and line is selected with *windex* and *lindex*. The parameter *value*, if given, is written to the tag. The optional parameter *tbmask* is xor-ed with the test check bits generated by the cache controller during the write.

RETURN VALUE

Command **icache diag** returns a list of all inconsistent entries. Each element of the list contains CPU id, way id, line id, word id, physical address, cached data and the data from the memory.

Command **icache tag** returns the tag value on read.

The other **icache** commands have no return value.

SEE ALSO

Section 3.4.16, “CPU cache support”

dcache

50. iccfg - syntax

NAME

iccfg - Display or set instruction cache configuration register

SYNOPSIS

iccfg *?value?* *?cpu#?*

DESCRIPTION

iccfg *?value?* *?cpu#?*

Display or set instruction cache configuration register for the active CPU. GRMON will not keep track of this register value and will not reinitialize the register when starting or resuming software execution.

RETURN VALUE

Upon successful completion **iccfg** will return the value of the instruction cache configuration register.

SEE ALSO

-nic and -ndc switches described in Section 6.3.1, “Switches”

SEE ALSO

Section 3.4.16, “CPU cache support”

51. info - syntax

NAME

info - GRMON extends the TCL command info with some subcommands to show information about the system.

SYNOPSIS

info subcommand *?args...?*

DESCRIPTION

info drivers

List all available device-drivers

info mkprom2

List the most basic mkprom2 commandline switches. GRMON will print flags to use the first GPTIMER and IRQMP controller and it will use the same UART for output as GRMON (see Section 3.9, “Forwarding application console I/O”). I.a. it will produce switches for all memory controllers found. In case that there exist more the one controller it's up to the user make sure that only switches belonging to one controller are used.

info reg *?options? ?dev?*

Show system registers. If a device name is passed to the command, then only the registers belonging to that device is printed. The device name can be suffixed with colon and a register name to only print the specified register.

If option `-v` is specified, then GRMON will print the field names and values of each registers. If a debug driver doesn't support this feature, then the register value is printed instead.

Setting `-l` will print the name of the registers, that can be used to access the registers via TCL variables. It also returns a list of all the register names. No registers values will be read.

Setting `-a` will also return the address in the list of all the register names. Will only have an effect if `-l` is also set.

Setting `-d` will also return the description in the list of all the register names. Will only have an effect if `-l` is also set.

Setting `-x` will interpret a constant value, instead of reading the register value from the system. It requires that every `dev` argument is followed by a value to be interpreted, i.e `dev0::reg0 value0 dev1::reg1 value1 ...`

Enabling `-all` will print all registers. Normally only a subset is printed. This option may print a lot of registers. I could also cause read accesses to FIFOs.

info sys *?options? ?dev ...?*

Show system configuration. If one or more device names are passed to the command, then only the information about those devices are printed.

If option `-v` is specified, then GRMON will print verbose information about the devices.

The option `-xml <file>` can be used to print a xml description of the system to a file instead of printing information on the screen.

RETURN VALUE

info drivers has no return value.

info mkprom2 returns a list of switches.

The command **info reg** returns a list of all registers if the `-l` is specified. If both options `-l` and `-v` have been entered it returns a list where each element is a list of the register name and the name of the registers fields. Otherwise it has no return value.

Upon successful completion **info sys** returns a list of all device names.

For other info subcommands, see TCL documentation.

EXAMPLE

Show all devices in the system

```
grmon3> info sys
ahbjtag0 Frontgrade Gaisler JTAG Debug Link
        AHB Master 0
adev1    Frontgrade Gaisler EDCL master interface
        AHB Master 2
...
```

Show only the DSU

```
grmon3> info sys dsu0
dsu0    Frontgrade Gaisler LEON4 Debug Support Unit
        AHB: E0000000 - E4000000
        AHB trace: 256 lines, 128-bit bus
        CPU0: win 8, hwbp 2, itrace 256, V8 mul/div, srmmu, lddel 1, GRFPU
            stack pointer 0x07ffffff0
            icache 4 * 4 kB, 32 B/line lru
            dcache 4 * 4 kB, 32 B/line lru
        CPU1: win 8, hwbp 2, itrace 256, V8 mul/div, srmmu, lddel 1, GRFPU
            stack pointer 0x07ffffff0
            icache 4 * 4 kB, 32 B/line lru
            dcache 4 * 4 kB, 32 B/line lru
```

Show detailed information on status register of uart0.

```
grmon3> info reg -v uart0::status
Generic UART
0xff900004 UART Status register          0x00000086
31:26 rcnt          0x0          Rx FIFO count
25:20 tcnt          0x0          Tx FIFO count
10   rf             0x0          Rx FIFO full
...
```

SEE ALSO

Section 3.4.1, “Examining the hardware configuration”

52. inst - syntax

NAME

inst - Print AHB transfer or instruction entries in the trace buffer

SYNOPSIS

inst *?length?*

inst subcommand *?args...?*

DESCRIPTION

inst *?length?* *?cpu#?*

Print the inst trace buffer. The *?length?* entries will be printed, default is 10. Use *cpu#* to select single CPU.

inst filter *?cpu#?*

Print the instruction trace buffer filter.

inst filter *?flt?* *?cpu#?*

Set the instruction trace buffer filter. See DSU manual for values of *flt*. (Only available in some DSU4 implementations). Use *cpu#* to set filter select a single CPU.

inst filter asldigit *?val...?* *?cpu#?*

Set which last digits that should be filtered. Only valid if filter is set to 0xE. (Only available in some DSU implementations)

inst filter range *?index?* *?addr?* *?mask?* *?excl?* *?cpu#?*

Setup a trace filter to include or exclude instructions that is within the range. Up to four range filters is supported. (Only available in some DSU implementations)

RETURN VALUE

Upon successful completion, **inst** returns a list of trace buffer entries. Each entry is a sublist on the format format: {INST *time addr inst result trap em mc*}. Detailed description about the different fields can be found in the DSU core documentation in document grip.pdf [<http://gaisler.com/products/grlib/grip.pdf>]

The other subcommands have no return value.

EXAMPLE

Print 10 rows

```
grmon3> inst
TIME      ADDRESS  INSTRUCTION      RESULT
266951    000021D4  restore %o0, %o0  [0000000D]
266954    000019E4  mov 0, %g1        [00000000]
266955    000019E8  mov %g1, %i0      [00000000]
266956    000019EC  ret               [000019EC]
266957    000019F0  restore           [00000000]
266960    0000106C  call 0x00009904   [0000106C]
266961    00001070  nop              [00000000]
266962    00009904  mov 1, %g1        [00000001]
266963    00009908  ta 0x0            [ TRAP ]
267009    00000800  ta 0x0            [ TRAP ]
```

TCL returns:

```
{INST 266951 0x000021D4 0x91E80008 0x0000000D 0 0 0} {INST 266954 0x000019E4
0x82102000 0x00000000 0 0 0} {INST 266955 0x000019E8 0xB0100001 0x00000000
0 0 0} {INST 266956 0x000019EC ...
```

Print 2 rows

```
grmon3> inst 2
TIME      ADDRESS  INSTRUCTION      RESULT
```

```
266951 000021D4 restore %o0, %o0 [0000000D]
266954 000019E4 mov 0, %g1 [00000000]
```

TCL returns:

```
{INST 266951 0x000021D4 0x91E80008 0x0000000D 0 0 0} {INST 266954 0x000019E4
0x82102000 0x00000000 0 0 0}
```

SEE ALSO

Section 3.4.9, “Using the trace buffer”

53. iommu - syntax

NAME

iommu - Control IO memory management unit

SYNOPSIS

`iommu subcommand ?args? ?iommu#?`

DESCRIPTION

This command provides functions to control the GRIOMMU core. If more than one core exists in the system, then the index of the core to control should be specified after the **iommu** command (before the subcommand). The 'info sys' command lists the controller indexes.

iommu apv allow *base start stop ?iommu#?*

Modify existing APV at *base* allowing access to the address range *start - stop*

iommu apv build *base prot ?iommu#?*

Create APV starting at *base* with default bit value *prot*

iommu apv decode *base ?iommu#?*

Decode APV starting at *base*

iommu apv deny *base start stop ?iommu#?*

Modify existing APV at *base* denying access to the address range *start - stop*

iommu cache addr *addr grp ?iommu#?*

Displays cached information for I/O address *addr* in group *grp*

iommu cache errinj *addr dt ?byte? ?iommu#?*

Inject data/tag parity error at set address *addr*, data byte *byte*. The parameter *dt* should be either 'tag' or 'data'

iommu cache flush *?iommu#?*

Invalidate all entries in cache

iommu cache show *line ?count? ?iommu#?*

Shows information about *count* line starting at *line*

iommu cache write *addr data0 ... dataN tag*

Write full cache line including tag at set address *addr*, i.e. the number of data words depends on the size of the cache line. See example below.

iommu disable *?iommu#?*

iommu enable *?iommu#?*

Disables/enables the core

iommu group *?grp? ?base passthrough active? ?iommu#?*

Show/set information about group(s). When no parameters are given, information about all groups will be shown. If the index *grp* is given then only that group will be shown. When all parameters are set, the fields will be assigned to the group.

iommu info *?iommu#?*

Displays information about IOMMU configuration

iommu mstbmap *?mst? ?grp? ?iommu#?*

Show/set information about master->group assignments. When no parameters are given, information about all masters will be shown. If the index *mst* is given then only that master will be shown. When all parameters are set, master *mst* will be assigned to group *grp*

iommu mstbmap *?mst? ?ahb? ?iommu#?*

Show/set information about master->AHB interface assignments. When no parameters are given, information about all masters will be shown. If the index *mst* is given then only that master will be shown. When all parameters are set, master *mst* will be assigned to AHB interface *ahb*

iommu pagetable build *base writeable valid ?iommu#?*

Create page table starting at *base* with all writable fields set to *writeable* and all valid fields set to *valid*. 1:1 map starting at physical address 0.

iommu pagetable lookup *base ioaddr ?iommu#?*

Lookup specified IO address in page table starting at *base*.

iommu pagetable modify *base ioaddr phyaddr writeable valid ?iommu#?*

Modify existing PT at *base*, translate *ioaddr* to *phyaddr*, *writeable*, *valid*

iommu status *?iommu#?*

Displays core status information

RETURN VALUE

Upon successful completion **iommu apv decode** returns a list of triples, where each triple contains start, stop and protection bit.

Command **iommu cache addr** returns a tuple, containing valid and protection bits.

Command **iommu cache show** returns a list of entries. Each entry contains line address, tag and the cached data words.

The other subcommands have no return value.

EXAMPLE

Show info on a system with one core

```
grmon3> iommu info
```

Show info of the second core in a system with multiple cores

```
grmon3> iommu info iommu1
```

Writes set address 0x23 with the 128-bit cache line 0x000000008F000000FFFFFFFF00000000 and tag 0x1 (valid line)

```
grmon3> iommu cache write 0x23 0x0 0x8F000000 0xFFFFFFFF 0x0 0x1
```

54. irq - syntax

NAME

irq - Force interrupts or read IRQ(A)MP status information

SYNOPSIS

`irq subcommand args...`

DESCRIPTION

This command provides functions to force interrupts and reading IRQMP status information. The command also support the ASMP extension provided in the IRQ(A)MP core. For IRQAMP with several internal controllers, the index of the internal controller to operate on can be added to the device name on the format `dev::index` at the end of the command.

irq boot *?options? ?mask? ?dev?*

Start the CPUs that are connected to core from the entry point. Add option `-addr address` to set address to start from. A mask can optionally be added to select which of the connected CPUs to start.

irq force *irq ?dev?*

Force interrupt *irq*

irq reg *?dev?*

Display some of the core registers

irq routing *?dev?*

Decode controller routing (for IRQ(A)MP)

irq tstamp *?dev?*

Show time stamp registers (for IRQ(A)MP)

irq wdog *?dev?*

Decode Watchdog control register (for IRQ(A)MP)

RETURN VALUE

Command **irq** has no return value.

55. l2cache - syntax

NAME

l2cache - L2 cache control

SYNOPSIS

l2cache *subcommand* *?args?*

DESCRIPTION

l2cache lookup *addr*

Prints the data and status of a cache line if *addr* generates a cache hit.

l2cache show data *?way? ?count? ?start?*

Prints the data of *count* cache line starting at cache line *start*.

l2cache show tag *?count? ?start?*

Prints the tag of *count* cache line starting at cache line *start*.

l2cache enable

Enable the cache.

l2cache disable

l2cache disable flushinvalidate

Disable the cache. If *flushinvalidate* is given, all dirty cache lines are invalidated and written back to memory as an atomic operation.

l2cache ft *?boolean?*

Enable or disable the EDAC. If *boolean* is not set, then the command will show if the EDAC is enabled or disabled.

l2cache flush

l2cache flush all *?mode?*

Perform a cache flush to all cache lines using a flush *mode*.

l2cache flush mem *address ?mode?*

Perform a cache flush to the cache lines with a cache hit for *address* using a flush *mode*.

l2cache flush direct *address ?mode?*

Perform a cache flush to the cache lines addressed with *address* using a flush *mode*.

l2cache invalidate

Invalidate all cache lines

l2cache flushinvalidate

Flush and invalidate all cache lines (copy-back)

l2cache hit

Prints the hit rate statistics.

l2cache wt *?boolean?*

Enable or disable the write-through. If *boolean* is not set, then the command will show if write-through is enabled or disabled.

l2cache hprot *?boolean?*

Enable or disable the HPROT. If *boolean* is not set, then the command will show if HPROT is enabled or disabled.

l2cache smode *?mode?*

Set the statistics mode. If the *mode* is not set, then the command will show the current statistics mode.

l2cache error

l2cache error inject

l2cache error reset

l2cache error dcb *?value?*

l2cache error tcb *?value?*

The **l2cache error** used to show information about an error in the L2-cache and the information is cleared with **l2cache error reset**. I.a. the **l2cache error inject** can be used to create an error. The **l2cache error dcb** and **l2cache error tcb** can be used to read or write the data/tag check bits.

l2cache mtrr *?index? ?value?*

Show all or a specific memory type range register. If value is present, then the specified register will be set.

l2cache split *boolean*

Enable or disable AHB SPLIT response support for the L2 cache controller.

RETURN VALUE

Upon successful completion **l2cache lookup** returns a list of addr, way, tag, index, offset, valid bit, dirty bit and LRU bit.

Commands **l2cache show data** and **l2cache show tags** returns a list of entries. For **data** each entry contains an address and 8 data words. The entry for **tag** contains index, address, LRU and list of valid bit, dirty bit and tag for each way.

Upon successful completion **l2cache ft**, **l2cache hprot**, **l2cache smode** and **l2cache wt** returns a boolean.

Command **l2cache hit** returns hit-rate and front bus usage-rate.

Command **l2cache status** returns control and status register values.

Upon successful completion **l2cache dcb** and **l2cache tcb** return check bits for data or tags.

Command **l2cache mtrr** returns a list of values.

SEE ALSO

Section 3.4.16, “CPU cache support”

56. l3stat - syntax

NAME

l3stat - Control Leon3 statistics unit

SYNOPSIS

l3stat *subcommand* *?args...?* *?l3stat#?*

DESCRIPTION

This command provides functions to control the L3STAT core. If more than one core exists in the system, then the index of the core to control should be specified after the **l3stat** command (before the subcommand). The 'info sys' command lists the device indexes.

l3stat events *?l3stat#?*

Show all events that can be selected/counted

l3stat status *?l3stat#?*

Display status of all available counters.

l3stat clear *cnt* *?l3stat#?*

Clear the counter *cnt*.

l3stat set *cnt* *cpu* *event* *?enable?* *?clearonread?* *?l3stat#?*

Count the *event* using counter *cnt* on processor *cpu*. The optional *enable* parameter defaults to 1 if left out. The optional *clearonread* parameter defaults to 0 if left out.

l3stat duration *cnt* *enable* *?lvl?* *?l3stat#?*

Enable the counter *cnt* to save maximum time the selected event has been at *lvl*. When enabling the *lvl* parameter must be present, but when disabling it be left out.

l3stat poll *start* *stop* *interval* *hold* *?l3stat#?*

Continuously poll counters between *start* and *stop*. The *interval* parameter sets how many seconds between each iteration. If *hold* is set to 1, then it will block until the first counter is enabled by other means (i.e. software). The polling stops when the first counter is disabled or a SIGINT signal (Ctrl-C) is sent to GRMON.

l3stat runpoll *start* *stop* *interval* *?l3stat#?*

Setup counters between *start* and *stop* to be polled while running an application (i.e. 'run', 'go' or 'cont' commands). The *interval* argument in this case does not specify the poll interval seconds but rather in terms of iterations when GRMON polls the Debug Support Unit to monitor execution. A suitable value for the *int* argument in this case depends on the speed of the host computer, debug link and target system.

EXAMPLE

Enable maximum time count, on counter 1, when no instruction cache misses has occurred.

```
grmon3> l3stat set 1 0 icmiss
grmon3> l3stat duration 1 1 0
```

Disable maximum time count on counter 1.

```
grmon3> l3stat duration 1 0
```

Poll for cache misses when running.

```
grmon3> l3stat set 0 0 dcmiss
grmon3> l3stat set 1 0 icmiss
grmon3> l3stat runpoll 0 1 5000
grmon3> run
```

57. l4stat - syntax

NAME

l4stat - Control Leon4 statistics unit

SYNOPSIS

l4stat *subcommand* *?args...?* *?l4stat#?*

DESCRIPTION

This command provides functions to control the L4STAT core. If more than one core exists in the system, then the index of the core to control should be specified after the **l4stat** command (before the subcommand). The 'info sys' command lists the device indexes.

l4stat events *?l4stat#?*

Show all events that can be selected/counted

l4stat status *?l4stat#?*

Display status of all available counters.

l4stat clear *cnt* *?l4stat#?*

Clear the counter *cnt*.

l4stat set *cnt* *cpu* *event* *?enable?* *?clearonread?* *?l4stat#?*

Count the *event* using counter *cnt* on processor *cpu*. The optional *enable* parameter defaults to 1 if left out. The optional *clearonread* parameter defaults to 0 if left out.

l4stat duration *cnt* *enable* *?lvl?* *?l4stat#?*

Enable the counter *cnt* to save maximum time the selected event has been at *lvl*. When enabling the *lvl* parameter must be present, but when disabling it be left out.

l4stat poll *start* *stop* *interval* *hold* *?l4stat#?*

Continuously poll counters between *start* and *stop*. The *interval* parameter sets how many seconds between each iteration. If *hold* is set to 1, then it will block until the first counter is enabled by other means (i.e. software). The polling stops when the first counter is disabled or a SIGINT signal (Ctrl-C) is sent to GRMON.

l4stat runpoll *start* *stop* *interval* *?l4stat#?*

Setup counters between *start* and *stop* to be polled while running an application (i.e. 'run', 'go' or 'cont' commands). The *interval* argument in this case does not specify the poll interval seconds but rather in terms of iterations when GRMON polls the Debug Support Unit to monitor execution. A suitable value for the *int* argument in this case depends on the speed of the host computer, debug link and target system.

EXAMPLE

Enable maximum time count, on counter 1, when no instruction cache misses has occurred.

```
grmon3> l4stat set 1 0 icmiss
grmon3> l4stat duration 1 1 0
```

Disable maximum time count on counter 1.

```
grmon3> l4stat duration 1 0
```

Poll for cache misses when running.

```
grmon3> l4stat set 0 0 dcmiss
grmon3> l4stat set 1 0 icmiss
grmon3> l4stat runpoll 0 1 5000
grmon3> run
```

58. la - syntax

NAME

la - Control the LOGAN core

SYNOPSIS

la

la *subcommand ?args...?*

DESCRIPTION

The LOGAN debug driver contains commands to control the LOGAN on-chip logic analyzer core. It allows to set various triggering conditions, and to generate VCD waveform files from trace buffer data. All logic analyzer commands are prefixed with la.

If more than one device exists in the system, the *logan#* can be used to select device, default is logan0.

la

la status *?logan#?*

Reports status of LOGAN.

la arm *?logan#?*

Arms the LOGAN. Begins the operation of the analyzer and sampling starts.

la config *filename ?logan#?*

la config *?name bits...? ?logan#?*

Set the configuration of the LOGAN device. Either a filename or an array of name and bits pairs.

la count *?value? ?logan#?*

Set/displays the trigger counter. The *value* should be between zero and depth-1 and specifies how many samples that should be taken after the triggering event.

la div *?value? ?logan#?*

Sets/displays the sample frequency divider register. If you specify e.g. "la div 5" the logic analyzer will only sample a value every 5th clock cycle.

la dump *?filename? ?logan#?*

This dumps the trace buffer in VCD format to the file specified (default is logan.vcd).

la mask *trigl bit ?value? ?logan#?*

Sets/displays the specified bit in the mask of the specified trig level to 0/1.

la page *?value? ?logan#?*

Sets/prints the page register of the LOGAN. Normally the user doesn't have to be concerned with this because dump and view sets the page automatically. Only useful if accessing the trace buffer manually via the GRMON mem command.

la pat *trigl bit ?value? ?logan#?*

Sets/displays the specified bit in the pattern of the specified trig level to 0/1.

la pm *?trigl? ?pattern mask? ?logan#?*

Sets/displays the complete pattern and mask of the specified trig level. If not fully specified the input is zero-padded from the left. Decimal notation only possible for widths less than or equal to 64 bits.

la qual *?bit value? ?logan#?*

Sets/displays which bit in the sampled pattern that will be used as qualifier and what value it shall have for a sample to be stored.

la reset *?logan#?*

Stop the operation of the LOGAN. Logic Analyzer returns to idle state.

la trigctrl *?trigl? ?count cond? ?logan#?*

Sets/displays the match counter and the trigger condition (1 = trig on equal, 0 = trig on not equal) for the specified trig level.

la view *start stop ?filename? ?logan#?*

Prints the specified range of the trace buffer in list format. If no filename is specified the commands prints to the screen.

SEE ALSO

Section 6.13, “On-chip logic analyzer driver”

59. leon - syntax

NAME

leon - Print leon specific registers

SYNOPSIS

leon

DESCRIPTION

leon

Print leon specific registers

60. load - syntax

NAME

load - Load a file or print filenames of uploaded files.

SYNOPSIS

load *?options...? filename ?address? ?cpu#?*

load *subcommand ?arg?*

DESCRIPTION

The load command may be used to upload a file to the system. It can also be used to list all files that have been loaded. When a file is loaded, GRMON will reset the memory controllers registers first.

To avoid overwriting the image file loaded, one must make sure that DMA is not active to the address range(s) of the image. Drivers can be reset using the **reset** command prior to loading.

load *?options...? filename ?address? ?cpu#?*

The load command may be used to upload the file specified by *filename*. If the *address* argument is present, then binary files will be stored at this address, if left out then they will be placed at the base address of the detected RAM. The *cpu#* argument can be used to specify which CPU it belongs to. The options is specified below.

load clear *?cpu#?*

This command will clear the information about the files that have been loaded to the CPU:s. If the *cpu#* argument is specified, then only that CPU will be listed.

load show *?cpu#?*

This command will list which files that have been loaded to the CPU:s. If the *cpu#* argument is specified, then only that CPU will be listed.

OPTIONS

-binary

The -binary option can be used to force GRMON to interpret the file as a binary file.

-bsize bytes

The -bsize option may be used to specify the size blocks of data in bytes that will be written. Sizes that are not even words may require a JTAG based debug link to work properly. See Chapter 5, *Debug link* more information.

-data

Only load sections containing data, i.e. skip instructions.

-delay ms

The -delay option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 4 bytes, but can be changed using the -bsize option.

-nmcr

If the -nmcr (No Memory Controller Reinitialize) option is given then the memory controller(s) are not reinitialized. Without the option set all memory controllers that data is loaded to are reinitialized.

-wprot

If the -wprot option is given then write protection on the core will be disabled

RETURN VALUE

Command **load** returns the entry point.

EXAMPLE

Load and then verify a hello_world application

```
grmon3> load ../hello_world/hello_world  
grmon3> verify ../hello_world/hello_world
```

SEE ALSO

Section 3.4.2, “Uploading application and data to target memory”

61. mcfg1 - syntax

mcfg1 - Show or set reset value of the memory controller register 1

SYNOPSIS

mcfg1 *?value?*

DESCRIPTION

mcfg1 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

SEE ALSO

Section 6.14, “Memory controllers ”

62. mcfg2 - syntax

mcfg2 - Show or set reset value of the memory controller register 2

SYNOPSIS

mcfg2 *?value?*

DESCRIPTION

mcfg2 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

SEE ALSO

Section 6.14, “Memory controllers ”

63. mcfg3 - syntax

mcfg3 - Show or set reset value of the memory controller register 3

SYNOPSIS

mcfg3 *?value?*

DESCRIPTION

mcfg3 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

SEE ALSO

Section 6.14, “Memory controllers ”

64. mdio - syntax

NAME

mdio - Show PHY registers

SYNOPSIS

mdio *paddr raddr ?greth#?*

mdio info *?greth#? ?paddr?*

mdio reg *?greth#? ?paddr?*

DESCRIPTION

mdio *paddr raddr ?greth#?*

Show value of PHY address *paddr* and register *raddr*. If more than one device exists in the system, the *greth#* can be used to select device, default is dev0. The command tries to disable the EDCL duplex detection if enabled.

mdio info *?greth#? ?paddr?*

Show PHY model and link state for each PHY accessible from each GRETH device. Use *greth#* and/or *paddr* to only show link state for a specific GRETH device or PHY.

mdio reg *?greth#? ?paddr?*

Show all registers for each PHY accessible from each GRETH device. Use *greth#* and/or *paddr* to only show registers for a specific GRETH device or PHY.

SEE ALSO

Section 6.4, "Ethernet controller"

65. memb - syntax

NAME

memb - AMBA bus 8-bit memory read access, list a range of addresses

SYNOPSIS

```
memb ?options? address ?length?
```

DESCRIPTION

```
memb ?options? address ?length?
```

Do an AMBA bus 8-bit read access at *address* and print the the data. The optional length parameter should be specified in bytes and the default size is 64 bytes.

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read and then parse out the unaligned data.

OPTIONS

`-bsize bytes`

The `-bsize` option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 1 bytes, but can be changed using the `-bsize` option.

`-ascii`

If the `-ascii` flag has been given, then a single ASCII string is returned instead of a list of values.

`-cstr`

If the `-cstr` flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

`-dec`

Give the `-dec` flag to make the Tcl return signed decimal values instead of hexadecimal strings.

`-asi asi`

Read from SPARC alternate space.

RETURN VALUE

Upon successful completion **memb** returns a list of the requested 8-bit words. Some options changes the result value, see options for more information.

EXAMPLE

Read 4 bytes from address 0x40000000:

```
grmon3> memb 0x40000000 4
```

TCL returns:

```
64 0 0 0
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

66. memd - syntax

NAME

memd - AMBA bus 64-bit memory read access, list a range of addresses

SYNOPSIS

`memd ?options? address ?length?`

DESCRIPTION

`memd ?options? address ?length?`

Do an AMBA bus read access at *address* and print the data as 64-bit words. The optional length parameter should be specified in bytes and the default size is 64 bytes (8 64-bit words).

OPTIONS

`-bsize bytes`

The `-bsize` option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 8 bytes, but can be changed using the `-bsize` option.

`-ascii`

If the `-ascii` flag has been given, then a single ASCII string is returned instead of a list of values.

`-cstr`

If the `-cstr` flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

`-dec`

Give the `-dec` flag to make the Tcl return signed decimal values instead of hexadecimal strings.

`-asi asi`

Read from SPARC alternate space.

RETURN VALUE

Upon successful completion **memd** returns a list of the requested 64-bit words. Some options changes the result value, see options for more information.

EXAMPLE

Read 2 64-bit words (16 bytes) from address 0x40000000:

```
grmon3> memd 0x40000000 16
```

TCL returns:

```
0xffff1244901022 0x543348
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

67. memh - syntax

NAME

memh - AMBA bus 16-bit memory read access, list a range of addresses

SYNOPSIS

memh *?options? address ?length?*

DESCRIPTION

memh *?options? address ?length?*

Do an AMBA bus 16-bit read access at *address* and print the the data. The optional length parameter should specified in bytes and the default size is 64bytes (32 words).

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read and then parse out the unaligned data.

OPTIONS

-bsize bytes

The **-bsize** option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

-delay ms

The **-delay** option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 2 bytes, but can be changed using the **-bsize** option.

-ascii

If the **-ascii** flag has been given, then a single ASCII string is returned instead of a list of values.

-cstr

If the **-cstr** flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

-dec

Give the **-dec** flag to make the Tcl return signed decimal values instead of hexadecimal strings.

-asi asi

Read from SPARC alternate space.

RETURN VALUE

Upon successful completion **memh** returns a list of the requested 16-bit words. Some options changes the result value, see options for more information.

EXAMPLE

Read 4 words (8 bytes) from address 0x40000000:

```
grmon3> memh 0x40000000 8
```

TCL returns:

```
16384 0 0 0
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

68. mem - syntax

NAME

mem - AMBA bus 32-bit memory read access, list a range of addresses

SYNOPSIS

mem *?-options? address ?length?*

DESCRIPTION

mem *?-options? address ?length?*

Do an AMBA bus 32-bit read access at *address* and print the the data. The optional length parameter should be specified in bytes and the default size is 64 bytes (16 words).

OPTIONS

-bsize bytes

The *-bsize* option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

-delay ms

The *-delay* option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 4 bytes, but can be changed using the *-bsize* option.

-ascii

If the *-ascii* flag has been given, then a single ASCII string is returned instead of a list of values.

-cstr

If the *-cstr* flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

-dec

Give the *-dec* flag to make the Tcl return signed decimal values instead of hexadecimal strings.

-asi asi

Read from SPARC alternate space.

RETURN VALUE

Upon successful completion **mem** returns a list of the requested 32-bit words. Some options changes the result value, see options for more information.

EXAMPLE

Read 4 words from address 0x40000000:

```
grmon3> mem 0x40000000 16
```

TCL returns:

```
1073741824 0 0 0
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

69. mil - syntax

mil - MIL-STD-1553B Interface commands

SYNOPSIS

mil *?subcommand?* *?args...?*

DESCRIPTION

mil active *bus device*

Select which device to control and which bus to use for **mil put** and **mil get**.

mil status

Display core status

mil bcx *addr ?count?*

Print BC descriptor contents and result values

mil bmx *addr ?count?*

Print BM log entries from the given memory address

mil bmlog *?count? ?logaddr?*

Print the latest entries from the currently running BM log

mil buf *?bufaddr? ?coreaddr?*

Set address of temporary buffer for transfer commands

mil bufmode *?mode?*

Select if the temporary buffer should be kept or restored. Valid *mode*-values are 'keep' or 'restore'

mil get *rtaddr subaddr count*

Perform an RT-to-BC transfer and display the result

mil getm *rtaddr subaddr count memaddr*

Perform an RT-to-BC transfer and store resulting data at *memaddr*

mil put *rtaddr subaddr count word0 ?... word31?*

Perform an BC-to-RT transfer

mil putm *rtaddr subaddr count memaddr*

Perform an BC-to-RT transfer of data located at *memaddr*

mil halt

Stop the core and store the state for resuming later.

mil resume

Resume operation with state stored earlier by the **mil halt** command.

mil lbtest rt

mil lbtest bc

Runs RT- or BC-part of loopback test

70. mmu - syntax

NAME

mmu - Translate virtual addresses

SYNOPSIS

mmu *?cpu#?*
mmu subcommand *?options...? ?args...? ?cpu#?*

DESCRIPTION

mmu *?cpu#?*

Print the MMU registers

mmu mctrl *?value? ?cpu#?*

Set the MMU control register (LEON only)

mmu ctxptr *?value? ?cpu#?*

Set the context pointer register (LEON only)

mmu ctx *?value? ?cpu#?*

Set the context register (LEON only)

mmu va *?options...? ?ctx? ?cpu#?*

Translate a virtual address. The command will use the MMU from the current active CPU and the *cpu#* can be used to select a different CPU.

mmu walk *?options...? ?ctx? ?cpu#?*

Translate a virtual address and print translation. The command will use the MMU from the current active CPU and the *cpu#* can be used to select a different CPU.

mmu table *?options...? ?ctx? ?cpu#?*

Print table, optionally specify context. The command will use the MMU from the current active CPU and the *cpu#* can be used to select a different CPU.

OPTIONS

-v

The -v will cause the subcommand **walk** to show the intermediate steps when doing a hypervisor two-stage translation. When issuing the **va** subcommand it will show a walk output.

-s

The subcommands **walk**, **va** and **table** will operate using the supervisor MMU table.

-h

The subcommands **walk**, **va** and **table** will operate using the hypervisor two-stage MMU tables.

-hg

The subcommands **walk**, **va** and **table** will operate using the hypervisor guest MMU table.

-vs

The subcommands **walk** and **va** will return the first stage address. First stage PTEs will still be translated using the second stage. The subcommand **table** will show the virtual supervisor MMU table.

RETURN VALUE

The commands **mmu** returns a list of the MMU registers.

The commands **mmu va** and **mmu walk** returns the translated address.

The command **mmu table** returns a list of ranges, where each range has the following format: `{vaddr_start vaddr_end paddr_start paddr_end access pages}`

EXAMPLE

Print MMU registers

```
grmon3> mmu
mctrl: 00904001 ctx: 00000001 ctxptr: 00622000 fsr: 000002DC far: 9CFB9000
```

TCL returns:

```
9453569 1 401920 732 -1661235200
```

Print MMU table

```
grmon3> puts [mmu table]
MMU Table for CTX1 for CPU0
0x00000000-0x00000fff -> 0x00000000-0x00000fff crwxrwx [1 page]
0x00001000-0x0061ffff -> 0x00001000-0x0061ffff crwx--- [1567 pages]
0x00620000-0x00620fff -> 0x00620000-0x00620fff -r-xr-x [1 page]
0x00621000-0x00621fff -> 0x00621000-0x00621fff crwx--- [1 page]
...
```

TCL returns:

```
{0x00000000 0x00000fff 0x00000000 0x00000fff crwxrwx 1} {0x00001000
0x0061ffff 0x00001000 0x0061ffff crwx--- 1567} {0x00620000 0x00620fff
0x00620000 0x00620fff -r-xr-x 1} {0x00621000 0x00621fff 0x00621000 0x00621fff
crwx--- 1} ...
```

SEE ALSO

Section 3.4.15, “Memory Management Unit (MMU) support”

71. nolog - syntax

NAME

nolog - Suppress logging of stdout of a command

SYNOPSIS

nolog *command* *?args...?*

DESCRIPTION

nolog *command* *?args...?*

The nolog command be put in front of other GRMON commands to suppress the logging of the output. This can be useful to remove unnecessary output when scripting.

EXAMPLE

Suppress the memory print.

```
grmon3>nolog mem 0x40000000
```

72. pci - syntax

NAME

pci - Control the PCI bus master

SYNOPSIS

pci subcommand *?args...?*

DESCRIPTION

The PCI debug drivers are mainly useful for PCI host systems. The **pci init** command initializes the host's target BAR1 to point to RAM (PCI address 0x40000000 -> AHB address 0x4000000) and enables PCI memory space and bus mastering. Commands are provided for initializing the bus, scanning the bus, configuring the found resources, disabling byte twisting and displaying information. Note that on non-host systems only the info command has any effect.

The **pci scan** command can be used to print the current configuration of the PCI bus. If a OS has initialized the PCI core and the PCI bus (at least enumerated all PCI buses) the scan utility can be used to see how the OS has configured the PCI address space. Note that scanning a multi-bus system that has not been enumerated will fail.

The **pci conf** command can fail to configure all found devices if the PCI address space addressable by the host controller is smaller than the amount of memory needed by the devices.

A configured PCI system can be registered into the GRMON device handling system similar to the on-chip AMBA bus devices, controlled using the **pci bus** commands. GRMON will hold a copy of the PCI configuration in memory until a new **pci conf**, **pci bus unreg** or **pci scan** is issued. The user is responsible for updating GRMON's PCI configuration if the configuration is updated in hardware. The devices can be inspected from **info sys** and Tcl variables making read and writing PCI devices configuration space easier. The Tcl variables are named in a similar fashion to AMBA devices, for example **puts \$pdev0::status** prints the STATUS register of PCI device0. See **pci bus** reference description below and the Tcl API description in the manual.

pci bt *?boolean?*

Enable/Disable the byte twisting (if supported by host controller)

pci bus reg

Register a previously configured PCI bus into the GRMON device handling system. If the PCI bus has not been configured previously the **pci conf** is automatically called first (similar to **pci conf -reg**).

pci bus unreg

Unregister (remove) a previously registered PCI bus from the GRMON device handling system.

pci cfg8 *deviceid offset*

pci cfg16 *deviceid offset*

pci cfg32 *deviceid offset*

Read a 8-, 16- or 32-bit value from configuration space. The device ID selects which PCI device/function is address during the configuration access. The offset must be located with the device's space and be aligned to access type. Three formats are allowed to specify the *deviceid*: 1. *bus:slot:func*, 2. device name (pdev#), 3. host. It's allowed to skip the bus index, i.e. only specifying *slot:func*, it will then default to bus index 0. The ID numbers are specified in hex. If "host" is given the Host Bridge Controller itself will be queried (if supported by Host Bridge). A device name (for example "pdev0") may also be used to identify a device found from the **info sys** command output.

pci conf *?-reg?*

Enumerate all PCI buses, configures the BARs of all devices and enables PCI-PCI bridges where needed. If -reg is given the configured PCI bus is registered into GRMON device handling system similar to **pci bus reg**, see above.

pci init

Initializes the host controller as described above

pci info

Displays information about the host controller

pci io8 *addr value*

pci io16 *addr value*

pci io32 *addr value*

Write a 8-, 16- or 32-bit value to I/O space.

pci scan *?-reg?*

Scans all PCI slots for available devices and their current configuration are printed on the terminal. The scan does not alter the values, however during probing some registers modified by rewritten with the original value. This command is typically used to look at the reset values (after pci init is called) or for inspecting how the Operating System has set PCI up (pci init not needed). Note that PCI buses are not enumerated during scanning, in multi-bus systems secondary buses may therefore not be accessible. If -reg is given the configured PCI bus is registered into GRMON device handling system similar to **pci bus reg**, see above.

pci wcfg8 *deviceid offset value*

pci wcfg16 *deviceid offset value*

pci wcfg32 *deviceid offset value*

Write a 8-, 16- or 32-bit value to configuration space. The device ID selects which PCI device/function is address during the configuration access. The offset must be located with the device's space and be aligned to access type. Three formats are allowed to specify the *deviceid*: 1. *bus:slot:func*, 2. device name (pdev#), 3. host. It's allowed to skip the bus index, i.e. only specifying *slot:func*, it will then default to bus index 0. The ID numbers are specified in hex. If "host" is given the Host Bridge Controller itself will be queried (if supported by Host Bridge). A device name (for example "pdev0") may also be used to identify a device found from the **info sys** command output.

pci wio8 *addr value*

pci wio16 *addr value*

pci wio32 *addr value*

Write a 8-, 16- or 32-bit value to I/O space.

*PCI Trace commands:***pci trace**

Reports current trace buffer settings and status

pci trace address *pattern*

Get/set the address pattern register.

pci trace amask *pattern*

Get/set the address mask register.

pci trace arm

Arms the trace buffer and starts sampling.

pci trace log *?length? ?offset?*

Prints the trace buffer data. Offset is relative the trigger point.

pci trace sig *pattern*

Get/set the signal pattern register.

pci trace smask *pattern*

Get/set the signal mask register.

pci trace start

Arms the trace buffer and starts sampling.

pci trace state

Prints the state of the PCI bus.

pci trace stop

Stops the trace buffer sampling.

pci trace tcount *value*

Get/set the number of matching trigger patterns before disarm

pci trace tdelay *value*

Get/set number of extra cycles to sample after disarm.

RETURN VALUE

Upon successful completion most **pci** commands have no return value.

The read commands return the read value. The write commands have no return value.

When the commands **pci trace address**, **pci trace amask**, **pci trace sig**, **pci trace smask**, **pci trace tcount** and **pci trace tdelay** are used to read values, they return their values.

The **pci trace log** command returns a list of triples, where the triple contains the address, a list of signals and buffer index.

Command **pci trace state** returns a tuple of the address and a list of signals.

EXAMPLE

Initialize host controller and configure the PCI bus

```
grmon3> pci init  
grmon3> pci conf
```

Inspect a PCI bus that has already been setup

```
grmon3> pci scan
```

SEE ALSO

Section 6.17, “PCI”

73. perf - syntax

perf - Measure performance

SYNOPSIS

perf

perf *?subcommand? ?args...?*

DESCRIPTION

The performance command is only available when a DSU4 exists in the system.

perf

Display result

perf *?disable?*

perf *?enable?*

Enable or disable the performance measure.

74. phyaddr - syntax

NAME

phyaddr - Set the default PHY address

SYNOPSIS

phyaddr *address* *?greth#?*

DESCRIPTION

phyaddr *address* *?greth#?*

Set the default PHY address to *address*. If more than one device exists in the system, the *greth#* can be used to select device, default is greth0.

EXAMPLE

Set PHY address to 1
grmon3> phyaddr 1

SEE ALSO

Section 6.4, “Ethernet controller”

75. profile - syntax

NAME

profile - Enable, disable or show simple profiling

SYNOPSIS

profile *?nlines?* *?cpu#?*

profile clear *?cpu#?*

profile enable *?cpu#?*

profile disable *?cpu#?*

DESCRIPTION

If profiling is enabled then GRMON will profile the application being executed on the system.

profile *?nlines?*

Show profiling information for all CPUs or specified CPU. When printing the information for all the CPUs, only a single table with the sum of all CPUs will be printed. Optionally you can limit the number of printed lines with the a *nlines* argument.

profile clear

Clear collected information on all CPUs or specified CPU.

profile enable

Turn on profiling all CPUs or a single CPU.

profile disable

Turn off profiling for all CPUs or a single CPU.

SEE ALSO

Section 3.4.10, “Profiling”

76. quit - syntax

NAME

quit - Shut down the GRMON application

SYNOPSIS

quit

DESCRIPTION

quit

Shut down the GRMON application. GRMON will set 0 as exit code if no internal error has occurred or a value greater or equal to 1 to indicate an internal error. This command is the same as 'exit'.

RETURN VALUE

Command **quit** has no return value.

EXAMPLE

Shut down the GRMON application.

```
grmon3> quit
```

77. reg - syntax

reg - Show or set integer registers

SYNOPSIS

reg *?name ...? ?name value ...?*

DESCRIPTION

reg *?name ...? ?name value ...? ?cpu#?*

Show or set integer registers of the current CPU, or the CPU specified by *cpu#*. If no register arguments are given then the command will print the current window and the special purpose registers. The register arguments can to both set and show each individual register. If a register name is followed by a value, it will be set else it will only be shown.

Floating-point registers should be set with a decimal point value. A decimal-point value can be suffixed with an *f* character to force the value to be interpreted as a single precision value. If an integer is written to a floating-point register, the value will be interpreted as a binary representation of a floating-point value.

Valid LEON window register names are:

Registers

r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15, r16, r17, r18, r19, r20, r21, r22, r23, r24, r25, r26, r27, r28, r29, r30, r31

Global registers

g0, g1, g2, g3, g4, g5, g6, g7

Current window in registers

i0, i1, i2, i3, i4, i5, i6, i7

Current window local registers

l0, l1, l2, l3, l4, l5, l6, l7

Current window out registers

o0, o1, o2, o3, o4, o5, o6, o7

Special purpose registers

sp, fp

Windows (N is the number of implemented windows)

w0, w1 ... wN

Single register from a window

w1i3 w1o3 w2i5 etc.

In addition the following non-window related LEON registers are also valid:

Floating point registers (native precision)

f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, f21, f22, f23, f24, f25, f26, f27, f28, f29, f30, f31

Virtual floating point registers (single precision)

sf0, sf1, sf2, sf3, sf4, sf5, sf6, sf7, sf8, sf9, sf10, sf11, sf12, sf13, sf14, sf15, sf16, sf17, sf18, sf19, sf20, sf21, sf22, sf23, sf24, sf25, sf26, sf27, sf28, sf29, sf30, sf31

Virtual floating point registers (double precision)

d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14, d15

Special purpose registers

psr, tbr, wim, y, pc, npc, fsr

Application specific registers

asr16, asr17, asr18

Valid NOEL-V register names are:

Registers

x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10, x11, x12, x13, x14, x15, x16, x17, x18, x19, x20, x21, x22, x23, x24, x25, x26, x27, x28, x29, x30, x31

Virtual registers

zero, ra, sp, gp, tp, a0, a1, a2, a3, a4, a5, a6, a7, t0, t1, t2, t3, t4, t5, t6, s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11

CSR registers

csr###, where ### is the hexadecimal register number, or the name of the CSR register

Virtual debug registers

prv

Floating point registers (native precision)

f0, f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, f21, f22, f23, f24, f25, f26, f27, f28, f29, f30, f31

fa0, fa1, fa2, fa3, fa4, fa5, fa6, fa7, ft0, ft1, ft2, ft3, ft4, ft5, ft6, ft7, ft8, ft9, ft10, ft11, fs0, fs1, fs2, fs3, fs4, fs5, fs6, fs7, fs8, fs9, fs10, fs11

Virtual floating point registers (single precision)

sf0, sf1, sf2, sf3, sf4, sf5, sf6, sf7, sf8, sf9, sf10, sf11, sf12, sf13, sf14, sf15, sf16, sf17, sf18, sf19, sf20, sf21, sf22, sf23, sf24, sf25, sf26, sf27, sf28, sf29, sf30, sf31

sfa0, sfa1, sfa2, sfa3, sfa4, sfa5, sfa6, sfa7, sft0, sft1, sft2, sft3, sft4, sft5, sft6, sft7, sft8, sft9, sft10, sft11, sfs0, sfs1, sfs2, sfs3, sfs4, sfs5, sfs6, sfs7, sfs8, sfs9, sfs10, sfs11

Virtual floating point registers (double precision)

d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14, d15, d16, d17, d18, d19, d20, d21, d22, d23, d24, d25, d26, d27, d28, d29, d30, d31

da0, da1, da2, da3, da4, da5, da6, da7, dt0, dt1, dt2, dt3, dt4, dt5, dt6, dt7, dt8, dt9, dt10, dt11, ds0, ds1, ds2, ds3, ds4, ds5, ds6, ds7, ds8, ds9, ds10, ds11

RETURN VALUE

Upon successful completion, command **reg** returns a list of the requested register values. When register windows are requested, then nested list of all registers will be returned. If a float/double is requested, then a tuple of the decimal and the binary value is returned.

EXAMPLE

Display the current window and special purpose registers

```
grmon3> reg
```

TCL returns:

```
{0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0} -213905184
2 1073741824 0 1073741824 1073741828
```

Display the g0, l3 in window 2, f1, pc and w1.

```
grmon3> reg g0 w2l3 f1 pc w1
```

TCL returns:

```
0 0 {0.0 0} 1073741824 {0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0}
0 0 0 0 0 0 0 0 0 0 0}
```

Set register g1 to the value 2 and display register g2

```
grmon3> reg g1 2 g2
```

TCL returns:

```
2 0
```

Set floating point registers

```
grmon3> reg f0 3.141593 f1 3.141593f d1 3.141593 d2 3.141593f
```

TCL returns:

```
{3.1415929794311523 0x40490fdc} {3.1415929794311523 0x40490fdc} {3.141593
0x400921fb82c2bd7f} {3.1415929794311523 0x400921fb80000000}
```

SEE ALSO

Section 3.4.5, “Displaying processor registers”

78. reset - syntax

NAME

reset - Reset drivers

SYNOPSIS

reset

DESCRIPTION

The **reset** will give all core drivers an opportunity to reset themselves into a known state. For example will the memory controllers reset it's registers to their default value and some drivers will turn off DMA. It is in many cases crucial to disable DMA before loading a new binary image since DMA can overwrite the loaded image and destroy the loaded Operating System.

EXAMPLE

Reset drivers
grmon3> reset

79. rtg4fddr - syntax

NAME

rtg4fddr - Print initialization sequence

SYNOPSIS

rtg4fddr *show ?fddr#?*

DESCRIPTION

rtg4fddr *show ?fddr#?*

Print initialization sequence

The RTG4 FDDR initcode is loaded into a procedure in the system shell. The procedure is executed in init level 6, therefore it is possible to override the script in level 5 by redefining the the `::fdir#::init` procedure using the `init#` hook.

EXAMPLE

Override the default initialization

```
proc MyInit5 {} {  
  proc ::fddr0::init {} {  
    # Add custom initialization code here  
  }  
  proc ::fddr1::init {} {  
    # Add custom initialization code here  
  }  
}  
lappend ::hooks::init5 MyInit5
```

SEE ALSO

Section 3, “User defined hooks”

80. rtg4serdes - syntax

NAME

rtg4serdes - Print initialization sequence

SYNOPSIS

```
rtg4serdes show ?serdes#?
```

DESCRIPTION

```
rtg4serdes show ?serdes#?
```

Print initialization sequence

The RTG4 SERDES initcode is loaded into a procedure in the system shell. The procedure is executed in init level 6, therefore it is possible to override the script in level 5 by redefining the the `::serdes#::init` procedure using the `init#` hook.

EXAMPLE

Override the default initialization

```
proc MyInit5 {} {  
  proc ::serdes0::init {} {  
    # Add custom initialization code here  
  }  
}  
lappend ::hooks::init5 MyInit5
```

SEE ALSO

Section 3, “User defined hooks”

81. run - syntax

run - Reset and start execution

SYNOPSIS

`run ?options? ?address? ?count?`

DESCRIPTION

`run ?options? ?address? ?count?`

This command will start the execution of instructions on the active CPU.

When omitting the address parameter this command will start execution at the entry point of the last loaded application.

This command will reset all drivers, unlike the **go** command. (see **reset** for more information)

If the *count* parameter is set then the CPU will run the specified number of instructions. Note that the *count* parameter is only supported by the DSU4.

OPTIONS

-noret

Do not evaluate the return value. When this options is set, no return value will be set.

RETURN VALUE

Upon successful completion **run** returns a list of signals, one per CPU. Possible signal values are SIGBUS, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM or SIGTRAP. If a CPU is disabled, then an empty string will be returned instead of a signal value.

EXAMPLE

Execute instructions starting at the entry point of the last loaded file.

```
grmon3> run
```

SEE ALSO

Section 3.4.3, "Running applications"

reset

82. rviommu - syntax

NAME

rviommu - Control RISC-V IOMMU

SYNOPSIS

rviommu *subcommand* ?args?
rviommu *index subcommand* ?args?

DESCRIPTION

This command provides functions to control the RISC-V IOMMU. If more than one core exists in the system, then the index of the core to control should be specified after the **rviommu** command (before the subcommand). The 'info sys' command lists the controller indexes.

rviommu tr ?options...? *device_id* *iova*

Translate a virtual address using the rviommu translation-request interface. Translations are done by the IOMMU as if it was requested by the *device_id* and the given access options.

Available options are `-priv`, `-exe` and `-nw` to set request attributes. The `-pid` option takes a *process_id* argument.

rviommu ddt show ?-base *addr*? ?-stage1? ?*device_id*?

Show information about device contexts. Device directory table is normally read from the rviommu. If `-base` is given, then a DDT at *addr* in 1LVL format is used instead.

When *device_id* is given, detailed information about that device context is displayed. The `-stage1` option will print the page tables for *device_id*.

rviommu ddt walk ?-base *addr*? *device_id* *iova*

Translate a virtual address and print translation.

If `-base` is given, then a DDT at *addr* in 1LVL format is used.

rviommu ddt build *base*

Create a one-level Device-Directory-Table starting at address *base* with all device contexts set to valid and Bare mode for the first-stage and second-stage page table.

rviommu ddt modify *base device_id mode1 root1 mode2 root2* ?-pdtv *val*?

Set mode and root address of first-stage and second-stage page tables. If any of the fields *mode1*, *root1*, *mode2* and *root2* are -1, then the corresponding field in the device directory table will be kept.

The `-pdtv` option can be used to set the `tc.pdtv` device context field to 0 or 1.

RETURN VALUE

rviommu ddt walk returns the translated address.

rviommu tr returns the translated address.

The other subcommands have no return value.

EXAMPLE

Show info about device contexts using a DDT in memory on address 0x40000000.

```
grmon3> rviommu ddt show -base 0x40000000
Device context for all device_id
device_id  PDTV  STAGE1 (IOVA/GVA -> GPA)  STAGE2 (GPA -> SPA)
device_id  PDTV  MODE  PDT / PAGE TABLE  GPA  MODE  PAGE TABLE  MSI
000000  0  Bare  -  -  -  Bare  -  -
000001  0  Sv39  0000000040001000  N  Bare  -  -
```

```

000002 1 PD8 0000000040005000 N Bare - -
000003 0 Bare - - Bare - -
...

```

Show detailed info about device context 1 and its first-stage page table

```

grmon3> rviommu ddt show -base 0x40000000 1 -stage1
Device context for device_id 0x000001
  TC: 0x0000000000000001 V=1 EN_ATS=0 PDTV=0 DPE=0 SBE=0 SXL=0
 IOHGATP: 0x0000000000000000 PADDR=0x0000000000000000 GSCID=0x00000 MODE=Bare
  TA: 0x0000000000000000 PSCID=0x00000
  FSC: 0x8000000000004001 PADDR=0x0000000040001000 (iosatp) MODE=Sv39
First-stage page table for device_id 0x000001
0x0000000000-0x02ffffffff -> 0x00000000-0x2ffffffff -rw---- [3145728 pages]
0x0300000000-0x03001fffff -> 0x4000c000-0x4020bfff -r----- [512 pages]
0x0300200000-0x03003fffff -> 0x4000c000-0x4020bfff -r----- [512 pages]
0x0300400000-0x03005fffff -> 0x4000c000-0x4020bfff -r----- [512 pages]
0x0300600000-0x03007fffff -> 0x4000c000-0x4020bfff -r----- [512 pages]
0x0300818000-0x0300818fff -> 0x40004000-0x40004fff -rw---- [1 page]
0x0304000000-0x7fffffffff -> 0x340000000-0x7fffffffff -rwx--- [130809856 pages]

```

Create an initial device directory table in memory

```
grmon3> rviommu ddt build 0x40000000
```

Modify the device context for device_id 1 to use Sv39 for the first stage and no second stage table. The DDT is located at 0x40000000.

```
grmon3> rviommu ddt modify 0x40000000 1 8 0x40001000 0 0
```

83. scrub - syntax

scrub - Control memory scrubber

SYNOPSIS

scrub *?subcommand?* *?args...?*

DESCRIPTION

scrub
scrub status

Display status and configuration

scrub ack

Clear error and done status and display status

scrub clear *start stop ?value?*

Set scrubber to clear memory area from address *start* up to *stop*. The parameter *value* defaults to 0.

scrub pattern *word1 ?word2 ...?*

Write pattern words into the scrubbers initialization register. If the number of words specified are larger than the size of the burst length, then the remaining words be ignored. If the number of words are less than the burst length, the pattern will be repeated up to a complete burst.

scrub init *start stop*

Initialize the memory area from address *start* up to *stop*.

scrub rst

Clear status and reset configuration.

EXAMPLE

Write pattern 0 1 to the memory 0x0000000 to 0x0000003F

```
grmon3> scrub pattern 0 1  
grmon3> scrub init 0 63
```

Clear a memory area

```
grmon3> scrub clear 0 63
```

84. sdcfg1 - syntax

sdcfg1 - Show or set reset value of SDRAM controller register 1

SYNOPSIS

sdcfg1 *?value?*

DESCRIPTION

sdcfg1 *?value?*

Set the reset value of the memory register. If value is left out, then the reset value will be printed.

SEE ALSO

Section 6.14, “Memory controllers ”

85. sddel - syntax

sddel - Show or set the SDCLK delay

SYNOPSIS

sddel *?value?*

DESCRIPTION

sddel *?value?*

Set the SDCLK delay value.

SEE ALSO

Section 6.14, “Memory controllers”

86. sf2mddr - syntax

NAME

sf2mddr - Print initialization sequence

SYNOPSIS

sf2mddr *show ?mddr#?*

DESCRIPTION

sf2mddr *show ?mddr#?*

Print initialization sequence

The IGLOO2/SmartFusion2 DDR initcode is loaded into a procedure in the system shell. The procedure is executed in init level 6, therefore it is possible to override the script in level 5 by redefining the the `::mddr#::init` procedure using the `init#` hook.

EXAMPLE

Override the default initialization

```
proc MyInit5 {} {  
  proc ::mddr0::init {} {  
    # Add custom initialization code here  
  }  
}  
lappend ::hooks::init5 MyInit5
```

SEE ALSO

Section 3, “User defined hooks”

87. sf2serdes - syntax

NAME

sf2serdes - Print initialization sequence

SYNOPSIS

`sf2serdes show ?serdes#?`

DESCRIPTION

`sf2serdes show ?serdes#?`

Print initialization sequence

The IGLOO2/SmartFusion2 SERDES initcode is loaded into a procedure in the system shell. The procedure is executed in init level 6, therefore it is possible to override the script in level 5 by redefining the the `::serdes#::init` procedure using the `init#` hook.

EXAMPLE

Override the default initialization

```
proc MyInit5 {} {  
  proc ::serdes0::init {} {  
    # Add custom initialization code here  
  }  
}  
lappend ::hooks::init5 MyInit5
```

SEE ALSO

Section 3, “User defined hooks”

88. shell - syntax

NAME

shell - Execute a shell command

SYNOPSIS

shell

DESCRIPTION

shell

Execute a command in the host system shell. The grmon **shell** command is just an alias for the TCL command `exec`, wrapped with `puts`, i.e. its equivalent to `puts [exec ...]`. For more information see documentation about the `exec` command (<http://www.tcl.tk/man/tcl8.6/TclCmd/exec.htm>).

EXAMPLE

List all files in the current working directory (Linux)

```
grmon3> shell ls
```

List all files in the current working directory (Windows)

```
grmon3> shell dir
```

89. silent - syntax

NAME

silent - Suppress stdout of a command

SYNOPSIS

silent *command* ?*args*...?

DESCRIPTION

silent *command* ?*args*...?

The silent command be put in front of other GRMON commands to suppress their output and it will not be logged. The TCL command puts will still be able to print to stdout and be logged. This can be useful to remove unnecessary output when scripting.

EXAMPLE

Suppress the memory print and print the TCL result instead.

```
grmon3> puts [silent mem 0x40000000]
```

SEE ALSO

Section 2, “Variables”

90. spim - syntax

NAME

spim - Commands for the SPI memory controller

SYNOPSIS

spim *subcommand* *?args...?* *?spim#?*

DESCRIPTION

This command provides functions to control the SPICTRL core. If more than one core exists in the system, then the index of the core to control should be specified after the **spim** command (before the subcommand). The 'info sys' command lists the device indexes.

spim altscaler *?spim#?*

Toggle the usage of alternate scaler to enable or disable.

spim reset *?spim#?*

Core reset

spim status *?spim#?*

Displays core status information

spim tx data *?[rx|tx|data] ...?* *?spim#?*

Shift a byte to the memory device. The *tx* or *rx* keywords can be used to switch between sending and receiving data without toggle user mode. This is useful if the device is in DSPI or QSPI mode.

spim rx data *?[rx|tx|data] ...?* *?spim#?*

Shift a byte from the memory device. You must specify one byte for each byte you want to read, however the values will be ignored. The *tx* or *rx* keywords can be used to switch between sending and receiving data without toggle user mode. This is useful if the device is in DSPI or QSPI mode.

SD Card specific commands:

spim sd csd *?spim#?*

Displays and decodes CSD register

spim sd reinit *?spim#?*

Reinitialize card

SPI Flash commands:

spim flash *?spim#?*

Prints a list of available commands

spim flash detect *?spim#?*

Try to detect type of memory device

spim flash dump *?options...?* *address length ?filename?* *?spim#?*

Dumps *length* bytes, starting at *address* of the SPI-device (i.e. not AMBA address), to a file. The default name of the file is "grmon-spiflash-dump.srec"

The *-binary* option can be used to store data to a binary file

Set the *-append* option to append the dumped data to the end of the file. The default is to truncate the file to zero length before storing the data into the file.

spim flash erase *?spim#?*

spim flash erase *start ?stop?* *?spim#?*

Erase performs a bulk erase clearing the whole device or the blocks from address *start* to address *stop*.

spim flash fast *?spim#?*

Enables or disables FAST READ command (memory device may not support this).

spim flash load *?options...? filename ?address? ?cpu#?*

Loads the contents in the file *filename* to the memory device. If the *address* is present, then binary files will be stored at the *address* of the SPI-device (i.e. not AMBA address), otherwise binary files will be written to the beginning of the device. The *cpu#* argument can be used to specify which CPU it belongs to.

The only available option is '-binary', which forces GRMON to interpret the file as binary file.

spim flash select *?index? ?spim#?*

Select memory device. If *index* is not specified, a list of the supported devices is displayed.

spim flash set *?options...? ?attribute value(s)...? ?spim#?*

Sets a custom memory device configuration or modify the current configuration. If the option *-new* is set, or no current configuration has been set, then it will create a new custom configuration. Otherwise if it modify the current configuration.

The following attributes and values are available

<i>pagesize size</i>	Page size
<i>adrbytes value</i>	Number of bytes in address
<i>wren value</i>	Write enable command
<i>wrdi value</i>	Write disable command
<i>rdsr value</i>	Read status register command
<i>wrsr value</i>	Write status register command
<i>read value</i>	Read data bytes command
<i>fast_read value</i>	Fast read data bytes command
<i>pp value</i>	Page programming command
<i>se value</i>	Sector erase command
<i>be value bytes</i>	Bulk/Die erase command and number of bytes
<i>sectors {count bytes ?se cmd0? ...}</i>	A list of number of sectors and number of bytes per sector. Up to 4 sector groups can be added. They are used with the <i>se</i> command to erase a single sector. Optionally you can also override the sector erase command for a group of sectors, if different sector erase commands are needed. If sector erase is not supported, then add one pair with <i>count = 1</i> and <i>bytes = size of memory</i> .
<i>dummy boolean</i>	Use dummy byte
<i>name string</i>	Name of device

Issue **spim flash show** to see current attribute values.

Status register of the memory must have the following bits defined for the custom device to be supported.

Bit 1 WEL (write enable latch)

Bit 0 WIP (write in progress)

spim flash show *?spim#?*

Shows current memory device configuration

spim flash ssva *?value? ?spim#?*

Sets slave value to be used with the SPICTRL core. When GRMON wants to select the memory device it will write this value to the slave select register. When the device is deselected, GRMON will write all ones to the slave select register. Example: Set slave select line 0 to low, all other lines high when selecting a device

```
grmon3> spi flash ssva 0xffffffff
```

Note: This value is not used when communicating via the SPIMCTRL core, i.e. it is only valid for **spi flash**.
spim flash status *?spim#?*

Displays device specific information

spim flash strict *?boolean? ?spim#?*

Enable/Disable strict communication mode. Enable if programming fails. Strict communication mode may be necessary when using very fast debug links or for SPI implementations with a slow SPI clock

spim flash verify *?options...? filename ?address? ?spim#?*

Verifies that data in the file *filename* matches data in memory device. If the *address* is present, then binary files will be compared with data at the *address* of the SPI-device (i.e. not AMBA address), otherwise binary files will be compared against data at the beginning of the device.

The `-binary` options forces GRMON to interpret the file as binary file.

The `-erase` option to automatically erase the flash before writing. It will only erase the sectors where data will be written.

The `-max` option can be used to force GRMON to stop verifying when num errors have been found.

When the `-errors` option is specified, the verify returns a list of all errors instead of number of errors. Each element of the list is a sublist whose format depends on the first item if the sublist. Possible errors can be detected are memory verify error (MEM), read error (READ) or an unknown error (UNKNOWN). The formats of the sublists are: MEM *address read-value expected-value* , READ *address num-failed-addresses* , UNKNOWN *address*

Upon successful completion **spim flash verify** returns the number of error detected. If the `-errors` has been given, it returns a list of errors instead.

spim flash wrdi *?spim#?*

spim flash wren *?spim#?*

Issue write disable/enable instruction to the device.

SEE ALSO

Section 3.11.4, “SPI memory device”

Section 6.14, “Memory controllers ”

91. spi - syntax

NAME

spi - Commands for the SPI controller

SYNOPSIS

spi *subcommand* *?args...?* *?spi#?*

DESCRIPTION

This command provides functions to control the SPICTRL core. If more than one core exists in the system, then the index of the core to control should be specified after the **spi** command (before the subcommand). The 'info sys' command lists the device indexes.

spi aslvsel *value* *?spi#?*

Set automatic slave select register

spi disable *?spi#?*

spi enable *?spi#?*

Enable/Disable core

spi rx *?spi#?*

Read receive register

spi selftest *?spi#?*

Test core in loop mode

spi set *?field ...?* *?spi#?*

Sets specified field(s) in Mode register.

Available fields: cpol, cpha, div16, len *value*, amen, loop, ms, pm *value*, tw, asel, fact, od, tac, rev, aseldel *value*, tto, igsel, cite

spi slvsel *value* *?spi#?*

Set slave select register

spi status *?spi#?*

Displays core status information

spi tx data *?spi#?*

Writes data to transmit register. GRMON automatically aligns the data

spi unset *?field ...?* *?spi#?*

Sets specified field(s) in Mode register.

Available fields: cpol, cpha, div16, amen, loop, ms, tw, asel, fact, od, tac, rev, tto, igsel, cite

Commands for automated transfers:

spi am cfg *?option ...?* *?spi#?*

Set AM configuration register.

Available fields: seq, strict, ovtb, ovdb

spi am per *value* *?spi#?*

Set AM period register to *value*.

spi am act *?spi#?*

spi am deact *?spi#?*

Start/stop automated transfers.

spi am extact *?spi#?*

Enable external activation of AM transfers

spi am poll *count ?spi#?*

Poll for *count* transfers

SPI Flash commands:

spi flash *?spi#?*

Prints a list of available commands

spi flash detect *?spi#?*

Try to detect type of memory device

spi flash dump *?options...? address length ?filename? ?spi#?*

Dumps *length* bytes, starting at *address* of the SPI-device (i.e. not AMBA address), to a file. The default name of the file is "grmon-spiflash-dump.srec"

The `-binary` option can be used to store data to a binary file

Set the `-append` option to append the dumped data to the end of the file. The default is to truncate the file to zero length before storing the data into the file.

spi flash erase *?spi#?*

spi flash erase *start ?stop? ?spi#?*

Erase performs a bulk erase clearing the whole device or the blocks from address *start* to address *stop*.

spi flash fast *?spi#?*

Enables or disables FAST READ command (memory device may not support this).

spi flash load *?options...? filename ?address? ?cpu#? ?spi#?*

Loads the contents in the file *filename* to the memory device. If the *address* is present, then binary files will be stored at the *address* of the SPI-device (i.e. not AMBA address), otherwise binary files will be written to the beginning of the device. The *cpu#* argument can be used to specify which CPU it belongs to.

The option `-binary` forces GRMON to interpret the file as binary file.

Use the `-erase` option to automatically erase the flash before writing. It will only erase the sectors where data will be written.

spi flash select *?index? ?spi#?*

Select memory device. If *index* is not specified, a list of the supported devices is displayed.

spi flash set *?options...? ?attribute value(s)...? ?spi#?*

Sets a custom memory device configuration or modify the current configuration. If the option `-new` is set, or no current configuration has been set, then it will create a new custom configuration. Otherwise if it modify the current configuration.

The following attributes and values are available

<code>pagesize</code> <i>size</i>	Page size
<code>adrbytes</code> <i>value</i>	Number of bytes in address
<code>wren</code> <i>value</i>	Write enable command
<code>wrdi</code> <i>value</i>	Write disable command
<code>rdsr</code> <i>value</i>	Read status register command
<code>wrsr</code> <i>value</i>	Write status register command
<code>read</code> <i>value</i>	Read data bytes command

<i>fast_read value</i>	Fast read data bytes command
<i>pp value</i>	Page programming command
<i>se value</i>	Sector erase command
<i>be value bytes</i>	Bulk/Die erase command and number of bytes
<i>sectors {count bytes ?se cmd0? ...}</i>	A list of number of sectors and number of bytes per sector. Up to 4 sector groups can be added. They are used with the <i>se</i> command to erase a single sector. Optionally you can also override the sector erase command for a group of sectors, if different sector erase commands are needed. If sector erase is not supported, then add one pair with <i>count = 1</i> and <i>bytes = size of memory</i> .
<i>dummy boolean</i>	Use dummy byte
<i>name string</i>	Name of device

Issue **spi flash show** to see current attribute values.

Status register of the memory must have the following bits defined for the custom device to be supported.

Bit 1 WEL (write enable latch)

Bit 0 WIP (write in progress)

spi flash show *?spi#?*

Shows current memory device configuration

spi flash ssva *?value? ?spi#?*

Sets slave value to be used with the SPICTRL core. When GRMON wants to select the memory device it will write this value to the slave select register. When the device is deselected, GRMON will write all ones to the slave select register. Example: Set slave select line 0 to low, all other lines high when selecting a device

```
grmon3> spi flash ssva 0xfffffffffe
```

Note: This value is not used when communicating via the SPIMCTRL core, i.e. it is only valid for **spi flash**.

spi flash status *?spi#?*

Displays device specific information

spi flash strict *?boolean? ?spi#?*

Enable/Disable strict communication mode. Enable if programming fails. Strict communication mode may be necessary when using very fast debug links or for SPI implementations with a slow SPI clock

spi flash verify *?options...? filename ?address? ?spi#?*

Verifies that data in the file *filename* matches data in memory device. If the *address* is present, then binary files will be compared with data at the *address* of the SPI-device (i.e. not AMBA address), otherwise binary files will be compared against data at the beginning of the device.

The *-binary* option forces GRMON to interpret the file as binary file.

The *-max* option can be used to force GRMON to stop verifying when num errors have been found.

When the *-errors* option is specified, the verify returns a list of all errors instead of number of errors. Each element of the list is a sublist whose format depends on the first item if the sublist. Possible errors can be detected are memory verify error (MEM), read error (READ) or an unknown error (UNKNOWN). The formats of the sublists are: MEM *address read-value expected-value*, READ *address num-failed-addresses*, UNKNOWN *address*

Upon successful completion **spi flash verify** returns the number of error detected. If the *-errors* has been given, it returns a list of errors instead.

spi flash wrdi *?spi#?*

spi flash wren *?spi#?*

Issue write disable/enable instruction to the device.

EXAMPLE

Set AM configuration register

```
grmon3> spi am cfg strict ovdb
```

Set AM period register

```
grmon3> spi am per 1000
```

Poll queue 10 times

```
grmon3> spi am poll 10
```

Set fields in Mode register

```
grmon3> spi set ms cpha len 7 rev
```

Unset fields in Mode register

```
grmon3> spi unset ms cpha rev
```

SEE ALSO

Section 3.11.4, “SPI memory device”

Section 6.14, “Memory controllers ”

92. spwrtr - syntax

NAME

spwrtr - SpaceWire router information

SYNOPSIS

```
spwrtr info ?port? ?spwrtr#?  
spwrtr rt ?options? ?port? ?endport? ?spwrtr#?  
spwrtr rt add ?options? port ?dst...? ?spwrtr#?  
spwrtr rt remove ?options? port ?dst...? ?spwrtr#?
```

DESCRIPTION

spwrtr info ?port? ?spwrtr#?

Print register information for the router or a single port.

spwrtr rt ?options? ?port? ?endport? ?spwrtr#?

Print the routing table. A single port or a range of ports can be specified, otherwise all ports will be printed.

Options `-physical` or `-logical` can be used to filter out ports.

Options `-nh` can be used to suppress the printing of the header.

spwrtr rt add ?options? port ?dst...? ?spwrtr#?

Enable one more destination ports to the routing table.

Options `-en`, `-hd`, `-pr`, `-sr` and `-pd` can be used to set the corresponding bits. If no destination port has been specified, the option flags will still set the corresponding bits.

spwrtr rt remove ?options? port ?dst...? ?spwrtr#?

Disable one more destination ports to the routing table.

Options `-en`, `-hd`, `-pr`, `-sr` and `-pd` can be used to unset the corresponding bits. If no destination port has been specified, the option flags will still unset the corresponding bits.

RETURN VALUE

Command **spwrtr** has no return value.

SEE ALSO

Section 6.20, "SpaceWire router"

93. stack - syntax

NAME

stack - Set or show the initial stack-pointer.

SYNOPSIS

stack *?cpu#?*
stack *address ?cpu#?*

DESCRIPTION

stack *?cpu#?*

Show current active CPUs initial stack-pointer, or the CPU specified by *cpu#*.

stack *address ?cpu#?*

Set the current active CPUs initial stack-pointer, or the CPU specified by *cpu#*.

RETURN VALUE

Upon successful completion **stack** returns a list of initial stack-pointer addresses, one per CPU.

EXAMPLE

Set current active CPUs initial stack-pointer to 0x4FFFFFF0
grmon3> stack 0x4FFFFFF0

SEE ALSO

Section 6.3.1, “Switches”

Section 3.4.13, “Multi-processor support”

94. step - syntax

step - Step one or more instructions

SYNOPSIS

```
step ?nsteps? ?cpu#?
```

DESCRIPTION

```
step ?nsteps? ?cpu#?
```

Step one or more instructions on all CPU:s. If *cpu#* is set, then only the specified CPU index will be stepped.

When single-stepping over a conditional or unconditional branch with the annul bit set, and if the delay instruction is effectively annulled, the delay instruction itself and the instruction thereafter are stepped over in the same go. That means that three instructions are executed by one single step command in this particular case.

EXAMPLE

```
Step 10 instructions  
grmon3> step 10
```

95. stop - syntax

stop - Interrupt current CPU execution

SYNOPSIS

`stop ?-nowait?`

DESCRIPTION

`stop ?-nowait?`

This command will interrupt the CPU execution initiated by another shell or by the graphical user interface. If the CPU is not currently executing the command will be ignored. By default **stop** will block until the CPU execution has stopped and it is safe to access CPU registers immediately after.

If `-nowait` option is given the command will not block until the CPU execution stop request has been completed. Instead the command will return immediately and accessing CPU registers afterwards might result in "CPU not in debug mode" messages a short time while GRMON stops the on-going CPU execution.

EXAMPLE

Block until on-going CPU execution has been interrupted and then read the registers of CPU0 safely.

```
grmon> stop; reg cpu0
```

Attempt to interrupt on-going CPU execution if started by another shell or GUI without blocking:

```
grmon> stop -nowait
```

96. svga - syntax

NAME

svga - Commands for the SVGA controller

SYNOPSIS

svga *subcommand* *?args...?* *?svga#?*

DESCRIPTION

This command provides functions to control the SVGACTRL core. If more than one core exists in the system, then the index of the core to control should be specified after the **svga** command (before the subcommand). The 'info sys' command lists the device indexes.

```
svga custom ?period horizontal_active_video horizontal_front_porch  
horizontal_sync horizontal_back_porch vertical_active_video  
vertical_front_porch vertical_sync vertical_back_porch? ?svga#?
```

The **svga custom** command can be used to specify a custom format. The custom format will have precedence when using the **svga draw** command. If no parameters are given, then it will print the current custom format.

svga draw *file bitdepth* *?svga#?*

The **svga draw** command will determine the resolution of the specified picture and select an appropriate format (resolution and refresh rate) based on the video clocks available to the core. The required file format is ASCII PPM which must have a suitable amount of pixels. For instance, to draw a screen with resolution 640x480, a PPM file which is 640 pixels wide and 480 pixels high must be used. ASCII PPM files can be created with, for instance, the GNU Image Manipulation Program (The GIMP). The color depth can be either 16 or 32 bits.

svga draw test_screen *fmt bitdepth* *?svga#?*

The **svga draw test_screen** command will show a simple grid in the resolution specified via the format *fmt* selection (see **svga formats** to list all available formats). The color depth can be either 16 or 32 bits.

svga frame *?address?* *?svga#?*

Show or set start address of framebuffer memory

svga formats *?svga#?*

Show available display formats

svga formatsdetailed *?svga#?*

Show detailed view of available display formats

EXAMPLE

Draw a 1024x768, 60Hz test image

```
grmon3> svga draw test_screen 12 32
```

97. symbols - syntax

NAME

symbols - Load, print or lookup symbols

SYNOPSIS

symbols *?options? filename ?cpu#?*

symbols *subcommand ?arg?*

DESCRIPTION

The symbols command is used to load symbols from an object file. It can also be used to print all loaded symbols or to lookup the address of a specified symbol.

symbols *?options? filename ?cpu#?*

Load the symbols from *filename*. If *cpu#* argument is omitted, then the symbols will be associated with the active CPU.

Options:

-tid id

Associate the file with a specific thread. Accepts a thread id or thread path as an argument.

-tname name

Associate the file with a specific thread

symbols clear *?cpu#?*

Remove all symbols associated with the active CPU or a specific CPU.

symbols list *?options? ?cpu#?*

This command lists loaded symbols. If no options are given, then all local and global functions and objects are listed. The optional argument *cpu#* can be used to limit the listing for a specific CPU.

Options:

-global

List global symbols

-local

List local symbols

-func

List functions

-object

List objects

-all

List all symbols

symbols lookup *symbol ?cpu#?*

Lookup the address of the specified symbol using the symbol table of the active CPU. If *cpu#* is specified, then it will only look in the symbol table associated with that CPU.

symbols lookup *address ?cpu#?*

Lookup symbol for the specified address using the symbol table of the active CPU. If *cpu#* is specified, then it will only look in the symbol table associated with that CPU. At most one symbol is looked up.

RETURN VALUE

Upon successful completion **symbols list** will return a list of all symbols and their attributes.

Nothing will be returned when loading or clearing.

Command **symbols lookup** will return the corresponding address or symbol.

EXAMPLE

Load the symbols in the file hello.

```
grmon3> symbols hello
```

List symbols.

```
grmon3> symbols list
```

List all loaded symbols.

```
grmon3> symbols list -all
```

List all function symbols.

```
grmon3> symbols list -func -local -global
```

List all symbols that begins with the letter m

```
grmon3> puts [lsearch -index {3} -subindices -all -inline [symbols list] m*]
```

SEE ALSO

Section 3.6, “Symbolic debug information”

98. system - syntax

NAME

system - Attach or detach devices

SYNOPSIS

system attach *?options? dev...*

system detach *dev...*

DESCRIPTION

system attach *?options? dev...*

Attach and initialize a core. It will also enable the core in the clockgate if needed for known ASIC devices. The `-ni` can be added to avoid initialization (but it will still probe for capabilities).

system detach *dev...*

Detach a device from GRMON to make it run in the background. If a device is detached while being clockgated, then it will remain detached when it becomes enabled in the clockgate.

99. thread - syntax

NAME

thread - Show OS-threads information or backtrace

SYNOPSIS

```
thread info ?cpu#?
thread current ?cpu#?
thread bt id ?cpu#?
thread os
thread tree ?options?
```

DESCRIPTION

The thread command may be used to list all threads or to show backtrace of a specified thread. Note that the only OS:s supported by GRMON are RTEMS, PikeOS and VxWorks.

The thread command tries to auto-detect the running OS on the target. If this mechanism doesn't work it is possible to force which OS thread backend to use by command line options to GRMON. For more information see for example the `-rtems`, `-bmnothreads` and `-nothreads` options.

```
thread info ?cpu#?
thread current ?cpu#?
```

List information about the threads. This should be used to get the id:s for the **thread bt** command.

```
thread bt id ?cpu#?
```

Show backtrace of the thread specified by *id*. The command **thread info** can be used find the available id:s.

```
thread os id
```

Print the name of the current OS.

```
thread tree ?options? ?cpu#?
```

Print a hierarchy of all threads. One of the options `-tid`, `-path` or `-fullname` can be set to print extra information about the threads.

RETURN VALUE

Upon successful completion, **thread info** returns a list of threads. Each entry is a sublist on the format format: `{id name current pc sp }`. See table below for a detailed description.

Name	Description
<i>id</i>	OS specific identification number
<i>name</i>	Name of the thread
<i>current</i>	Boolean describing if the thread is the current running thread.
<i>pc</i>	Program counter
<i>sp</i>	Stack pointer
<i>cpu</i>	Value greater or equal to 0 means that the thread is executing on CPU. Negative value indicates that the thread is idle.

The **thread current** command returns information about the current thread only, using the format described for the return value of the command **thread info** above.

The other subcommands have no return value.

EXAMPLE

List all threads

```
grmon3> thread info
  NAME  TYPE      ID          PRIO  TIME (h:m:s)  ENTRY POINT      PC          ...
* Int.  internal  0x09010001  255   0:0:0.000000000  Test_task        0x4000a5b4  <+0xFFF...
  TA1   classic  0x0a010002   1     0:0:0.064709999  Test_task        0x40016ab8  <_Threa...
  TA2   classic  0x0a010003   1     0:0:0.061212000  Test_task        0x40016ab8  <_Threa...
  TA3   classic  0x0a010004   1     0:0:0.060206998  Test_task        0x40016ab8  <_Threa...
```

TCL returns:

```
{151060481 Int. 1 1073784244 0} {167837698 {TA1 } 0 1073834680 0} {167837699
{TA2 } 0 1073834680 0} {167837700 {TA3 } 0 1073834680 0}
```

SEE ALSO

Section 3.8, “Thread support”

Section 3.8.1, “GRMON thread options”

Section 3.7.6, “GDB Thread support”

100. timer - syntax

timer - Show information about the timer devices

SYNOPSIS

timer *?devname?*

timer reg *?devname?*

DESCRIPTION

timer *?devname?*

This command will show information about the timer device. Optionally which device to show information about can be specified. Device names are listed in 'info sys'.

timer reg *?devname?*

This command will get the timers register. Optionally which device to get can be specified. Device names are listed in 'info sys'.

EXAMPLE

Execute instructions starting at 0x40000000.

```
grmon3> timer 0x40000000
```

101. tmode - syntax

tmode - Select tracing mode between none, processor-only, AHB only or both

SYNOPSIS

tmode

tmode none

tmode both

tmode ahb *boolean*

tmode proc *?boolean? ?cpu#?*

DESCRIPTION

tmode

Print the current tracing mode

tmode none

Disable tracing

tmode both

Enable both AHB and instruction tracing

tmode ahb *?boolean?*

Enable or disable AHB transfer tracing

tmode proc *?boolean? ?cpu#?*

Enable or disable instruction tracing. Use *cpu#* to toggle a single CPU.

EXAMPLE

Disable AHB transfer tracing

```
grmon3> tmode ahb disable
```

SEE ALSO

Section 3.4.9, “Using the trace buffer”

102. tps - syntax

tps - Control the TPS service

SYNOPSIS

```
tps ?port? ?address?  
tps stop  
tps status
```

DESCRIPTION

The **tps** command allows the VxWorks 7 workbench to use the active debug link to debug applications. This removes the need for hardware Ethernet support.

It implements the host side of a virtual interface that only uses memory reads/writes over the debug link for communication. Instead of connecting directly to the target, the workbench needs to connect to GRMON using the port specified when issuing the **tps** command.

See the "*LEON Architectural Support for VxWorks 7*" manual for information on how to configure the target side support.

```
tps ?port? ?address?
```

Start the TPS service. The default port used is 5780 and the default address is taken from the symbol `TPS_DRIVER_REGS` if provided by the application. If the TPS service is already started the command will print the current status.

```
tps stop
```

Stop the TPS service.

```
tps status
```

Print status.

RETURN VALUE

The command **tps** returns a tuple with the port and address used.

103. uhci - syntax

NAME

uhci - Control the USB host's UHCI core

SYNOPSIS

uhci subcommand ?args...?

DESCRIPTION

uhci endian ?devname?

Displays the endian conversion setting

uhci opregs ?devname?

Displays contents of the I/O registers

uhci reset ?devname?

Performs a Host Controller Reset

RETURN VALUE

Upon successful completion, **uhci** have no return value.

SEE ALSO

Section 6.6, “USB Host Controller”

104. usrsh - syntax

NAME

usrsh - Run commands in threaded user shell

SYNOPSIS

usrsh

usrsh *subcommand* *?arg?*

DESCRIPTION

The **usrsh** command is used to create custom user shells. Each custom shell has an associated Tcl interpreter running in a separate thread. Log output from a custom user shell is prefix with its name (see description of the `-log` option in Section 3.2.3, "General options").

usrsh

usrsh list

List all custom user shells.

usrsh add *name*

Create a user shell named *name*. The name is used as an identifier for the shell when using other **usrsh** commands.

usrsh delete *name*

Delete user shell *name*.

usrsh eval *?-bg? ?-std? name arg ?arg ...?*

Evaluate command *arg* in the user shell identified as *name*. If a script is running, then the command will fail with the error code set to EBUSY.

If the option `-bg` is set, then the script will be evaluated in the background, and GRMON will return to the prompt.

If the option `-std`, in combination with option `-bg`, then output from the background operation will be forwarded to the current shells stdout.

usrsh result *name*

Retrieve the result from the last evaluation. If a script is running, then the command will fail with the error code set to EBUSY.

RETURN VALUE

Upon successful completion **usrsh list** will return a list of all custom user shells.

usrsh eval will return the result from the script. If the option `-bg` then nothing will be returned. Instead the **usrsh result** will return the result when the script is finished.

EXAMPLE

Create a user shell named `myshell` and evaluate a command in it.

```
grmon3> usrsh add myshell
Added user shell: myshell
```

```
grmon3> usrsh eval myshell puts "Hello World!"
Hello World!
```

Evaluate command in user shell named `myshell` in the background and wait for it to finish.

```
grmon3> usrsh eval -bg myshell {after 2000; expr 1+1}
grmon3> while {[catch {usrsh result myshell}] && $errorCode == "EBUSY"} {puts "waiting"; after 1000}
waiting
```

```
waiting  
grmon3> puts [usrsh result myshell]  
2
```

SEE ALSO

Section 3.5, “Tcl integration”

105. va - syntax

NAME

va - Translate a virtual address

SYNOPSIS

va *address* *?cpu#?*

DESCRIPTION

va *address* *?cpu#?*

Translate a virtual address. The command will use the MMU from the current active CPU and the *cpu#* can be used to select a different CPU.

OPTIONS

-v

The -v will show a walk output.

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

-vs

The command will return the first stage address. First stage PTEs will still be translated using the second stage.

RETURN VALUE

Command **va** returns the translated address.

SEE ALSO

Section 3.4.15, “Memory Management Unit (MMU) support”

106. verify - syntax

NAME

verify - Verify that a file has been uploaded correctly.

SYNOPSIS

```
verify ?options...? filename ?address?
```

DESCRIPTION

```
verify ?options...? filename ?address?
```

Verify that the file *filename* has been uploaded correctly. If the *address* argument is present, then binary files will be compared against data at this address, if left out then they will be compared to data at the base address of the detected RAM.

RETURN VALUE

Upon successful completion **verify** returns the number of error detected. If the `-errors` has been given, it returns a list of errors instead.

OPTIONS

`-binary`

The `-binary` option can be used to force GRMON to interpret the file as a binary file.

`-bsize bytes`

The `-bsize` option may be used to specify the size blocks of data in bytes that will be written. Sizes that are not even words may require a JTAG based debug link to work properly. See Chapter 5, *Debug link* more information.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 4 bytes, but can be changed using the `-bsize` option.

`-max num`

The `-max` option can be used to force GRMON to stop verifying when *num* errors have been found.

`-bsize bytes`

The `-bsize` option may be used to specify the size of blocks of data in bytes that will be read. Sizes that are not even words may require a JTAG based debug link to work properly. See Chapter 5, *Debug link* more information.

`-errors`

When the `-errors` option is specified, the `verify` returns a list of all errors instead of number of errors. Each element of the list is a sublist whose format depends on the first item if the sublist. Possible errors can be detected are memory verify error (MEM), read error (READ) or an unknown error (UNKNOWN). The formats of the sublists are: `MEM address read-value expected-value`, `READ address num-failed-addresses`, `UNKNOWN address`

EXAMPLE

Load and then verify a `hello_world` application

```
grmon3> load ../hello_world/hello_world
grmon3> verify ../hello_world/hello_world
```

SEE ALSO

Section 3.4.2, “Uploading application and data to target memory”

blod

eload
load

107. vmemb - syntax

NAME

vmemb - AMBA bus 8-bit virtual memory read access, list a range of addresses

SYNOPSIS

```
vmemb ?-ascii? address ?length?
```

DESCRIPTION

```
vmemb ?-ascii? address ?length?
```

GRMON will translate *address* to a physical address, do an AMBA bus read 8-bit read access and print the data. The optional length parameter should be specified in bytes and the default size is 64 bytes. If no MMU exists or if it is turned off, this command will behave like the command **vwmemb**

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read and then parse out the unaligned data.

OPTIONS

-bsize bytes

The **-b**size option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

-delay ms

The **-d**elay option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 1 bytes, but can be changed using the **-b**size option.

-ascii

If the **-a**scii flag has been given, then a single ASCII string is returned instead of a list of values.

-cstr

If the **-c**str flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

-dec

Give the **-d**ec flag to make the Tcl return signed decimal values instead of hexadecimal strings.

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

Upon successful completion **vmemb** returns a list of the requested 8-bit words. Some options changes the result value, see options for more information.

EXAMPLE

Read 4 bytes from address 0x40000000:

```
grmon3> vmemb 0x40000000 4
```

TCL returns:

64 0 0 0

SEE ALSO

Section 3.4.7, “Displaying memory contents”

Section 3.4.15, “Memory Management Unit (MMU) support”

108. vmemd - syntax

NAME

vmemd - AMBA bus 64-bit virtual memory read access, list a range of addresses

SYNOPSIS

```
vmemd ?-ascii? address ?length?
```

DESCRIPTION

```
vmemd ?-ascii? address ?length?
```

GRMON will translate *address* to a physical address, do an AMBA bus read access and print the data as 64-bit words. The optional length parameter should be specified in bytes and the default size is 64 bytes (8 words).

OPTIONS

`-bsize bytes`

The `-bsize` option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 8 bytes, but can be changed using the `-bsize` option.

`-ascii`

If the `-ascii` flag has been given, then a single ASCII string is returned instead of a list of values.

`-cstr`

If the `-cstr` flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

`-dec`

Give the `-dec` flag to make the Tcl return signed decimal values instead of hexadecimal strings.

`-s`

The command will operate using the supervisor MMU table.

`-h`

The command will operate using the hypervisor two-stage MMU tables.

`-hg`

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

Upon successful completion **vmemd** returns a list of the requested 64-bit words. Some options change the result value, see options for more information.

EXAMPLE

Read 2 64-bit words (16 bytes) from address 0xfffffe0000000000:

```
grmon3> vmemd 0xfffffe0000000000 16
```

TCL returns:

```
0xfffff1244901022 0x543348
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

Section 3.4.15, “Memory Management Unit (MMU) support”

109. vmemh - syntax

NAME

vmemh - AMBA bus 16-bit virtual memory read access, list a range of addresses

SYNOPSIS

`vmemh ?-ascii? address ?length?`

DESCRIPTION

`vmemh ?-ascii? address ?length?`

GRMON will translate *address* to a physical address, do an AMBA bus read 16-bit read access and print the data. The optional length parameter should be specified in bytes and the default size is 64 bytes (32 words). If no MMU exists or if it is turned off, this command will behave like the command **vwmemh**

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read and then parse out the unaligned data.

OPTIONS

`-bsize bytes`

The `-bsize` option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 2 bytes, but can be changed using the `-bsize` option.

`-ascii`

If the `-ascii` flag has been given, then a single ASCII string is returned instead of a list of values.

`-cstr`

If the `-cstr` flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

`-dec`

Give the `-dec` flag to make the Tcl return signed decimal values instead of hexadecimal strings.

`-s`

The command will operate using the supervisor MMU table.

`-h`

The command will operate using the hypervisor two-stage MMU tables.

`-hg`

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

Upon successful completion **vmemh** returns a list of the requested 16-bit words. Some options changes the result value, see options for more information.

EXAMPLE

Read 4 words (8 bytes) from address 0x40000000:

```
grmon3> vmemh 0x40000000 8
```

TCL returns:

16384 0 0 0

SEE ALSO

Section 3.4.7, “Displaying memory contents”

Section 3.4.15, “Memory Management Unit (MMU) support”

110. vmem - syntax

NAME

vmem - AMBA bus 32-bit virtual memory read access, list a range of addresses

SYNOPSIS

```
vmem ?-ascii? address ?length?
```

DESCRIPTION

```
vmem ?-ascii? address ?length?
```

GRMON will translate *address* to a physical address, do an AMBA bus read 32-bit read access and print the data. The optional length parameter should be specified in bytes and the default size is 64 bytes (16 words). If no MMU exists or if it is turned off, this command will behave like the command **vwmem**

OPTIONS

-bsize bytes

The *-b*size option can be used to specify the size blocks of data in bytes that will be read between each print to the screen. Setting a high value may increase performance but cause a less smooth printout when using a slow debug link.

-delay ms

The *-d*elay option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 4 bytes, but can be changed using the *-b*size option.

-ascii

If the *-a*scii flag has been given, then a single ASCII string is returned instead of a list of values.

-cstr

If the *-c*str flag has been given, then a single ASCII string, up to the first null character, is returned instead of a list of values.

-dec

Give the *-d*ec flag to make the Tcl return signed decimal values instead of hexadecimal strings.

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

Upon successful completion **vmem** returns a list of the requested 32-bit words. Some options change the result value, see options for more information.

EXAMPLE

Read 4 words from address 0x40000000:

```
grmon3> vmem 0x40000000 16
```

TCL returns:

```
1073741824 0 0 0
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

Section 3.4.15, “Memory Management Unit (MMU) support”

111. vwmemb - syntax

NAME

vwmemb - AMBA bus 8-bit virtual memory write access

SYNOPSIS

```
vwmemb ?options...? address data ?...?
```

DESCRIPTION

```
vwmemb ?options...? address data ?...?
```

Do an AMBA write access. GRMON will translate *address* to a physical address and write the 8-bit value specified by *data*. If more than one data word has been specified, they will be stored at consecutive physical addresses. If no MMU exists or if it is turned off, this command will behave like the command **vwmemb**

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read-modify-write when writing unaligned data.

OPTIONS

-b*size* bytes

The *-bsize* option may be used to specify the size blocks of data in bytes that will be written.

-d*elay* ms

The *-delay* option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 1 bytes, but can be changed using the *-bsize* option.

-w*prot*

Disable memory controller write protection during the write.

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

vwmemb has no return value.

EXAMPLE

Write 0xAB to address 0x40000000 and 0xCD to 0x40000004:

```
grmon3> vwmemb 0x40000000 0xAB 0xCD
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

Section 3.4.15, “Memory Management Unit (MMU) support”

112. vwmemd - syntax

NAME

vwmemd - AMBA bus 64-bit virtual memory write access

SYNOPSIS

```
vwmemd ?options...? address data ?...?
```

DESCRIPTION

```
vwmemd ?options...? address data ?...?
```

Do an AMBA write access. GRMON will translate *address* to a physical address and write the 64-bit value specified by *data*. If more than one data word has been specified, they will be stored at consecutive physical addresses.

OPTIONS

`-bsize bytes`

The `-bsize` option may be used to specify the size blocks of data in bytes that will be written.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 8 bytes, but can be changed using the `-bsize` option.

`-wprot`

Disable memory controller write protection during the write.

`-s`

The command will operate using the supervisor MMU table.

`-h`

The command will operate using the hypervisor two-stage MMU tables.

`-hg`

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

`vwmemd` has no return value.

EXAMPLE

Write 0xffff1244901022 to address 0xfffffe000000000 and 0x1234 to 0xfffffe000000008:
grmon3> vwmemd 0xffffffe000000000 0xffff1244901022 0x1234

SEE ALSO

Section 3.4.7, “Displaying memory contents”
Section 3.4.15, “Memory Management Unit (MMU) support”

113. vwmemh - syntax

NAME

vwmemh - AMBA bus 16-bit virtual memory write access

SYNOPSIS

```
vwmemh ?options...? address data ?...?
```

DESCRIPTION

```
vwmemh ?options...? address data ?...?
```

Do an AMBA write access. GRMON will translate *address* to a physical address and write the 16-bit value specified by *data*. If more than one data word has been specified, they will be stored at consecutive physical addresses. If no MMU exists or if it is turned off, this command will behave like the command **vwmemh**

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read-modify-write when writing unaligned data.

OPTIONS

-bsize bytes

The *-bsize* option may be used to specify the size blocks of data in bytes that will be written.

-delay ms

The *-delay* option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 2 bytes, but can be changed using the *-bsize* option.

-wprot

Disable memory controller write protection during the write.

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

vwmemh has no return value.

EXAMPLE

Write 0xABCD to address 0x40000000 and 0x1234 to 0x40000004:

```
grmon3> vwmemh 0x40000000 0xABCD 0x1234
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

Section 3.4.15, “Memory Management Unit (MMU) support”

114. vwmems - syntax

NAME

vwmems - Write a string to an AMBA bus virtual memory address

SYNOPSIS

vwmems *address data*

DESCRIPTION

vwmems *address data*

Do an AMBA write access. GRMON will translate *address* to a physical address and write the string value specified by *data*, including the terminating NULL-character. If no MMU exists or if it is turned off, this command will behave like the command **vwmems'**

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read-modify-write when writing unaligned data.

OPTIONS

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

vwmems has no return value.

EXAMPLE

Write "Hello World" to address 0x40000000-0x4000000C:
grmon3> vwmems 0x40000000 "Hello World"

SEE ALSO

Section 3.4.7, "Displaying memory contents"
Section 3.4.15, "Memory Management Unit (MMU) support"

115. vwmem - syntax

NAME

vwmem - AMBA bus 32-bit virtual memory write access

SYNOPSIS

```
vwmem ?options...? address data ?...?
```

DESCRIPTION

```
vwmem ?options...? address data ?...?
```

Do an AMBA write access. GRMON will translate *address* to a physical address and write the 32-bit value specified by *data*. If more than one data word has been specified, they will be stored at consecutive physical addresses. If no MMU exists or if it is turned off, this command will behave like the command **vwmem**

OPTIONS

-bsize bytes

The *-bsize* option may be used to specify the size blocks of data in bytes that will be written.

-delay ms

The *-delay* option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 4 bytes, but can be changed using the *-bsize* option.

-wprot

Disable memory controller write protection during the write.

-s

The command will operate using the supervisor MMU table.

-h

The command will operate using the hypervisor two-stage MMU tables.

-hg

The command will operate using the hypervisor guest MMU table.

RETURN VALUE

vwmem has no return value.

EXAMPLE

Write 0xABCD1234 to address 0x40000000 and to 0x40000004:
grmon3> vwmem 0x40000000 0xABCD1234 0xABCD1234

SEE ALSO

Section 3.4.7, “Displaying memory contents”
Section 3.4.15, “Memory Management Unit (MMU) support”

116. walk - syntax

NAME

walk - Translate a virtual address, print translation

SYNOPSIS

```
walk ?options...? address ?cpu#?
```

DESCRIPTION

```
walk ?options...? address ?cpu#?
```

Translate a virtual address and print translation. The command will use the MMU from the current active CPU and the `cpu#` can be used to select a different CPU.

OPTIONS

`-v`

The `-v` will show the intermediate steps when doing a hypervisor two-stage translation.

`-s`

The command will operate using the supervisor MMU table.

`-h`

The command will operate using the hypervisor two-stage MMU tables.

`-hg`

The command will operate using the hypervisor guest MMU table.

`-vs`

The command will return the first stage address. First stage PTEs will still be translated using the second stage.

RETURN VALUE

Command **walk** returns the translated address.

SEE ALSO

Section 3.4.15, “Memory Management Unit (MMU) support”

117. wash - syntax

wash - Clear memory or set all words in a memory range to a value.

SYNOPSIS

```
wash ?options...? ?start stop? ?value?
```

DESCRIPTION

```
wash ?options...?
```

Clear all memories.

```
wash ?options...? start stop ?value?
```

Wash the memory area from *start* up to *stop* and set each word to *value*. The parameter *value* defaults to 0.

OPTIONS

```
-delay ms
```

The `-delay` option can be used to specify a delay between each word written.

```
-nic
```

Disable the instruction cache while washing the memory

```
-nocpu
```

Do not use the CPU to increase performance.

```
-wprot
```

If the `-wprot` option is given then write protection on the memory will be disabled

EXAMPLE

Clear all memories

```
grmon3> wash
```

Set a memory area to 1

```
grmon3> wash 0x40000000 0x40000FFF 1
```

SEE ALSO

Section 3.10.1, “Using EDAC protected memory”

118. wmdio - syntax

NAME

wmdio - Set PHY registers

SYNOPSIS

```
wmdio paddr raddr value ?greth#?
```

DESCRIPTION

```
wmdio paddr raddr value ?greth#?
```

Set *value* of PHY address *paddr* and register *raddr*. If more than one device exists in the system, the *greth#* can be used to select device, default is greth0. The command tries to disable the EDCL duplex detection if enabled.

SEE ALSO

Section 6.4, “Ethernet controller”

119. wmemb - syntax

NAME

wmemb - AMBA bus 8-bit memory write access

SYNOPSIS

```
wmemb ?options...? address data ?...?
```

DESCRIPTION

```
wmemb ?options...? address data ?...?
```

Do an AMBA write access. The 8-bit value specified by *data* will be written to *address*. If more than one data word has been specified, they will be stored at consecutive addresses.

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read-modify-write when writing unaligned data.

OPTIONS

-bsize bytes

The *-bsize* option may be used to specify the size blocks of data in bytes that will be written.

-delay ms

The *-delay* option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 1 bytes, but can be changed using the *-bsize* option.

-wprot

Disable memory controller write protection during the write.

-delay ms

Insert a delay between each block of data

-asi asi

Write from SPARC alternate space.

RETURN VALUE

wmemb has no return value.

EXAMPLE

Write 0xAB to address 0x40000000 and 0xBC to 0x40000001:

```
grmon3> wmemb 0x40000000 0xAB 0xBC
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

120. wmemd - syntax

NAME

wmemd - AMBA bus 64-bit memory write access

SYNOPSIS

```
wmemd ?options...? address data ?...?
```

DESCRIPTION

```
wmemd ?options...? address data ?...?
```

Do an AMBA write access. The 64-bit value specified by *data* will be written to *address*. If more than one data word has been specified, they will be stored at consecutive addresses.

OPTIONS

`-bsize bytes`

The `-bsize` option may be used to specify the size blocks of data in bytes that will be written.

`-delay ms`

The `-delay` option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 8 bytes, but can be changed using the `-bsize` option.

`-wprot`

Disable memory controller write protection during the write.

`-delay ms`

Insert a delay between each block of data

`-asi asi`

Write from SPARC alternate space.

RETURN VALUE

wmemd has no return value.

EXAMPLE

Write 0xffff1244901022 to address 0x40000000 and 0x1234 to 0x40000008:

```
grmon3> wmemd 0x40000000 0xffff1244901022 0x1234
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

121. wmemh - syntax

NAME

wmemh - AMBA bus 16-bit memory write access

SYNOPSIS

```
wmemh ?options...? address data ?...?
```

DESCRIPTION

```
wmemh ?options...? address data ?...?
```

Do an AMBA write access. The 16-bit value specified by *data* will be written to *address*. If more than one data word has been specified, they will be stored at consecutive addresses.

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read-modify-write when writing unaligned data.

OPTIONS

-bsize bytes

The *-bsize* option may be used to specify the size blocks of data in bytes that will be written.

-delay ms

The *-delay* option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 2 bytes, but can be changed using the *-bsize* option.

-wprot

Disable memory controller write protection during the write.

-delay ms

Insert a delay between each block of data

-asi asi

Write from SPARC alternate space.

RETURN VALUE

wmemh has no return value.

EXAMPLE

Write 0xABCD to address 0x40000000 and 0x1234 to 0x40000002:

```
grmon3> wmem 0x40000000 0xABCD 0x1234
```

SEE ALSO

Section 3.4.7, “Displaying memory contents”

122. wmems - syntax

NAME

wmems - Write a string to an AMBA bus memory address

SYNOPSIS

wmems *?options...? address data*

DESCRIPTION

wmems *address data*

Write the string value specified by *data*, including the terminating NULL-character, to *address*.

Only JTAG debug links supports byte accesses. Other debug links will do a 32-bit read-modify-write when writing unaligned data.

OPTIONS

*-b*size bytes

The *-b*size option may be used to specify the size blocks of data in bytes that will be written.

*-d*elay ms

The *-d*elay option can be used to specify a delay between each word written. If the delay is non-zero then the default block size will be 8 bytes, but can be changed using the *-b*size option.

*-w*prot

Disable memory controller write protection during the write.

*-d*elay ms

Insert a delay between each block of data

*-a*si asi

Write from SPARC alternate space.

RETURN VALUE

wmems has no return value.

EXAMPLE

Write "Hello World" to address 0x40000000-0x4000000C:
grmon3> wmems 0x40000000 "Hello World"

SEE ALSO

Section 3.4.7, "Displaying memory contents"

123. wmem - syntax

NAME

wmem - AMBA bus 32-bit memory write access

SYNOPSIS

```
wmem ?options...? address data ?...?
```

DESCRIPTION

```
wmem ?options...? address data ?...?
```

Do an AMBA write access. The 32-bit value specified by *data* will be written to *address*. If more than one data word has been specified, they will be stored at consecutive addresses.

OPTIONS

-bsize bytes

The *-bsize* option may be used to specify the size blocks of data in bytes that will be written.

-wprot

Disable memory controller write protection during the write.

-delay ms

Insert a delay between each block of data

-asi asi

Write from SPARC alternate space.

RETURN VALUE

wmem has no return value.

EXAMPLE

Write 0xABCD1234 to address 0x40000000 and to 0x40000004:
grmon3> wmem 0x40000000 0xABCD1234 0xABCD1234

SEE ALSO

Section 3.4.7, "Displaying memory contents"

Appendix C. Tcl API

GRMON will automatically load the scripts in GRMON appdata folder. On Linux the appdata folder is located in `~/ .grmon-4.0/` and on Windows it's typically located at `C:\Users\%username%\AppData\Roaming\Frontgrade Gaisler\GRMON\4.0`. In the folder there are two different sub folders where scripts may be found, `<appdata>/scripts/sys` and `<appdata>/scripts/user`. Scripts located in the sys-folder will be loaded into the system shell only, before the Plug and Play area is scanned, i.e. drivers and fix-ups should be defined here. The scripts found in the user-folder will be loaded into all shells (including the system shell), i.e. all user defined commands and hooks should be defined there.

In addition there are two commandline switches `-udrv <filename>` and `-ucmd <filename>` to load scripts into the system shell or all shells.

TCL API switches:

- `-udrv<filename>`
Load script specified by filename into system shell. This option is mainly used for user defined drivers.
- `-ucmd<filename>`
Load script specified by filename into all shells, including the system shell. This option is mainly used for user defined procedures and hooks.

Also the TCL command **source** or GRMON command **batch** can be used to load a script into a single shell.

The variable TCL `grmon_shell` can be used to identify a shell. This can be used to run shell specific code from a script is intended to be used in multiple shells. GRMON creates the following shells:

sys	System shell
exec	Execution shell
cli	Command line interface shell
term#	GUI terminal shell (# is replaced by a number)
gdb	GDB remote server shell

Example using shell name:

```
if {$grmon_shell == "cli"} {
    puts "Hello CLI!"
}
```

1. Device names

All GRLIB cores are assigned a unique `adevN` name, where N is a unique number. The debug driver controlling the core also provides an alias which is easier to remember. For example the name `mctrl0` will point to the first MCTRL regardless in which order the AMBA Plug and Play is assigned, thus the name will be consistent between different chips. The names of the cores are listed in the output of the GRMON command **info sys**.

PCI devices can also be registered into GRMON's device handling system using one of the **pci conf -reg**, **pci scan -reg** or **pci bus reg** commands. The devices are handled similar to GRLIB devices, however their base name is `pdevN`.

It is possible to specify one or more device names as an argument to the GRMON commands **info sys** and **info reg** to show information about those devices only. For **info reg** a register name can also be specified by appending the register name to the device name separated by colon. Register names are the same as described in Section 2, "Variables".

For each device in a GRLIB system, a namespace will be created. The name of the namespace will be the same as the name of the device. Inside the namespace Plug and Play information is available as variables. Most debug drivers also provide direct access to APB or AHB registers through variables in the namespace. See Section 2, "Variables" for more details about variables.

Below is an example of how the first MCTRL is named and how the APB register base address is found using Plug and Play information from the GRMON `mctrl0` variable. The eleventh PCI device (a network card) is also listed using the unique name `pdev10`.

```

grmon3> info sys mctrl0
mctrl0   Frontgrade Gaisler Memory controller with EDAC
         AHB: 00000000 - 20000000
         AHB: 20000000 - 40000000
         AHB: 40000000 - 80000000
         APB: 80000000 - 80000100
         8-bit prom @ 0x00000000
         32-bit static ram: 1 * 8192 kbyte @ 0x40000000
         32-bit sdram: 2 * 128 Mbyte @ 0x60000000
         col 10, cas 2, ref 7.8 us
grmon3> info sys pdev10
pdev10   Bus 02 Slot 03 Func 00 [2:3:0]
         vendor: 0x1186 D-Link System Inc
         device: 0x4000 DL2000-based Gigabit Ethernet
         class: 020000 (ETHERNET)
         subvendor: 0x1186, subdevice: 0x4004
         BAR1: 00001000 - 00001100 I/O-32 [256B]
         BAR2: 82203000 - 82203200 MEMIO [512B]
         ROM: 82100000 - 82110000 MEM [64kB]
         IRQ INTA# -> IRQW

```

2. Variables

GRMON provides variables that can be used in scripts. A list of the variables can be found below.

`grmon_version`

The version number of GRMON

`grmon_shell`

The name of the shell

`grmon::settings::suppress_output`

The variable is a bitmask to control GRMON output.

- bit 0 Block all output from GRMON commands to the terminal
- bit 1 Block all output from TCL commands (i.e. puts) to the terminal
- bit 2 Block all output from GRMON commands to the log
- bit 3 Block all output from TCL commands to the log (i.e. puts)

`grmon::settings::echo_result`

If setting this to one, then the result of a command will always be printed in the terminal.

`grmon::settings::nb`

`grmon::settings::nb_mask`

When the `nb` option is set, the OS running on the CPU must handle all error traps. The CPUs will not go into debug mode when a error trap occurs. The `nb` variable configures all CPUs and the `nb_mask` variable is a mask (one bit per CPU) for an asymmetric configuration. (LEON only)

`grmon::settings::nswb`

`grmon::settings::nswb_mask`

When the `nswb` flag is set, the CPUs will not go into debug mode when a software breakpoint occur. This option is required when a native software debugger like GDB is running on the target CPU. The `nswb` variable configures all CPUs and the `nswb_mask` variable is a mask (one bit per CPU) for an asymmetric configuration.

`gplib_device`

The device ID of the system, read from the plug and play area.

`grmon::interrupt`

This variable will be set to 1 when a user issues an interrupt (i.e. pressing Ctrl-C from the commandline), it's always set to zero before a commands sequence is issued. It can be used to abort user defined commands.

It is also possible to write this variable from inside hooks and procedures. E.g. writing a 1 from a `exec` hook will abort the execution

grlib_build

The build ID of the system, read from the plug and play area.

grlib_system

The name of the system. Only valid on known systems.

grlib_freq

The frequency of the system in Hz.

<devname#>¹::pnp::device
 <devname#>¹::pnp::vendor
 <devname#>¹::pnp::mst::custom0
 <devname#>¹::pnp::mst::custom1
 <devname#>¹::pnp::mst::custom2
 <devname#>¹::pnp::mst::irq
 <devname#>¹::pnp::mst::idx
 <devname#>¹::pnp::ahb::0::start
 <devname#>¹::pnp::ahb::0::mask
 <devname#>¹::pnp::ahb::0::type
 <devname#>¹::pnp::ahb::custom0
 <devname#>¹::pnp::ahb::custom1
 <devname#>¹::pnp::ahb::custom2
 <devname#>¹::pnp::ahb::irq
 <devname#>¹::pnp::ahb::idx
 <devname#>¹::pnp::apb::start
 <devname#>¹::pnp::apb::mask
 <devname#>¹::pnp::apb::irq
 <devname#>¹::pnp::apb::idx

The AMBA Plug and Play information is available for each AMBA device. If a device has an AHB Master (mst), AHB Slave (ahb) or APB slave (apb) interface, then the corresponding variables will be created.

¹Replace with device name.

```

<devname#>1::vendor
<devname#>1::device
<devname#>1::command
<devname#>1::status
<devname#>1::revision
<devname#>1::ccode
<devname#>1::csize
<devname#>1::tlat
<devname#>1::htype
<devname#>1::bist
<devname#>1::bar0
<devname#>1::bar1
<devname#>1::bar2
<devname#>1::bar3
<devname#>1::bar4
<devname#>1::bar5
<devname#>1::cardbus
<devname#>1::subven
<devname#>1::subdev
<devname#>1::rombar
<devname#>1::pri
<devname#>1::sec
<devname#>1::sord
<devname#>1::sec_tlat
<devname#>1::io_base
<devname#>1::io_lim
<devname#>1::secsts
<devname#>1::memio_base
<devname#>1::memio_lim
<devname#>1::mem_base
<devname#>1::mem_lim
<devname#>1::mem_base_up
<devname#>1::mem_lim_up
<devname#>1::io_base_up
<devname#>1::io_lim_up
<devname#>1::capptr
<devname#>1::res0
<devname#>1::res1
<devname#>1::rombar
<devname#>1::iline
<devname#>1::ipin
<devname#>1::min_gnt
<devname#>1::max_lat
<devname#>1::bridge_ctrl

```

If the PCI bus has been registered into the GRMON's device handling system the PCI Plug and Play configuration space registers will be accessible from the Tcl variables listed above. Depending on the PCI header layout (standard or bridge) some of the variables list will not be available. Some of the read-only registers such as DEVICE and VENDOR are stored in GRMON's memory, accessing such variables will not generate PCI configuration accesses.

```

<devname#>1::<regname>2
<devname#>1::<regname>2::<fldname>3

```

Many devices exposes their registers, and register fields, as variables. When writing these variables, the registers on the target system will also be written.

```

grmon3> info sys
...

```

²Replace with a register name

³Replace with a register field name

```

mctrl0    Frontgrade Gaisler Memory controller with EDAC
          AHB: 00000000 - 20000000
          AHB: 20000000 - 40000000
          AHB: 40000000 - 80000000
          APB: 80000000 - 80000100
          8-bit prom @ 0x00000000
          32-bit static ram: 1 * 8192 kbyte @ 0x40000000
          32-bit sdram: 2 * 128 Mbyte @ 0x60000000
          col 10, cas 2, ref 7.8 us
...
grmon3> puts [ format 0x%x $mctrl0::                [TAB-COMPLETION]
mctrl0::mcfg1 mctrl0::mcfg2 mctrl0::mcfg3 mctrl0::pnp::
mctrl0::mcfg1:: mctrl0::mcfg2:: mctrl0::mcfg3::
grmon3> puts [ format 0x%x $mctrl0::pnp::          [TAB-COMPLETION]
mctrl0::pnp::ahb:: mctrl0::pnp::device mctrl0::pnp::ver
mctrl0::pnp::apb:: mctrl0::pnp::vendor
grmon3> puts [ format 0x%x $mctrl0::pnp::apb::    [TAB-COMPLETION]
mctrl0::pnp::apb::irq mctrl0::pnp::apb::mask mctrl0::pnp::apb::start
grmon3> puts [ format 0x%x $mctrl0::pnp::apb::start ]
          0x80000000

```

3. User defined hooks

GRMON supports user implemented hooks using Tcl procedures. Each hook is variable containing a list of procedure names. GRMON will call all the procedures in the list.

Like normal procedures in TCL, each hook can return a code and a result value using the TCL command **return**. If a hook returns a code that is not equal to zero, then the GRMON will skip the rest of the hooks that are registered in that list. Some hooks will change GRMONs behavior depending on the return code, see hook descriptions below.

Hooks in the system shell can only be installed using a startup script, see `-udrv<filename>` for more information.[api.udrv](#)

Hooks can be installed in the execution shell using the command **grmon::execsh eval <script>** or using a startup script. The hooks that are installed in the execution shell will be called when the target system is started from a shell or from the GUI.

To uninstall hooks, either remove the procedure name from the list using the Tcl **lreplace** or delete the variable using **unset** to uninstall all hooks. Hooks in the system shell can only be uninstalled in the startup script or by letting the hook uninstall itself. Always use **lreplace** when uninstalling hooks in the system shell, otherwise it's possible to delete hooks the GRMON has installed that may lead to undefined behavior.

preinit

The preinit hooks is called after GRMON has connected to the board and before any driver initialization is done. It is also called before the plug and play area is scanned. The hook may only be defined in the system shell.

postinit

The post init hook is called after all drivers have been initialized. The hook may only be defined in the system shell.

init#

During GRMON's startup, 9 hooks are executed. These hooks are called `init1`, `init2`, etc. Each hook is called before the corresponding init function in a user defined driver is called. In addition `init1` is called after the plug and play area is scanned, but before any initialization. The `init#` hooks may only be defined in the system shell.

deinit

Called when GRMON is closing down. The `deinit` hooks may only be defined in the system shell.

closedown

Called when a TCL is closing down.

preexec

These hooks are called before the CPU:s are started, when issuing a **run**, **cont** or **go** command. They may be defined in the execution shell or the local shell that is executing the command.

exec

The `exec` hooks are called once each iteration of the polling loop, when issuing a **run**, **cont** or **go** command. They may be defined in the execution shell or the local shell that is executing the command.

postexec

These hooks are called after the CPU:s have stopped, when issuing a **run**, **cont** or **go** command. They may be defined in the execution shell or the local shell that is executing the command.

load

This hook is called before each block of data is written to the target. See tables below for argument description and return code definitions for the hook procedure.

Argument	Type	Description
<i>addr</i>	integer	Destination addr
<i>bytes</i>	integer	Number of bytes
<i>data</i>	byte-array	Data to be written

Return Code	Value	Description
0	-	The hook was successful, but let GRMON continue as usual. This can be used to do extra configuration or fix-ups. Any return value will be ignored.
-1	Integer value	The hook overrides GRMON and the access was successful. Any return value will be ignored.
1	Error text	The hook overrides GRMON and the access failed. Any return value will be ignored.

pcicfg

This hook is called when a PCI configuration read access is issued. It can be used to override GRMON's PCI configuration space access routines. See tables below for argument descriptions and return codes/value definitions for the hook procedure.

Argument	Type	Description
<i>bus</i>	integer	Bus index
<i>slot</i>	integer	Slot index
<i>func</i>	integer	Function index
<i>ofs</i>	integer	Offset into the device's configuration space
<i>size</i>	integer	Size in bits of the access (8, 16 or 32)

Return Code	Value	Description
0	-	The hook was successful, but let GRMON continue as usual. This can be used to do extra configuration or fix-ups. Any return value will be ignored.
-1	Integer value	The hook overrides GRMON and the access was successful. Return the value read.
1	Error text	The hook overrides GRMON and the access failed. Return an error description.

pciwcfg

This hook is called when a PCI configuration write access is issued. It can be used to override GRMON's PCI configuration space access routines. See tables below for argument descriptions and return codes/value definitions the hook procedure.

Argument	Type	Description
<i>bus</i>	integer	Bus index
<i>slot</i>	integer	Slot index
<i>func</i>	integer	Function index
<i>ofs</i>	integer	Offset into the device's configuration space
<i>size</i>	integer	Size in bits of the access (8, 16 or 32)
<i>value</i>	integer	The value to be written

Return Code	Value	Description
0	-	The hook was successful. GRMON continue doing the access. This can be used to do extra configuration or fix-ups. Any return value will be ignored.
-1	-	The hook overrides GRMON and the access was successful. Any return value will be ignored.
1	Error text	The hook overrides GRMON and the access failed. Return an error description.

reset

The reset hook is called after GRMON has connected to the board and when a command reset or run is issued.

Example C.1. Using hooks

```
# Define hook procedures
grmon::execsh eval {
  proc myhook1 {} {puts "Hello World"}
  proc myhook2 {} {puts "Hello again"; return -code 1 "Blocking next hook"}
  proc myhook3 {} {puts "Will never run"}
  lappend ::hooks::preexec ::myhook1 ::myhook2 ::myhook3 ;# Add hooks
}

run
grmon::execsh eval {unset ::hooks::preexec ;# Remove all hooks}

proc mypcicfg {bus slot func ofs size} {
  if {$size == 32} {
    return -code -1 0x01234567
  } elseif {$size == 16} {
    return -code -1 0x89AB
  } elseif {$size == 8} {
    return -code -1 0xCD
  }
  return -code 1 "Unknown size"
}
lappend ::hooks::pcicfg ::mypcicfg ;# Add hooks
puts [format 0x%x [pci cfg16 0:1:0 0]]
```

4. User defined driver

It is possible to extend GRMON with user defined drivers by implementing certain hooks and variables in Tcl. GRMON scans the namespace `::drivers` for user defined drivers. Each driver must be located in the subnamespace with the name of the driver. Only the variables `vendor`, `device`, `version_min`, `version_max` and `description` are required to be implemented, the other variables and procedures are optional. The script must be loaded into the system shell.

Cores that GRMON finds while scanning the plug and play area, will be matched against the defined `vendor`, `device` and `version_min/max` variables. If it matches, then the core will be paired with the driver. If a driver is called 'mydrv', then the first found core will be named 'mydrv0', the second 'mydrv1', etc. This name will be passed to the to all the procedures defined in the driver, and can be used to identify the core.

The name of the driver may *not* end with a number.

variable `vendor`

The plug and play vendor identification number.

variable `device`

The plug and play device identification number.

variable `version_min`

variable `version_max`

Minimum and maximum version of the core that this driver supports

variable `description`

A short description of the device

variable regs (optional)

If implemented, the `regs` variable contains information used to parse the registers and present them to the user, i.e. they will be printed in 'info reg' and Tcl-variables will be created in each shell. All register descriptions must be put in the `regs` variable. Each register consists of a name, description and an optional list of fields. The field entries are a quadruple on the format {name pos bits description}. Name should be max 16 characters and description should be max 38 characters long.

proc info devname (optional)

Optional procedure that may be used to present parsed information when 'info sys' is called. Returns a newline separated string.

proc init {devname level} (optional)

Optional procedure that will be called during initialization. The procedure will be called nine times for each device, with level argument set to 1-9. This way drivers that depend on another driver can be initialized in a safe way. Normally initialization of devices is done in level 7.

proc restart devname (optional)

Procedure to reinitialize the device to a known state. This is called when GRMON starts (after initialization) and when commands 'run' or 'reset' is issued.

proc regaddr {devname regname} (optional)

Required only if registers have been defined. It returns the address of the requested register. It's required to be implemented if the variable `regs` is implemented.

If the variable `regs` is implemented, then the procedure `regaddr` is required.

```
namespace eval drivers::mydrv {
  # These variables are required
  variable vendor 0x1
  variable device 0x16
  variable version_min 0
  variable version_max 0
  variable description "My device description"

  # Proc    init
  # Args    devname: Device name
  #         level  : Which stage of initialization
  # Return  -
  #
  # Optional procedure that will be called during initialization. The procedure
  # will be called with level argmuent set to 1-9, this way drivers that depend
  # on another driver can be initialized in a safe way. Normally
  # initialization is done in level 7.
  #
  # Commands wmem and mem can be used to access the registers. Use the driver procedure
  # regaddr to calculate addresses or use static addresses.
  proc init {devname level} {
    puts "init $devname $level"
    if {$level == 7} {
      puts "Hello $devname!"
      puts "Reg1 = mem [regaddr $devname myreg1] 4"
    }
  }

  # Proc    restart
  # Args    devname: Device name
  # Return  -
  #
  # Optional procedure to reinit the device. This is called when GRMON start,
  # when commands 'run' or 'reset' is issued.
  proc restart devname {
    puts "restart $devname"
  }

  # Proc    info
  # Args    devname: Device name
  # Return  A newline-separated string
  #
  # Optional procedure that may be used to present parsed information when
  # 'info sys' is called.
  proc info devname {
    set str "Some extra information about $devname"
    append str "\nSome more information about $devname"
    return $str
  }

  # Proc    regaddr
  # Args    devname: Device name,
  #         regname: Register name
}
```

```

# Return Address of requested register
#
# Required only if any registers have been defined.
# This is a suggestion how the procedure could be implemented
proc regaddr {devname regname} {
    array set offsets {myreg1 0x0 myreg2 0x4}
    if {[namespace exists ::[set devname]::pnp::apb]} {
        set start [set ::[set devname]::pnp::apb::start]
    } elseif {[namespace exists ::[set devname]::pnp::ahb]} {
        set start [set ::[set devname]::pnp::ahb::0::start]
    } else {
        error "Unknown register address for $devnam::$regname"
    }
    return [format 0x%08x [expr ($start + $offsets($regname)) & 0xFFFFFFFF]]
}

# Register descriptions
#
# All description must be put in the regs-namespace. Each register consist
# of a name, description and an optional list of fields.
# The fields are quadruple of the format {name pos bits description}
# Name should be max 16 characters and description should be max 38 characters long.
#
# Registers and fields can be added, removed or changed up to initialization
# level 8. After level 8 TCL variables are created and the regs variable
# should be considered to a constant.
variable regs {
    {"myreg1" "Register1 description"
        {"myfld3" 4 8 "Field3 description"}
        {"myfld2" 1 1 "Field2 description"}
        {"myfld1" 0 1 "Field1 description"}
    }
    {"myreg2" "Register2 description"
    }
}
}; # End of mydrv

```

5. User defined commands

User defined commands can be implemented as Tcl procedures, and then loaded into all shells. See the documentation of the proc command [<http://www.tcl.tk/man/tcl8.6/TclCmd/proc.htm>] on the Tcl website for more information.

6. Links

More about Tcl, its syntax and other useful information can be found at:

Tcl Website [<http://www.tcl.tk>]

Tcl Commands [<http://www.tcl.tk/man/tcl8.6/TclCmd/contents.htm>]

Tcl Tutorial [<http://www.tcl.tk/man/tcl8.5/tutorial/tcltutorial.html>]

Tcler's Wiki [<http://wiki.tcl.tk/>]

Appendix D. Fixed target configuration file format

To use a fixed configuration file, GRMON should be started with `-cfg file`. A fixed configuration file can be used to describe the target system instead of reading the plug and play information. The configuration file describes which IP cores are present on the target and on which addresses they are mapped, using an XML format. An description file can be generated from an plug and play system using the command `info sys -xml file`.

Valid tags for the XML format are described below.

<grxml>

- Parents:
- Children: grlib

Attribute	Description
version	Version of the XML syntax

<grlib>

- Parents: grxml
- Children: bus

Attribute	Description
build	GRLIB build identification number
device	GRLIB device identification number

<bus>

- Parents: grlib, slave, bus
- Children: master, slave, bus

Attribute	Description
type	Valid values are AHB or APB
ffactor	Frequency factor relatvive parent bus

<master>

- Parents: bus
- Children:

Attribute	Description
vendor	Core vendor identification number
device	Core device identification number
version	Version number
irq	Assigned interrupt number

<slave>

- Parents: bus
- Children: bus, bar, custom

Attribute	Description
vendor	Core vendor identification number
device	Core device identification number
version	Version number
irq	Assigned interrupt number

<bar>

- Parents: slave
- Children:

Attribute	Description
address	Base address of the bar
length	Length of the bar in bytes

<custom>

- Parents: slave
- Children:

Attribute	Description
register	Value of the user defined bar

Below is an example configuration file for a simple LEON3 system.

```
<?xml version="1.0" standalone="yes"?>
<grxml version="1.0">
  <gplib device="0x0" build="4109">
    <bus type="AHB" ffactor="1.000000">
      <!-- LEON3 SPARC V8 Processor -->
      <master vendor="0x1" device="0x3">
        </master>
      <!-- JTAG Debug Link -->
      <master vendor="0x1" device="0x1c" version="1">
        </master>
      <!-- LEON2 Memory Controller -->
      <slave vendor="0x4" device="0xf">
        <bar address="0x00000000" length="0x20000000"/>
        <bar address="0x20000000" length="0x20000000"/>
        <bar address="0x40000000" length="0x40000000"/>
      </slave>
      <!-- AHB/APB Bridge -->
      <slave vendor="0x1" device="0x6">
        <bar address="0x80000000" length="0x100000"/>
        <bus type="APB" ffactor="1.000000">
          <!-- LEON2 Memory Controller -->
          <slave vendor="0x4" device="0xf">
            <bar address="0x80000000" length="0x100"/>
          </slave>
          <!-- Generic UART -->
          <slave vendor="0x1" device="0xc" irq="2" version="1">
            <bar address="0x80000100" length="0x100"/>
          </slave>
          <!-- Multi-processor Interrupt Ctrl. -->
          <slave vendor="0x1" device="0xd" version="3">
            <bar address="0x8000200" length="0x100"/>
          </slave>
          <!-- Modular Timer Unit -->
          <slave vendor="0x1" device="0x11" irq="8">
            <bar address="0x8000300" length="0x100"/>
          </slave>
          <!-- General Purpose I/O port -->
          <slave vendor="0x1" device="0x1a" version="1">
            <bar address="0x8000500" length="0x100"/>
          </slave>
        </bus>
      </slave>
      <!-- LEON3 Debug Support Unit -->
      <slave vendor="0x1" device="0x4" version="1">
        <bar address="0x90000000" length="0x10000000"/>
      </slave>
    </bus>
  </gplib>
</grxml>
```

Appendix E. License key installation

GRMON is licensed using a Sentinel LDK USB hardware key and has support for node-locked and floating license keys. The type of key can be identified by the color of the USB dongle. The node-locked keys are purple and the floating license keys are red.

1. Sentinel LDK Run-time

The latest run-time can be found at the GRMON download page. Included in the downloaded Sentinel LDK run-time archive is a README file which contains system requirements and detailed installation instructions. However, ignore all instructions about installing `haspplib_<vendorID>.so` and/or `haspplib_x86_64_<vendorID>.so`.

Administrator privileges are required on Windows. On Linux it is required that the run-time is installed as root user.

GRMON download page [<http://www.gaisler.com/index.php/downloads/debug-tools>]

2. Node-locked keys (purple USB key)

For node-locked keys, the Sentinel LDK Run-time for the key must be installed before the key can be used.

3. Floating keys (red USB key)

In the case of floating keys, the Sentinel LDK Run-time must be installed on the server and the client computer.

Sentinel LDK communicates via TCP and UDP on socket 1947. This socket is IANA-registered exclusively for this purpose. By default the client will find the server by issuing a UDP broadcast to local subnets on port 1947.

If broadcasting is not working or unwanted, then advanced network settings can be setup via the Sentinel Admin Control Center. The Sentinel Admin Control Center is accessed by opening the URL `localhost:1947` in a web browser. The network settings are reached by selecting "Configuration" in the menu and then selecting the "Access to Remote License Managers" tab. Detailed information on how to setup the network settings can be found by selecting "Help" in the menu.

Appendix F. Appending environment variables

1. Windows

Open the environment variables dialog by following the steps below:

Windows 7

1. Select `Computer` from the `Start` menu
2. Choose `System Properties` from the context menu
3. Click on `Advanced system settings`
4. Select `Advanced` tab
5. Click on `Environment Variables` button

Windows XP

1. Select `Control Panel` from the `Start` menu
2. Open `System`
3. Select `Advanced` tab
4. Click on `Environment Variables` button

Variables listed under `User variables` will only affect the current user and `System variables` will affect all users. Select the desired variable and press `Edit` to edit the variable value. If the variable does not exist, a new can be created by pressing the button `New`.

To append the `PATH`, find the variable under `System variables` or `User variables` (if the user variable does not exist, then create a new) and press `Edit`. At the end of the value string, append a single semicolon (;) as a separator and then append the desired path, e.g. ;C:\my\path\to\append

2. Linux

Use the `export <name>=<value>` command to set an environment variable. The paths in the variables `PATH` or `LD_LIBRARY_PATH` should be separated with a single colon (:).

To append a path to `PATH` or `LD_LIBRARY_PATH`, add the path to the end of the variable. See example below.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/my/path/to/append
```

Appendix G. Compatibility

Table of Contents

G.1. Compatibility notes for GRMON3	277
G.2. Compatibility notes for GRMON2	277
G.3. Compatibility notes for GRMON1	277

G.1. Compatibility notes for GRMON3

Command `spim/spi/i2c/iommu/svgal4stat/l5stat`

Removed support to add an device index before the subcommand. Add device name at the end to select index. I.e. "cmd index subcommand..." is not supported.

G.2. Compatibility notes for GRMON2

Default startup-behavior

If GRMON4 is started without a debug link option on the command line, then the GRMON4 GUI connection dialog will be opened. Furthermore, if no debug link option is given on the command line, then any other command line options are also ignored. The user can select them in the connection dialog.

If GRMON version 2.0 and earlier is started without an explicit debug link option on the command line, then it will try to connect to the target using the serial debug link by default. The behavior of GRMON version 2.0 and earlier can be achieved in GRMON4 by giving the `-uart` option.

System-specific command-line options

The GRMON 2.0 options

- `-leon2`
- `-at697`
- `-at697e`
- `-at697f`
- `-agga4`

are no longer available. Corresponding options in GRMON4 are:

- `-sys leon2`
- `-sys at697`
- `-sys at697e`
- `-sys at697f`
- `-sys agga4`

Execution hooks

Execution hooks must be installed in the execution shell.

Command `mem/memh/memb`

`-hex/-x` options was removed in version 3.2.1.

G.3. Compatibility notes for GRMON1

Breakpoints

Tcl has a native command called `break`, that terminates loops, which conflicts the the GRMON1 command `break`. Therefore **`break`**, **`hbreak`**, **`watch`** and **`bwatch`** has been replaces by the command **`bp`**.

Cache flushing

Tcl has a native command called `flush`, that flushed channels, which conflicts the the GRMON1 command `flush`. Therefore **`flush`** has been replaced by the command **`ctrl flush`**. In addition the command **`icache flush`** can be used to flush the instruction cache and the command **`dcache flush`** can be used to flush the data cache .

Case sensitivity

GRMON4 command interpreter is case sensitive whereas GRMON1 is insensitive. This is because Tcl is case sensitive.

-eth -ip

-ip flag is not longer required for the Ethernet debug link, i.e. it is enough with *-eth 192.168.0.51*.

Appendix H. Third-party licenses

GRMON incorporates source code and libraries from several Open Source software projects. Therefore the use of these is governed by different Open Source licenses. This appendix provides a list of the Open Source project used and their respective license file.

The license files can be found in `share/grmon/3rdparty` folder of the GRMON installation.

Name	Version	File
TomsFastMath	1.13.0	LICENSE.tfm
LibTomCrypt	1.17.0	LICENSE.ltc
OpenSSL	1.1.0g	LICENSE.openssl
uthash	2.1.0	LICENSE.uthash
Netscape Portable Runtime	4.9.2	LICENSE.nspr
Expat	2.1.1	COPYING.expat
libFTDI	0.20	COPYING.libftdi
Libusb-win32	1.2.6.0	COPYING_LGPL.libusb-win32
Libusb	0.1.12	COPYING.libusb
ncurses	5.9	LICENSE.ncurses
FreeBSD	13.1.0	COPYRIGHT.freebsd
Editline	3.0	COPYING.libedit
Regex	3.8a	COPYRIGHT.regex
eltclsh	1.17	LICENSE.eltclsh
TCL	8.6.10	LICENSE.tcl
Zlib	1.2.3	LICENSE.zlib
TCF Agent Prototype	1.7	LICENSE.tcf-agent
Eclipse.org	Various ¹	LICENSE.eclipse

¹All specific versions of the Eclipse plug-ins can be found in the GUI, if you open menu `Help`, select `About GRMON3`, press `Installation details`

Frontgrade Gaisler AB

Kungsgatan 12
411 19 Göteborg
Sweden
frontgrade.com/gaisler
sales@gaisler.com
T: +46 31 7758650
F: +46 31 421407

Frontgrade Gaisler AB, reserves the right to make changes to any products and services described herein at any time without notice. Consult the company or an authorized sales representative to verify that the information in this document is current before using this product. The company does not assume any responsibility or liability arising out of the application or use of any product or service described herein, except as expressly agreed to in writing by the company; nor does the purchase, lease, or use of a product or service from the company convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual rights of the company or of third parties. All information is provided as is. There is no warranty that it is correct or suitable for any purpose, neither implicit nor explicit.

Copyright © 2025 Frontgrade Gaisler AB