

GRMON User's Manual

Version 1.1.39

January 2010

Copyright 2004-2009 Aeroflex Gaisler AB.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

1	Introduction.....	7
1.1	Overview.....	7
1.2	Supported platforms and system requirements.....	7
1.3	Obtaining GRMON	7
1.4	Installation	7
1.5	GRMON Evaluation version	7
1.6	GrmonRCP	8
1.7	Problem reports.....	8
2	Debugging concept	9
2.1	Overview.....	9
2.2	Target initialization.....	10
2.3	LEON2 target systems.....	12
3	Operation	14
3.1	General.....	14
3.2	Starting GRMON.....	14
3.3	GRMON command-line interface	15
3.4	Common debug operations	16
3.4.1	Loading of files to target memory	16
3.4.2	Running applications	17
3.4.3	Inserting breakpoints and watchpoints	17
3.4.4	Displaying processor registers	18
3.4.5	Displaying memory contents	18
3.4.6	Using the trace buffer	19
3.4.7	Profiling	20
3.4.8	Forwarding application console I/O	21
3.4.8.1	UART debug mode.....	21
3.4.9	Attaching to a target system without initialization	22
3.4.10	Multi-processor support.....	22
3.4.11	Using EDAC protected memory.....	22
3.5	Symbolic debug information	23
3.5.1	Symbol table	23
3.6	GDB interface	24
3.6.1	Attaching to GDB.....	24
3.6.2	Running application in GDB	24
3.6.3	Executing GRMON commands in GDB	25
3.6.4	Detaching.....	25
3.6.5	Specific GDB optimisation.....	25
3.6.6	Limitations of GDB interface	25
3.7	Thread support.....	26
3.7.1	GRMON thread commands	26
3.7.2	GDB thread commands.....	27

4	Debug interfaces	29
4.1	Overview.....	29
4.2	Serial debug interface	29
4.3	Ethernet debug interface	30
4.4	JTAG debug interfaces	30
4.4.1	Xilinx Parallel Cable III or IV	30
4.4.2	Altera USB Blaster or Byte Blaster	31
4.4.3	Xilinx Platform USB Cable (Linux and Windows).....	31
4.4.4	FTDI FT2232 MPSSE-JTAG-emulation mode (Linux).....	31
4.4.5	Choosing JTAG device.....	32
4.5	Direct USB debug interface (Linux and Windows)	32
4.6	PCI debug interface	34
4.7	GRESB debug interface.....	35
4.8	WildCard debug interface.....	35
5	Debug drivers.....	37
5.1	LEON2 and LEON3 debug support unit (DSU) drivers	37
5.1.1	Internal commands.....	37
5.1.2	Command line switches	38
5.2	Memory controller driver	38
5.2.1	Internal commands.....	38
5.2.2	Command line switches	38
5.3	On-chip logic analyser driver (LOGAN).....	39
5.3.1	Internal commands.....	39
5.4	AMBA wrapper for Xilinx System Monitor	41
5.4.1	Internal commands.....	41
5.5	ATA/IDE controller.....	41
5.6	DDR memory controller (DDRSPA).....	42
5.6.1	Command line switches	42
5.7	DDR2 memory controller (DDR2SPA).....	42
5.7.1	Internal commands.....	42
5.7.2	Command line switches	42
5.8	I ² C-master (I2CMST).....	43
5.8.1	Internal commands.....	43
5.9	GRPCI master/target (PCI_MTF).....	43
5.9.1	Internal commands.....	43
5.10	PCIF master/target (PCIF).....	44
5.10.1	Internal commands.....	44
5.11	On-chip PCI trace buffer driver (PCITRACE).....	44
5.12	SPI Controller (SPICTRL)	44
5.13	SPI Memory Controller (SPIMCTRL)	45
5.14	SVGA Frame buffer (SVGACTRL).....	46
5.15	USB Host Controller (GRUSBHC).....	46

5.15.1	Internal commands.....	47
5.15.2	Command line switches.....	47
5.16	AMBA AHB trace buffer driver (AHBTRACE).....	47
5.16.1	Internal Commands.....	47
5.17	10/100 Mbit/s Ethernet Controller (GRETH).....	48
5.17.1	Internal commands.....	48
5.17.2	Command line switches.....	48
5.18	USB 2.0 Device Controller (GRUSBDC).....	48
5.18.1	Internal commands.....	48
5.19	L2-Cache Controller (L2C).....	48
5.19.1	Internal commands.....	49
6	FLASH programming.....	50
6.1	CFI compatible Flash PROM.....	50
6.2	SPI memory device.....	50
7	Error injection.....	52
8	Extending GRMON.....	54
8.1	Loadable command module.....	54
8.2	Custom DSU communications module.....	55
APPENDIX A:	GRMON Command description.....	57
A.1	GRMON built-in commands.....	57
A.2	LEON2/3 DSU commands.....	58
A.3	FLASH programming commands.....	59
APPENDIX B:	License key installation.....	60
B.1	Installing HASP Device Driver.....	60
B.2	Node-locked license file.....	60
APPENDIX C:	Fixed Configuration file format.....	61
APPENDIX D:	JTAG Configuration File.....	62
APPENDIX E:	USB-Blaster Driver setup for Linux.....	63
E.1	Driver Setup on Linux.....	63

1 Introduction

1.1 Overview

GRMON is a general debug monitor for the LEON 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, RS232, PCI, Ethernet and SpaceWire debug links

1.2 Supported platforms and system requirements

GRMON is currently provided for four platforms: Linux-x86, Solaris-2.x, Windows (2K/XP) and Windows with cygwin.

1.3 Obtaining GRMON

The primary site for GRMON is <http://www.gaisler.com/>, where the latest version of GRMON can be ordered and evaluation versions downloaded.

1.4 Installation

GRMON can be installed anywhere on the host computer - for convenience the installation directory should be added to the search path. The commercial versions use a HASP license key on Intel hosts (linux/windows) and a license file on Solaris hosts. See *appendix B* for installation of HASP device drivers and Solaris license files.

1.5 GRMON Evaluation version

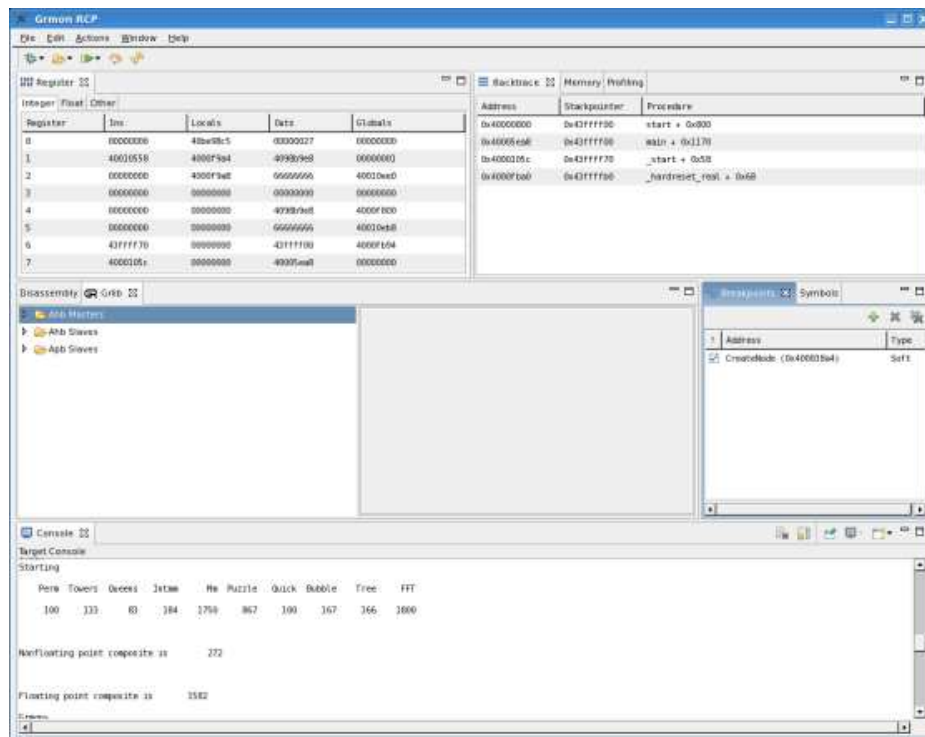
The evaluation version of GRMON can be downloaded from www.gaisler.com. The evaluation version may be used during a period of 21 days without purchasing a license. After this period, any commercial use of GRMON is not permitted without a valid license. The following features are **not** available in the evaluation version:

- Support for LEON2, LEON2-FT, LEON3-FT, LEON4-FT
- Loadable modules
- Custom JTAG configuration files
- Error injection

1.6 GrmonRCP

GrmonRCP is a graphical user interface (GUI) for GRMON, based on the Eclipse Rich Client Platform. It provides a graphical interface to all GRMON functions. GrmonRCP is available as a separate package from <http://www.gaisler.com/>. Below is a screenshot of GrmonRCP in action.

Read more in the online manual at: <http://www.gaisler.com/doc/grmonrpc/html/toc.html>



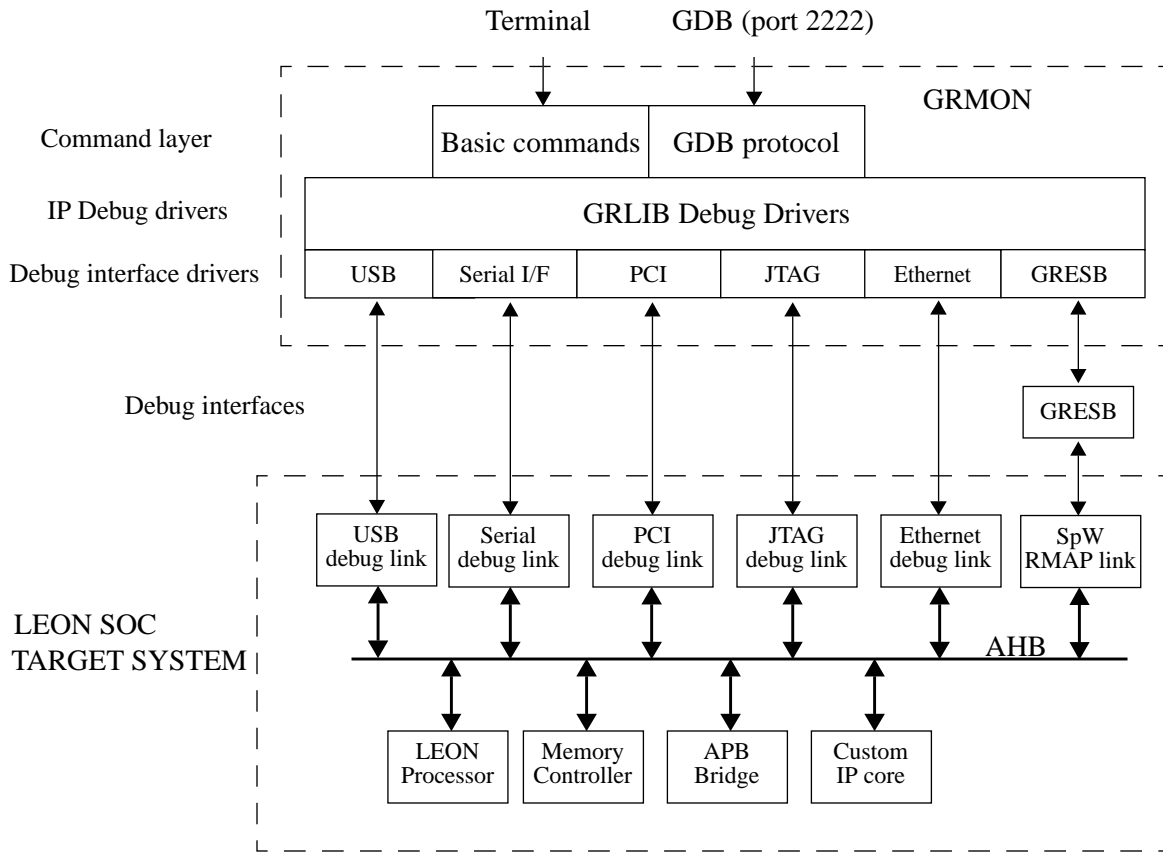
1.7 Problem reports

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

2 Debugging concept

2.1 Overview

The GRMON debug monitor is intended to debug *system-on-chip* (SOC) designs based on the LEON processor. 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 LEON2 processor supports debugging over a serial UART and 32-bit PCI, while LEON3 also supports 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 LEON system, and a target system does in fact not even need to have a processor present.



GRMON can operate in two modes: command-line mode and GDB mode. In command-line mode, GRMON commands are entered manually through a terminal window. In GDB mode, GRMON acts as a GDB gateway and translates the GDB extended-remote protocol to debug commands on the target system.

GRMON is implemented using three functional layers: command layer, debug driver layer, and debug interface layer. The command layer consist of a general command parser which implements commands that are independ-

dent of the used debug interface or target system. These commands include program downloading and flash programming.

The debug driver layer implements custom commands which are related to the configuration of the target system. GRMON scans the target system at startup, and detects which IP cores are present and how they are configured. For each supported IP core, a debug driver is enabled which implements additional debug commands for the specific core. Such commands can consist of memory detection routines for memory controllers, or program debug commands for the LEON processors.

The debug interface layer implements the debug link protocol for each supported debug interface. The protocol depends on which interface is used, but provides a uniform read/write interface to the upper layers. Which interface to use for a debug session is specified through command-line options during the start of GRMON.

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&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:

```
GRMON LEON debug monitor v1.1
```

```
Copyright (C) 2004,2005 Gaisler Research - all rights reserved.  
For latest updates, go to http://www.gaisler.com/  
Comments or bug-reports to support@gaisler.com
```

```
using port /dev/ttyS0 @ 115200 baud  
initialising .....  
detected frequency: 40 MHz
```

Component	Vendor
Leon3 SPARC V8 Processor	Gaisler Research
AHB Debug UART	Gaisler Research
AHB Debug JTAG TAP	Gaisler Research
Simple 32-bit PCI Target	Gaisler Research
AHB interface for 10/100 Mbit MA	Gaisler Research
LEON2 Memory Controller	European Space Agency
AHB/APB Bridge	Gaisler Research
Leon3 Debug Support Unit	Gaisler Research
AHB interface for 10/100 Mbit MA	Gaisler Research
Generic APB UART	Gaisler Research
Multi-processor Interrupt Ctrl	Gaisler Research
Modular Timer Unit	Gaisler Research
Generic APB UART	Gaisler Research

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

```
grmon>
```

More detailed system information can be printed using the 'info sys' command:

```
grmon> inf sys
00.01:003  Gaisler Research  Leon3 SPARC V8 Processor (ver 0)
           ahb master 0
02.01:007  Gaisler Research  AHB Debug UART (ver 0)
           ahb master 2
           apb: 80000700 - 80000800
           baud rate 115200, ahb frequency 40.00
03.01:01c  Gaisler Research  AHB Debug JTAG TAP (ver 0)
           ahb master 3
04.01:012  Gaisler Research  Simple 32-bit PCI Target (ver 0)
           ahb master 4
05.01:005  Gaisler Research  AHB interface for 10/100 Mbit MA (ver 0)
           ahb master 5
00.04:00f  European Space Agency  LEON2 Memory Controller (ver 0)
           ahb: 00000000 - 20000000
           ahb: 20000000 - 40000000
           ahb: 40000000 - 80000000
           apb: 80000000 - 80000100
           32-bit prom @ 0x00000000
           64-bit sdram: 2 * 128 Mbyte @ 0x40000000, col 9, cas 2, ref 15.6 us
01.01:006  Gaisler Research  AHB/APB Bridge (ver 0)
           ahb: 80000000 - 80100000
02.01:004  Gaisler Research  Leon3 Debug Support Unit (ver 0)
           ahb: 90000000 - a0000000
           AHB trace 128 lines, stack pointer 0x4ffffff0
           CPU#0 win 8, hwbp 2, itrace 128, V8 mul/div, srmmu, lddel 1
           icache 2 * 16 kbyte, 32 byte/line lrr
           dcache 2 * 16 kbyte, 32 byte/line lrr
05.01:005  Gaisler Research  AHB interface for 10/100 Mbit MA (ver 0)
           irq 12
           ahb: fffb0000 - fffb1000
01.01:00c  Gaisler Research  Generic APB UART (ver 1)
           irq 2
           apb: 80000100 - 80000200
           baud rate 38400
02.01:00d  Gaisler Research  Multi-processor Interrupt Ctrl (ver 3)
           apb: 80000200 - 80000300
03.01:011  Gaisler Research  Modular Timer Unit (ver 0)
           irq 8
           apb: 80000300 - 80000400
           8-bit scaler, 2 * 32-bit timers, divisor 40
09.01:00c  Gaisler Research  Generic APB UART (ver 1)
           irq 3
           apb: 80000900 - 80000a00
           baud rate 38400
```

The detailed system view also provides information about address mapping, interrupt allocation and IP core configuration.

2.3 LEON2 target systems

The plug&play information was introduced in the LEON3 processor (GRLIB), and is not available in LEON2. If GRMON is not able to detect the plug&play area, it will switch to a LEON2 legacy mode. LEON2 mode can also be forced by starting GRMON with the **-leon2** switch. A LEON2 system has a fixed set of IP cores and address mapping, and GRMON will use an internal plug&play table that describes this configuration:

```
GRMON LEON debug monitor v1.1
```

```
using port /dev/ttyUSB1 @ 115200 baud
GRLIB plug&play not found, switching to LEON2 legacy mode
initialising .....
detected frequency: 40 MHz
```

Component	Vendor
LEON2 Memory Controller	European Space Agency
LEON2 SPARC V8 processor	European Space Agency
LEON2 Configuration register	European Space Agency
LEON2 Timer Unit	European Space Agency
LEON2 UART	European Space Agency
LEON2 UART	European Space Agency
LEON2 Interrupt Ctrl	European Space Agency
AHB Debug UART	Gaisler Research
LEON2 Debug Support Unit	Gaisler Research

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

```
grmon[grlib]> inf sys
00.04:00f European Space Agency LEON2 Memory Controller (ver 0)
    ahb: 00000000 - 20000000
    ahb: 20000000 - 40000000
    ahb: 40000000 - 80000000
    apb: 80000000 - 80000010
    8-bit prom @ 0x00000000
    32-bit sdram: 1 * 64 Mbyte @ 0x40000000, col 9, cas 2, ref 15.6 us
01.04:002 European Space Agency LEON2 SPARC V8 processor (ver 0)
    apb: 80000014 - 80000018
02.04:008 European Space Agency LEON2 Configuration register (ver 0)
    apb: 80000024 - 80000028
    val: 6877bf00
03.04:006 European Space Agency LEON2 Timer Unit (ver 0)
    apb: 80000040 - 80000070
04.04:007 European Space Agency LEON2 UART (ver 0)
    apb: 80000070 - 80000080
    baud rate 38400
05.04:007 European Space Agency LEON2 UART (ver 0)
    apb: 80000080 - 80000090
    baud rate 38400
06.04:005 European Space Agency LEON2 Interrupt Ctrl (ver 0)
    apb: 80000090 - 800000a0
07.01:007 Gaisler Research AHB Debug UART (ver 0)
    apb: 800000c0 - 800000d0
    baud rate 115200, ahb frequency 40.00
08.01:002 Gaisler Research LEON2 Debug Support Unit (ver 0)
    ahb: 90000000 - a0000000
    trace buffer 512 lines, stack pointer 0x43ffff0
    CPU#0 win 8, hwbp 2, V8 mul/div, lddel 1
    icache 1 * 8 kbyte, 32 byte/line
    dcache 1 * 8 kbyte, 32 byte/line
grmon[grlib]>
```

The plug&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 the **-cfg** startup option:

```
./grmon -cfg leon2.cfg
```

```
GRMON LEON debug monitor v1.1
```

```
using port /dev/ttyS0 @ 115200 baud
reading configuration from leon2.cfg
initialising .....
detected frequency: 40 MHz
```

Component	Vendor
AHB Debug UART	Gaisler Research
Generic APB UART	Gaisler Research
LEON2 Interrupt Ctrl	European Space Agency
LEON2 Timer Unit	European Space Agency
LEON2 Memory Controller	European Space Agency
LEON2 Debug Support Unit	Gaisler Research

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

```
grmon[grlib]> inf sys
00.01:007 Gaisler Research AHB Debug UART (ver 0)
          apb: 800000c0 - 800000d0
          baud rate 115200, ahb frequency 40.00
01.01:00c Gaisler Research Generic APB UART (ver 0)
          apb: 80000070 - 80000080
          baud rate 38400
02.04:005 European Space Agency LEON2 Interrupt Ctrl (ver 0)
          apb: 80000090 - 800000a0
03.04:006 European Space Agency LEON2 Timer Unit (ver 0)
          apb: 80000040 - 80000070
04.04:00f European Space Agency LEON2 Memory Controller (ver 0)
          ahb: 00000000 - 20000000
          ahb: 20000000 - 40000000
          ahb: 40000000 - 80000000
          apb: 80000000 - 80000010
          8-bit prom @ 0x00000000
          32-bit sdram: 1 * 64 Mbyte @ 0x40000000, col 9, cas 2, ref 15.6 us
05.01:002 Gaisler Research LEON2 Debug Support Unit (ver 0)
          ahb: 90000000 - a0000000
          trace buffer 512 lines, stack pointer 0x43ffffff
          CPU#0 win 1, lddel 1
              icache 4 * 1 kbyte, 4 byte/line lru
              dcache 4 * 1 kbyte, 4 byte/line lru
grmon[grlib]>
```

The format of the plug&play configuration file is described in section appendix C. It can be used for both LEON3 and LEON2 systems. An example configuration file is also supplied with the GRMON-PRO distribution in src/cfg/leon2.cfg .

3 Operation

3.1 General

A GRMON debug session typically consists of the following steps:

- Attaching to the target system and examining the configuration
- Uploading application program and executing using GRMON commands
- Attaching to GDB and debugging through the GDB protocol

The following sections will describe how the various steps are performed.

3.2 Starting GRMON

GRMON is started by giving the GRMON command in a terminal window. Without options, GRMON will try to connect to the target using the serial debug link. UART1 of the host (ttyS0 or COM1) will be used, with a default baud rate of 115200 baud. On windows hosts, GRMON can be started in a command window (command.com) or in a cygwin shell. Below is the syntax and the accepted command line options:

grmon [*options*]

Options:

-abaud *baudrate*

Set application *baudrate* for UART 1 & 2. By default, 38400 baud is used.

-altcable *nr* Choose which Altera cable to connect to if multiple cables are connected. Connect without **-altcable** to see the cable numbers.

-altjtag Connect to the JTAG Debug Link using Altera USB Blaster or Byte Blaster.

-baud *brate* Use *brate* 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.

-c *batch_file* Run the commands in the batch file at start-up.

-dsudelay *val* Delay the DSU polling *val* ms. Normally GRMON will poll the DSU as fast as possible.

-edac Enable EDAC operation in memory controllers that support it.

-eth Connect using ethernet. Requires the EDCL core to be present in the target system.

-freq *sysclk* Overrides the detected system frequency. The frequency is specified in MHz.

-gdb Listen for GDB connection directly at start-up.

-gresb Connect through the GRESB bridge. The target needs SpW core with RMAP.

-ioarea *addr* Specify the location of the I/O area. (Default is 0xFFFF0000)

-jtag Connect to the JTAG Debug Link using Xilinx Parallel Cable III or IV.

-jtagcfg *file* Read custom JTAG Configuration file.

-jtagdevice *nr* Choose which JTAG device to debug.

-leon2 Force LEON2 legacy mode.

-log *file* Log session to the specified file. If the file already exists the new session is appended. This should be used when requesting support.

-ni Attach to running program without target initialisation.

- nothreads** Disable thread support.
- pci *vid:did[:i]*** Connect to target using PCI. The board is identified by vendor id, device id and optionally instance number.
- port *gdbport*** Set the port number for GDB communications. Default is 2222.
- rtems *ver*** Override autodetected RTEMS version for thread support. *ver* should be 46 or 48.
- stack *val*** Set *val* as stack pointer for applications, overriding the auto-detected value.
- 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 3.4.8 "Forwarding application console I/O" on page 21. 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 busscan. If the device parameter is not used the first UART is selected.
- 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.
- ucmd *file*** Load a user command module.
- usb** Connect to target using USB (GRUSB IP core).
- wildcard** Connect to a WildCard PC Card via CardBus (Windows only).
- xilusb** Connect to the JTAG Debug Link using Xilinx Platform USB cable.

In addition, the debug drivers can also accept command line options.

3.3 GRMON command-line interface

GRMON dynamically loads `libreadline.so` if available on your host system, and uses `readline()` to enter and edit monitor commands. Short forms of the commands are allowed, e.g **c**, **co**, or **con**, are all interpreted as **cont**. Tab completion is available for commands, text-symbols and filenames. If `libreadline.so` is not found, `fgets()` is used instead (no history, poor editing capabilities and no tab-completion). Below is a description of some of the more common commands that are available regardless of loaded debug drivers. For the full list of commands, see appendix A.1.

- | | |
|--|---|
| batch <i>file_name</i> | execute a batch file of GRMON commands |
| disas <i><addr></i> [<i>length</i>] | disassemble memory |
| dump <i><addr></i> <i><addr></i> [<i>file</i>] | dump target memory to file in srecord format |
| echo | echo string in monitor window |
| eeload <i><file></i> [<i>delay</i>] | Load PROM file using the delay in milliseconds between each word.
Suitable for loading PROM files to E ² PROM. |
| help | show available commands or usage for specific command |
| info [drivers libs reg sys] | show available debug drivers, system registers or system configuration |
| load <i><file_name></i> | load a file into target memory (elf32 or srecord). |
| log <i><file_name></i> | Open a logfile and output some system information to it then continue to log the current session to it. This command or the -log start-up option |

	should be used when requesting support. If the start-up option <code>-log</code> is given then <code>file_name</code> can be omitted.
mem [<i>addr</i>] [<i>length</i>]	display memory
symbols	show symbols or load symbols from file
quit	exit GRMON
wmem < <i>addr</i> > < <i>data</i> >	write word to memory

Below is a list of some of commands provided by the LEON debug support unit (DSU) debug driver. These commands are available when a LEON processor and associated debug support unit is present in the target system. See appendix A.2 for a full list of DSU commands.

break < <i>addr</i> >	print or add breakpoint
bwatch < <i>addr</i> > [<i>delay</i>] [break]	add bus watchpoint
cont	continue execution
dcache	show data cache
delete < <i>bptnum</i> >	delete breakpoint(s)
float	display FPU registers
gdb [<i>port</i>]	connect to GDB debugger
go [<i>addr</i>]	start execution without initialization
hbreak	print breakpoints or add hardware breakpoint (if available)
icache	show instruction cache
profile [0 1]	enable/disable/show simple profiling
register	show/set integer registers
run [<i>addr</i>]	reset and start execution at last entry point, or at <i>addr</i>
tmode [none ahb cpu both]	enable instruction and AHB trace buffers
stack [<i>val</i>]	show/set the stack pointer
step [<i>n</i>]	single step one or [<i>n</i>] times
wash	clear all SRAM/SDRAM memory
watch [<i>addr</i>]	print or add data watchpoint

3.4 Common debug operations

3.4.1 Loading of files to target memory

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

```
grmon> load stanford_leon
section: .text at 0x40000000, size 54368 bytes
section: .data at 0x4000d460, size 2064 bytes
section: .jcr at 0x40024e68, size 4 bytes
total size: 56436 bytes (90.9 kbit/s)
read 196 symbols
entry point: 0x40000000
```

The supported file format is elf32-sparc and srecord. Each section is loaded to its link address. The program entry point of the file is used to set the %pc when the application is later started with **run**. It is also possible to verify that the file has been loaded correctly using the **verify** command:

```
grmon> veri stanford_leon
section: .text at 0x40000000, size 54368 bytes
section: .data at 0x4000d460, size 2064 bytes
section: .jcr at 0x40024e68, size 4 bytes
total size: 56436 bytes (64.9 kbit/s)
entry point: 0x40000000
```

Any discrepancies will be reported in the GRMON console.

3.4.2 Running applications

To run a program, first use the **load** command to download the application and the **run** to start it. The program should have been compiled with either the BCC, RCC or sparc-linux tool-chain.

```
grmon> lo stanford_leon
section: .text at 0x40000000, size 54368 bytes
section: .data at 0x4000d460, size 2064 bytes
section: .jcr at 0x40024e68, size 4 bytes
total size: 56436 bytes (90.8 kbit/s)
read 196 symbols
entry point: 0x40000000
grmon> 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

Program exited normally.
grmon>
```

The output from the application normally appears on the LEON UARTs and thus not in the GRMON console. However, if GRMON is started with the **-u** switch, the UART output is looped back to its own receiver and printed on the console by GRMON. The application must be reloaded before it can be executed again, in order to restore the .data segment. If the application uses the LEON MMU (e.g. linux-2.6) or installs data exception handlers (e.g. eCos), the GRMON should be started with **-nb** to avoid going into break mode on a page-fault or data exception. Note that the **-u** option does not work for snapgear linux applications. Instead, a terminal emulator should be connected to UART 1 of the target system. When running a debugger on the target the *"ta 0x01"* instruction will be used to set a breakpoint. To prevent GRMON from interpreting it as its own breakpoints and stop use the **-nswb** switch.

3.4.3 Inserting breakpoints and watchpoints

Instruction breakpoints are inserted using the **break** or **hbreak** commands. The **break** command inserts a software breakpoint (ta 1), while **hbreak** will insert a hardware breakpoint using one of the IU watchpoint registers. To debug code in read-only memories, only hardware breakpoints can be used. Note that it is possible to debug any RAM-based code using software breakpoints, even where traps are disabled such as in trap handlers.

Data write watchpoints are inserted using the **watch** command. A watchpoint can only cover one word address, block watchpoints are not supported by GRMON.

The **bwatch** command inserts bus watchpoints that will freeze the trace buffer when hit. The processor can optionally be put in debug mode when the bus watchpoint is hit. This is controlled using the **tmode** command:

tmode *ahbbre* *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.

3.4.4 Displaying processor registers

The current register window of a LEON processor can be displayed using the **reg** command:

```

grmon> reg

      INS      LOCALS      OUTS      GLOBALS
0:  00000008  0000000C  00000008  00000000
1:  80000070  00000020  80000070  00000008
2:  0000000D  43FFFDf0  0000000D  43FFF6F8
3:  FFFFFFFF  00000003  FFFFFFFF  4000D010
4:  43FFF7B8  00000001  43FFF7B8  00000001
5:  4000D008  00000004  4000D008  00000000
6:  43FFF618  00000000  43FFF618  00000000
7:  00000001  00000010  00000001  4000633C

psr: F20000E2   wim: 00000080   tbr: 40000060   y: 00000000

pc: 40003e44   be 0x40003fb8
npc: 40003e48   mov %i1, %i3

```

Other register windows can be displayed using **reg *wn***, when *n* denotes the window number. Use the **float** command to show the FPU registers (if present).

3.4.5 Displaying memory contents

Any memory location can be displayed using the **mem** (or **x**) command. The command requires an address and an optional length. If a length argument is provided, that is interpreted as the number of bytes to display. If a program has been loaded, text symbols can be used instead of a numeric address. The memory content is displayed hexa-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  ... )....Å .....
40000010  91d02000  01000000  01000000  01000000  . .....
40000020  91d02000  01000000  01000000  01000000  . .....
40000030  91d02000  01000000  01000000  01000000  . .....

grmon> mem 0x40000000 16

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

grmon> mem main 48

40003278  9de3bf98  2f100085  31100037  90100000  .ä¿./...1..7....
40003288  d02620c0  d025e178  11100033  40000b4b  & Å%âx...3@..K
40003298  901223b0  11100033  40000af4  901223c0  ..#*...3@..ô..#À

```

If the memory contents is SPARC machine code, the contents can be displayed in assembly code using the **disas** command:

```
grmon> dis 0x40000000 10
40000000 a0100000 clr %l0
40000004 29100004 sethi %hi(0x40001000), %l4
40000008 81c52000 jmp %l4
4000000c 01000000 nop
40000010 91d02000 ta 0x0
40000014 01000000 nop
40000018 01000000 nop
4000001c 01000000 nop
40000020 91d02000 ta 0x0
40000024 01000000 nop
```

```
grmon> dis main
40003278 9de3bf98 save %sp, -104, %sp
4000327c 2f100085 sethi %hi(0x40021400), %l7
40003280 31100037 sethi %hi(0x4000dc00), %i0
40003284 90100000 clr %o0
40003288 d02620c0 st %o0, [%i0 + 0xc0]
4000328c d025e178 st %o0, [%l7 + 0x178]
40003290 11100033 sethi %hi(0x4000cc00), %o0
40003294 40000b4b call 0x40005fc0
40003298 901223b0 or %o0, 0x3b0, %o0
4000329c 11100033 sethi %hi(0x4000cc00), %o0
400032a0 40000af4 call 0x40005e70
400032a4 901223c0 or %o0, 0x3c0, %o0
```

3.4.6 Using the trace buffer

The LEON processor and associated debug support unit (DSU) 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 startup, but can also be individually enabled and disabled using **tmode** command. The commands **ahb**, **inst** and **hist** are used to display the contents of the buffers. Below is an example debug session that shows the usage of breakpoints, watchpoints and the trace buffer:

```
grmon> load samples/stanford
section: .text at 0x40000000, size 41168 bytes
section: .data at 0x4000a0d0, size 1904 bytes
total size: 43072 bytes (94.2 kbit/s)
read 158 symbols
grmon> tm both
combined processor/AHB tracing
grmon> break Fft
grmon> watch 0x4000a500
grmon> bre
num  address      type
  1 : 0x40003608  (soft)
  2 : 0x4000a500  (watch)
grmon> run
watchpoint 2 free + 0x1c8 (0x400042d0)
grmon> ahh
time  address  type  data  trans size burst mst lock resp  tt  pil  irl
239371467 400042d8 read 38800002 3 2 1 0 0 0 0 06 0 0
239371469 400042dc read d222a100 3 2 1 0 0 0 0 0 06 0 0
239371472 4000a4fc read 00000000 2 2 0 0 0 0 0 0 06 0 0
239371480 4000a4fc write 000005d0 2 2 0 0 0 0 0 0 06 0 0
239371481 90000000 read 000055f9 2 2 0 3 0 0 0 0 06 0 0
```

```

grmon> inst
    time      address  instruction      result
239371473  400042bc  ld [%o2 + 0xfc], %o0  [00000000]
239371475  400042c0  cmp %o1, %o0        [000005d0]
239371476  400042c4  bgu,a 0x400042cc     [00000000]
239371478  400042c8  st %o1, [%o2 + 0xfc] [4000a4fc 000005d0]
239371479  400042cc  sethi %hi(0x4000a400), %o2 [4000a400]
grmon> del 2
grmon> break
num  address      type
  1 : 0x40003608  (soft)
grmon> cont
breakpoint 1 Fft (0x40003608)
grmon> hist
254992755  40003870  sethi %hi(0x4001f800), %i0 [4001f800]
254992759                ahb read, mst=0, size=2  [40003880 94146198]
254992760  40003874  mov 19, %i0          [00000013]
254992761                ahb read, mst=0, size=2  [40003884 961423cc]
254992762  40003878  mov 256, %o0         [00000100]
254992763                ahb read, mst=0, size=2  [40003888 190fec00]
254992764  4000387c  or %i2, 0x28c, %o1   [40014e8c]
254992765                ahb read, mst=0, size=2  [4000388c 7fffffff5f]
254992766  40003880  or %i1, 0x198, %o2   [40014598]
254992767                ahb read, mst=0, size=2  [40003890 9a102000]
254992769                ahb read, mst=0, size=2  [40003894 b0863fff]
254992771  40003884  or %i0, 0x3cc, %o3   [4001fbcc]
254992772  40003888  sethi %hi(0x3fb00000), %o4 [3fb00000]
254992773  4000388c  call 0x40003608     [4000388c]
254992774  40003890  mov 0, %o5          [00000000]

```

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 appear within brackets. The time indicates when the transfer completed, i.e. when HREADY was asserted. Note: when switching between tracing modes the contents of the trace buffer will not be valid until execution has been resumed and the buffer refilled.

3.4.7 Profiling

GRMON supports profiling of LEON applications when run on real hardware. The profiling function collects (statistical) information on the amount of execution time spend 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.

```

grmon> profile 1
Profiling enabled
grmon> run
resuming at 0x40000000
Starting
  Perm  Towers  Queens  Intmm    Mm  Puzzle  Quick  Bubble  Tree  FFT
    50    33    17    116   1100   217    33    34    266   934

Nonfloating point composite is      126

Floating point composite is      862

Program exited normally.

```

```
grmon> prof

function          samples      ratio(%)
__unpack_f        23627       16.92
__mulsf3          22673       16.24
__pack_f          17051       12.21
__divdi3          14162       10.14
.umul             8912        6.38
Fit               7594        5.44
__muldi3          6453        4.62
_window_overflow  3779        2.70
Insert           3392        2.42
__addsf3          3327        2.38
_window_underflow 2734        1.95
__subsf3          2409        1.72
Fft               2207        1.58
start            2165        1.55
Innerproduct     2014        1.44
Bubble           1767        1.26
rInnerproduct    1443        1.03
Place            1371        0.98
Remove           1335        0.95
Try              1275        0.91
Permute          1125        0.80
```

NOTE: profiling is not supported in systems based on the first-generation LEON2-FT processors, such as LEON2FT-UMC and AT697E.

3.4.8 Forwarding application console I/O

If GRMON is started with **-u**, the LEON UART1 will be placed in FIFO debug mode if supported by the hardware. If not loop-back mode will be used instead. In both modes flow-control will be enabled.

Debug mode was added in GRLIB 1.0.17-b2710, and will be reported by **info sys** in GRMON as `DSU mode (FIFO debug)`.

Both in loop-back mode and in FIFO debug mode the UART will be polled regularly during execution of an application and all console output will be printed on the GRMON console. It is then not necessary to connect a separate terminal to UART1 to see the application output.

With FIFO debug mode it will also be possible to enter text in GRMON which will be 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 O/S such as Linux.

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

- The UART control register must not be modified such that neither loop-back nor FIFO debug mode is disabled
- 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).

Note: 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.

3.4.8.1 UART debug mode

When the application is running with UART debug mode enabled the following key sequences will be available.

- 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

These can be used to adjust the input to what the target system expects.

3.4.9 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 resume debugging without reloading the application, 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 system information print-out (info sys) will then however not be able to display the correct memory settings. The use of the **-stack** option is also necessary in case the application is later restarted.

3.4.10 Multi-processor support

In systems with more than one LEON3 processors, 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, the processor will be started when any of the **run**, **cont** or **go** commands are given. When disabled, the processor will remain halted regardless of which command that are given. In addition, one of the enabled processor will also be *active*. All debugging commands such as displaying registers or adding break points will be directed to the active processor only. Switching of active processor can be done using the '**cpu active n**' command

At start-up, processor 0 is enabled and active while remaining processors are disabled. This allows non-MP software to execute on MP systems. Additional processors can be enabled using the '**cpu enable n**' command:

```
grmon[grlib]> cpu
cpu 0: enabled active
cpu 1: disabled
grmon[grlib]> cpu en 1
cpu 0: enabled active
cpu 1: enabled
grmon[grlib]> cpu act 1
cpu 0: enabled
cpu 1: enabled active
```

If GRMON is started with the **-mp** switch, all processors will be enabled by default.

Breakpoints are maintained on processor basis. When executing, only the breakpoints of the active processor are enabled, the breakpoints of the other processors are not enabled or inserted into the main memory.

It is possible to debug MP systems using GDB. When GDB is attached, it is the currently active processor that will receive the GDB commands. Switching of processor can only be done by detaching GDB, selecting a different processor to become active, and the re-attaching GDB. Note that GDB remembers the breakpoints between detach and re-attachment.

3.4.11 Using EDAC protected memory

Some versions of LEON2FT or LEON3FT might use EDAC protected memory. To enable the memory EDAC during execution, start GRMON with the *-edac* switch. Before any application is loaded, issue the **wash** command to write all RAM memory locations and thereby initialize the EDAC checksums.

```
$ grmon -ramws 1 -edac

gplib> wash
clearing 191 kbyte SRAM: 40000000 - 4002ffff
clearing 1024 kbyte SDRAM: 60000000 - 60100000
gplib>
```

By default wash writes to the whole memory area which has been detected or forced with a command line switch. A parameter can also be given which sets the amount of memory in bytes to be washed for SRAM (SDRAM always uses the detected size of the memory).

```
$ grmon -ramws 1 -edac

gplib> wash 65536
clearing 64 kbyte SRAM: 40000000 - 40010000
clearing 1024 kbyte SDRAM: 60000000 - 60100000
gplib>
```

If the memory controller has support for EDAC with 8-bit wide SRAM memory, the upper part of the memory will consist of checkbits. In this case the wash will only write to the data area (the checkbits will automatically be written by the memory controller). The amount of memory written will be displayed in GRMON.

3.5 Symbolic debug information

3.5.1 Symbol table

GRMON will automatically extract the symbol information from elf-files. The symbols can be used where an address is expected:

```
grmon> break main
grmon> run
breakpoint 1 main (0x40001ac8)
grmon> disas strlen 3
40001e4c 808a2003 andcc %o0, 0x3, %g0
40001e50 12800016 bne 0x40001ea8
40001e54 96100008 mov %o0, %o3
```

The **symbols** command can be used to display all symbols, or to read in symbols from an alternate (elf) file:

```
grmon[dsu]> symbols samples/hello
read 71 symbols
grmon[dsu]> symbols
0x40000000 L _trap_table
0x40000000 L start
0x4000102c L _window_overflow
0x40001084 L _window_underflow
0x400010dc L _fpdis
```

```
0x400011a4 T _flush_windows  
0x400011a4 T _start  
0x40001218 L fstat
```

Reading symbols from alternate files is necessary when debugging self-extracting applications, such as boot-proms created with mkprom or linux/uClinux.

3.6 GDB interface

3.6.1 Attaching to GDB

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** command:

```
$ grmon -gdb

using port /dev/ttyS0 @ 115200 baud

gdb interface: using 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:

```
(gdb) target extended-remote pluto:2222
Remote debugging using pluto:2222
0x40000800 in start ()
(gdb)
```

3.6.2 Running application in GDB

To load and start an application, use the GDB **load** and **run** command.

```
$ gdb stanford
(gdb) target extended-remote pluto:2222
Remote debugging using pluto:2222
0x40000800 in start ()
(gdb) load
Loading section .text, size 0xcb90 lma 0x40000000
Loading section .data, size 0x770 lma 0x4000cb90
Start address 0x40000000, load size 54016
Transfer rate: 61732 bits/sec, 278 bytes/write.
(gdb) bre main
Breakpoint 1 at 0x400039c4: 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/john/samples/stanford

Breakpoint 1, main () at stanford.c:1033
1033         fixed = 0.0;
(gdb)
```

To interrupt execution, Ctrl-C can be typed in both GDB and GRMON windows. 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 1) is used by GDB to insert breakpoints and should not be used by the application.

3.6.3 Executing GRMON commands in GDB

While GDB is attached to GRMON, normal GRMON commands can be executed using the GDB **monitor** command. Output from the GRMON commands is then displayed in the GDB console:

```
(gdb) monitor hist
time      address  instruction  result
4484188   40001e90  add %g2, %o2, %g3  [6e1f766e]
4484194   40001e94  andn %g3, %g2, %g2  [001f0000]
4484195   40001e98  andcc %g2, %o0, %g0  [00000000]
4484196   40001e9c  be,a 0x40001e8c     [40001e3c]
4484197   40001ea0  add %o1, 4, %o1     [40003818]
4484198   40001e8c  ld [%o1], %g2       [726c6421]
4484200   40001e90  add %g2, %o2, %g3  [716b6320]
4484201   40001e94  andn %g3, %g2, %g2  [01030300]
4484202   40001e98  andcc %g2, %o0, %g0  [00000000]
4484203   40001e9c  be,a 0x40001e8c     [40001e3c]
```

3.6.4 Detaching

If GDB is detached using the **detach** command, the monitor returns to the command prompt, and the program can be debugged using the standard GRMON commands. The monitor can also be re-attached to GDB by issuing the **gdb** command to the monitor (and the **target** command to GDB).

GRMON translates SPARC traps 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.

3.6.5 Specific GDB optimisation

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.

GRMON detects the insertion of GDB breakpoints, in form of the *'ta I'* instruction. When a breakpoint is inserted, the corresponding instruction cache tag is examined, and if the memory location was cached the tag is cleared to keep memory and cache synchronized.

3.6.6 Limitations of GDB interface

For optimal operation, GDB 6.4 configured for GRMON should be used (provided with RCC and BCC compilers).

Do not use the GDB **where** command in parts of an application where traps are disabled (e.g.trap handlers). Since the stack pointer is not valid at this point, GDB might go into an infinite loop trying to unwind false stack frames.

3.7 Thread support

GRMON has thread support for the RTEMS, VXWORKS and eCos operating systems. The GDB interface of GRMON is also thread aware and the related GDB commands are described later.

3.7.1 GRMON thread commands

thread info - lists all known threads. The currently running thread is marked with an asterisk.

```
grrlib> thread info
```

Name	Type	Id	Prio	Ticks	Entry point	PC	State
Int.	internal	0x09010001	255	138	_CPU_Thread_Idle_body	0x4002f760 _Thread_Dispatch + 0x11c	READY
UI1	classic	0x0a010001	120	290	Init	0x4002f760 _Thread_Dispatch + 0x11c	READY
ntwk	classic	0x0a010002	100	11	rtems_bsdnet_schedneti	0x4002f760 _Thread_Dispatch + 0x11c	READY
DCrx	classic	0x0a010003	100	2	rtems_bsdnet_schedneti	0x4002f760 _Thread_Dispatch + 0x11c	Wevnt
Dctx	classic	0x0a010004	100	4	rtems_bsdnet_schedneti	0x4002f760 _Thread_Dispatch + 0x11c	Wevnt
FTPa	classic	0x0a010005	10	1	split_command	0x4002f760 _Thread_Dispatch + 0x11c	Wevnt
FTPD	classic	0x0a010006	10	1	split_command	0x4002f760 _Thread_Dispatch + 0x11c	Wevnt
* HTPD	classic	0x0a010007	40	79	rtems_initialize_webse	0x40001b60 console_outbyte_polled + 0x34	READY

thread bt <id> - do a backtrace of a thread.

Backtrace of inactive thread:

```
grrlib> 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 normal bt command):

```
grrlib> 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
```

3.7.2 GDB thread commands

GRMON needs the symbolic information of the image that is being debugged to be able to check for thread information. Therefore the symbols need to be read from the image using the **symbol** command before issuing the **gdb** command.

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.

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

0x40001b5c in console_outbyte_polled (port=0, ch=113 'q') at ../../../../../../rtems-4.6.5/c/src/lib/
libbsp/sparc/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-4.6.5/cpukit/score/
src/threaddispatch.c:109
   7 Thread 167837701 (FTPa Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/
src/threaddispatch.c:109
   6 Thread 167837700 (DCtx Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/
src/threaddispatch.c:109
   5 Thread 167837699 (DCrx Wevnt) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/
src/threaddispatch.c:109
   4 Thread 167837698 (ntwk ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/
src/threaddispatch.c:109
   3 Thread 167837697 (UI1 ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/
src/threaddispatch.c:109
   2 Thread 151060481 (Int. ready) 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/
src/threaddispatch.c:109
*  1 Thread 167837703 (HTPD ready) 0x40001b5c in console_outbyte_polled (port=0, ch=113 'q')
   at ../../../../../../rtems-4.6.5/c/src/lib/libbsp/sparc/leon3/console/debugputs.c:38
```

Using the **thread** command a specified thread can be selected:

```
(gdb) thread 8

[Switching to thread 8 (Thread 167837702)]#0 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/
cpukit/score/src/threaddispatch.c:109
109     _Context_Switch( &executing->Registers, &heir->Registers );
```

Then a backtrace of the selected thread can be printed using the **bt** command:

```
(gdb) bt

#0 0x4002f760 in _Thread_Dispatch () at ../../../../../../rtems-4.6.5/cpukit/score/src/threaddispatch.c:109
#1 0x40013ee0 in rtems_event_receive (event_in=33554432, option_set=0, ticks=0, event_out=0x43feccl4)
   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=0x43feccl4)
   at ../../../../../../rtems-4.6.5/cpukit/libnetworking/rtems/rtems_glue.c:641
#3 0x40027548 in socnnsleep (so=0x43f0cd70) at ../../../../../../rtems-4.6.5/cpukit/libnetworking/rtems/
rtems_glue.c:465
#4 0x40029118 in accept (s=3, name=0x43feccf0, namelen=0x43feccec) at ../../../../../../rtems-4.6.5/cpukit/
libnetworking/rtems/rtems_syscall.c:215
#5 0x40004028 in daemon () at ../../../../../../rtems-4.6.5/c/src/libnetworking/rtems_servers/ftpd.c:1925
#6 0x40053388 in _Thread_Handler () at ../../../../../../rtems-4.6.5/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-4.6.5/cpukit/libnetworking/libc/res_mkquery.c:199
#8 0x00000008 in ?? ()
#9 0x00000008 in ?? ()
Previous frame identical to this frame (corrupt stack?)
```

It is possible to use the **frame** command to select a stack frame of interest and examine the registers using the **info registers** command. Note that the **info registers** command only can see the following registers for an inactive task: g0-g7, l0-l7, i0-i7, o0-o7, pc and psr. The other registers will be displayed as 0:

```
(gdb) frame 5
#5 0x40004028 in daemon () at ../../../../../../rtems-4.6.5/c/src/libnetworking/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          0xf34000e4 -213909276
o5          0x4007cc00 1074252800
sp          0x43fecc88 0x43fecc88
o7          0x40004020 1073758240
l0          0x4007ce88 1074253448
l1          0x4007ce88 1074253448
l2          0x400048fc 1073760508
l3          0x43feccf0 1140772080
l4          0x3      3
l5          0x1      1
l6          0x0      0
l7          0x0      0
i0          0x0      0
i1          0x40003f94 1073758100
i2          0x0      0
i3          0x43ffaafc8 1140830152
i4          0x0      0
i5          0x4007cd40 1074253120
fp          0x43feccd08 0x43feccd08
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.

4 Debug interfaces

4.1 Overview

The default communications interface between GRMON and the target system is the host's serial port connected to the AHB uart of the target system. Connecting using USB, JTAG, PCI or ethernet can be performed using the switches listed below:

- eth** Connect using ethernet. Requires the EDCL core to be present in the target system.
- pci** *vid:did[:i]* Connect through PCI. Board is identified by vendor id, device id and optionally instance number. Requires a supported PCI core.
- jtag** Connect to the JTAG Debug Link using Xilinx Parallel Cable III or IV.
- altjtag** Connect to the JTAG Debug Link using Altera download cable (USB or parallel).
- gresb** Connect through the GRESB bridge. The target needs a SpW core with RMAP.
- xilusb** Connect to the JTAG Debug Link using Xilinx Platform USB cable.
- usb** Connect to the USB debug link. Requires the USBDCCL core to be present in the target.
- wildcard** Connect to WildCard PC Card. Requires the WILD2AHB core to be present in target.
- ftdi** Connect to FTDI FT2232 chip in MPSSE-JTAG-emulation mode (linux only).

4.2 Serial debug interface

To successfully attach GRMON using the AHB uart, first connect the serial cable between the uart connectors on target board and the host system. Then power-up and reset the target board and start GRMON. Use the **-uart** option in case the target is not connected to the first uart port of your host. Below is a list of start-up switches applicable for the AHB uart interface:

- baud** *baudrate* Use *baudrate* 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.
- ibaud** *baudrate* Use *baudrate* to determine the target processor frequency. Lower rate means higher accuracy. The detected frequency is printed on the console during startup. By default, 115200 baud is used.
- 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 unix systems (and cygwin), serial devices are named as /dev/ttyXX. On windows, use com1 - 4.

When GRMON connects to the target with the serial interface, the system clock frequency is calculated by comparing the setting in the AHB uart baud rate generator to the used communications baud rate. This detection has limited accuracy, but can be improved by selecting a lower detection baud rate using the **-ibaud** switch. 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.

4.3 Ethernet debug interface

If the target system uses the EDCL ethernet communication link core, GRMON can connect to the system using ethernet. In this case, start GRMON with **-eth**. The default network parameters can be set through additional switches:

- emem** *<size>* Use *size* for the target system's EDCL packet buffer. Default is 2 (kbytes)
- ip** *<ipnum>* Use *ipnum* for the target system IP number. Default is 192.168.0.51.
- udp** *<port>* Use *port* for the target system UDP port. Default is 8000.

The IP address of the EDCL is determined at synthesis time, but can be changed using the **edcl** command:

```
grlib> edcl
      edcl ip 192.168.0.51, buffer 2 kbyte

grlib> edcl 192.168.0.56
      edcl ip 192.168.0.56, buffer 2 kbyte
```

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 attach GRMON using the EDCL with the new IP number. If the **edcl** command is issued through the EDCL interface, GRMON must be re-started using the new IP address of the EDCL interface. The current IP number is also visible in 'info sys' :

```
grlib> info sys
.
.
03.01:01d  Gaisler Research  GR Ethernet MAC (ver 0)
          ahb master 3
          apb: 80000b00 - 80000c00
          edcl ip 192.168.0.52, buffer 2 kbyte
```

4.4 JTAG debug interfaces

4.4.1 Xilinx Parallel Cable III or IV

If target system has the JTAG AHB debug interface, 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. On linux systems, the GRMON binary has to be owned by the superuser (root) and have 's' (set user or group ID on execution) permission bit set (chmod +s grmon). GRMON will report the devices in the JTAG chain. If an unknown device is found, initialization of the JTAG chain will fail and GRMON will report the JTAG ID of the unknown device. In this case, use JTAG configuration file to describe the unknown device (see appendix D). If you report the device ID and corresponding JTAG instruction register length to Aeroflex Gaisler and the device will be supported in a future release of GRMON. Following switch can be used with the JTAG debug interface:

- pport** *<1,2>* Use parallel port 1 or 2 (default is 1).

```
$ grmon -jtag -u

using JTAG cable on parallel port
JTAG chain: xc3s1500 xcf04s xcf04s
```

4.4.2 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. GRMON will automatically detect cable connected to the host computer. On Linux systems, the path to Quartus shared libraries has to be defined in the LD_LIBRARY_PATH environment variable, e.g.:

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

GRMON LEON debug monitor v1.1.16c
```

Note that the automatic frequency detection over USB is not always working due to unpredictable communication latencies. It is recommended to use the **-freq xx** option to define the target frequency when using the Altera USB cable.

4.4.3 Xilinx Platform USB Cable (Linux and Windows)

JTAG debugging using the Xilinx USB Platform cable is supported on Linux and Windows systems. The platform cable models DLC9G and DLC10 are supported. The legacy model DLC9 is not supported. GRMON should be started with **-xilusb** switch, e.g.:

```
$ grmon -xilusb -u

GRMON LEON debug monitor v1.1.35

Xilinx cable: Cable type/rev : 0x3
JTAG chain: xc2v6000 xc18v04 xc18v04 xc18v04 xc18v04 xc18v04 xc18v04
GRLIB build version: 2384

initialising .....
```

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

On Windows hosts, the USB cable drivers should be installed from ISE or ISE-Webpack. Xilinx ISE 9.2i or later is required. Then, the filter driver from the libusb project (<http://libusb-win32.sourceforge.net>, version 0.1.12.1 or later is recommended) should be installed.

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. Tested boards are Digilent XUP and XC3S1600E Development boards. Feed-back on other boards is welcome.

If multiple Xilinx Platform USB Cables are connected the **-xilport <num>** switch can be added to select one, where `<num>` is an index starting from 0. By adding **-xillist** you can get a list of connected USB devices and the **-xilport <num>** combination to use.

4.4.4 FTDI FT2232 MPSSE-JTAG-emulation mode (Linux)

JTAG debugging using a FTDI FT2232/FT4232 chip in MPSSE-JTAG-emulation mode is supported when grmon is started with the **-ftdi** switch. As with parallel port JTAG cables, the GRMON binary has to be owned by the superuser (root) and have `'s'` (set user or group ID on execution) permission bit set (`chmod +s grmon`).

Extra options to the FTDI interface:

- ftdifreq <value>** Set the JTAG frequency divider. <value> can be in the range of 0-0xffff. The JTAG frequency will be approximately 6 MHz / (value + 1). Default is 0x05 (1 MHz).
- ftdipid <hex>** Specify the device ID (DID) of the chip to use. Default is 0x6010 (FT2232H in the Gaisler/Pender USB/JTAG cable). In linux you can lookup the VID:DID pair by typing /sbin/lusb. The VID is FTDI's 0x0403. Other supported device are FT2232L (0xcff8) and FT4232H (0x6011).

4.4.5 Choosing JTAG device

If the JTAG device chain contains more than one FPGA GRMON will not know which device to debug. To choose device manually use the option **-jtagdevice X** where X is the device number to connect to.

4.5 Direct USB debug interface (Linux and Windows)

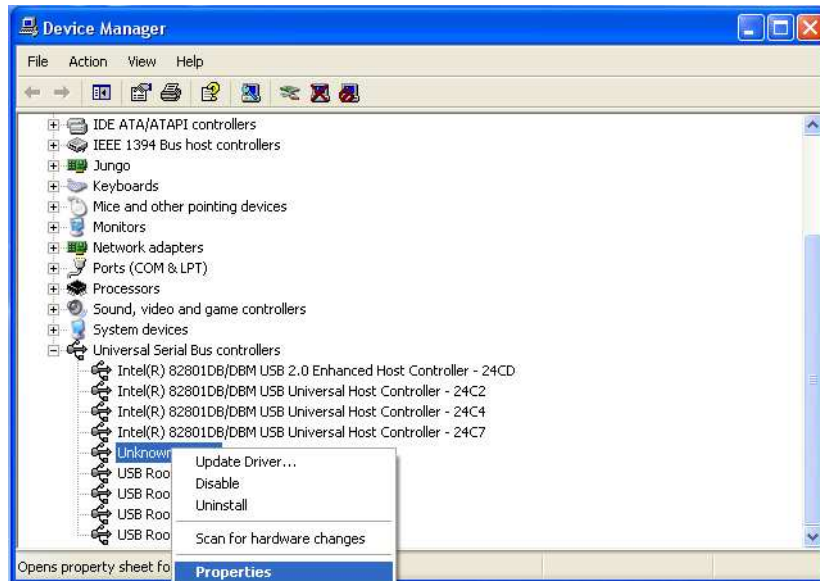
GRMON can connect to targets equipped with the USBDCCL core using the USB bus. To do so start GRMON with the **-usb** switch. Both USB 1.1 and 2.0 are supported. Several target systems can be connected to a single host at the same time. GRMON scans all the USB busses and claims the first free USBDCCL 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 has to be owned by the superuser (root) and have 's' (set user or group ID on execution) permission bit set (chmod +s grmon).

On Windows a driver has to be installed. The first the time the device is plugged in it should be automatically detected as seen in figure below.



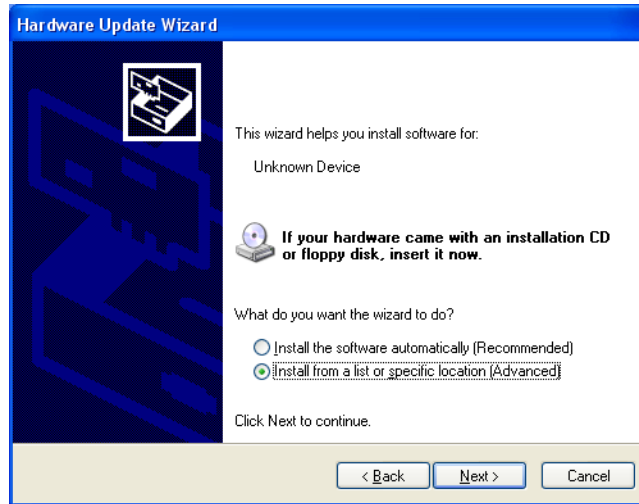
Then the new device can be found in the Device Manager. Press the right mouse button over the unknown device and choose update driver in the menu.



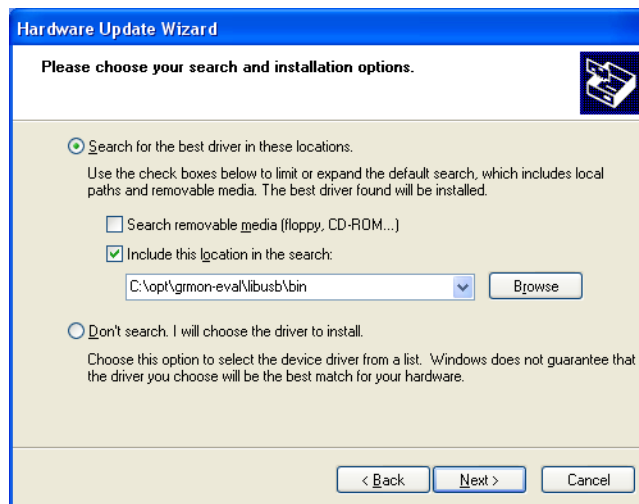
Select "No, not this time" in the next window and click next.



Select "Install from a list or specific location".



In the last step choose to search for the best driver and include the path "GRMON_ROOT\libusb\bin" where GRMON_ROOT is the root directory of the GRMON installation, usually 'C:\opt\grmon-eval' or 'C:\opt\grmon-pro'. Then click next and Windows should detect the driver. Click on "finish" when its done and then it should be possible to connect with GRMON using the -usb switch.



4.6 PCI debug interface

If target system has a PCI interface, GRMON can connect to the system using the PCI bus. Start GRMON with the `-pci vendor_id:device_id[:instance]` option and specify vendor id and device id in hexadecimal (with or without '0x' prefix):

```
.$ grmon -pci 16e3:1e0f
```

The default is to use the first instance of the board. If there are more than one board with the same vendor and device id the different boards can be selected with the instance number:

```
$ grmon -pci 16e3:1e0f:2
```

GRMON supports the Aeroflex Gaisler PCI cores inside GRLIB (pci_gr, pci_target, pci_mtf, pcidma) and the Insilicon PCI core (pci_is) on the AT697 (LEON2-FT) device.

On Linux GRMON needs root privilege to be able to access PCI memory and I/O ports. This can be accomplished by letting the GRMON binary be owned by root (chown root grmon) and setting the 's' (set user or group ID on execution) permission bit (chmod +s grmon).

On Windows a special PCI driver must be installed. It is available as an installer 'GRPCISetup.exe' in the folder 'pci'. It is a standard Windows installer which will install everything that is needed for PCI to work with GRMON.

4.7 GRESB debug interface

Targets equipped with a Spacewire core with RMAP support can be debugged through the GRESB debug interface using the GRESB Ethernet to Spacewire bridge. To do so start GRMON with the **-gresb** switch and use the following switches to set the needed parameters:

- ip** <ipnum> Connect to the bridge using the IP address *ipnum* . Default is 192.168.0.51.
- link** <linknum> 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.
- 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** <n> Timeout period in seconds for RMAP replies. Defaults is 8.
- gresbretry** <n> Number of retries for each timeout. Defaults to 0.

- grusb** Use USB rather than ethernet. The **-gresb** option must be omitted

Examples: `grmon -gresb -ip 192.168.0.55 -dna 0xfe -sna 32 -link 0 -clkdiv 1 -u`
`grmon -grusb -dna 0xfe -sna 32 -link 0 -clkdiv 1 -u`

NOTE: When using the **-grusb** switch the GRESB can not be used for normal operation. Only debugging through GRMON.

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

4.8 WildCard debug interface

The WildCard™ is a development board from Annapolis Micro Systems, Inc. It contains a 32-bit CardBus interface, programmable clock generator, a Processing Element (Virtex™ XCV300E -6 FPGA), two SSRAM memory ports and two I/O ports.

The necessary Windows drivers and API are delivered with the card. The *wcapi.dll* file should be in the search path. For more information, visit <http://www.annapmicro.com>

For GRMON to communicate with the WildCard, the GRLIB WILD2AHB - WildCard Debug Interface IP core must be present in the Processing Element FPGA design. The WildCard debug interface supports only 32-bit AMBA accesses. GRMON and the WILD2AHB IP core have been tested on Windows XP with the following WildCard software and hardware versions:

- API version 1.8
- Driver version 1.8
- Firmware version 2.4
- Hardware version 4.0, Rev. D:
 - XCV300E-BG352-6
 - 2 x 256kByte SSRAM, 10 ns

Below is a list of start-up switches applicable to the WildCard debug interface:

- | | |
|-------------------------------|---|
| -wildmhz <i>mhz</i> | Set the frequency of the F_CLK clock from the clock generator on the WildCard. The default is 20 MHz (see WildCard documentation for minimum and maximum values). |
| -wildfile <i>file</i> | Set the name of the WildCard programming file. The default is “ <i>wildcard-xcv300e.bin</i> ”, as generated by GRLIB. |
| -wilddev <i>device</i> | Set the WildCard device number. The default is 0. |
| -wildstat | Enable WildCard version and hardware information printout. |

5 Debug drivers

5.1 LEON2 and LEON3 debug support unit (DSU) drivers

The DSU debug drivers for LEON2 and LEON3 processors handle most of the functions regarding application debugging, processor register access and trace buffer handling. Since the DSU for LEON2 and LEON3 are not identical, two separate drivers are used.

5.1.1 Internal commands

The driver for the LEON2/3 debug support unit provides the following internal commands:

ahb [<i>length</i>]	Print the AHB trace buffer. The <i>length</i> AHB transfers will be printed, default is 10.
break < <i>addr</i> >	print or add breakpoint
bwatch < <i>addr</i> > [<i>delay</i>] [break]	Add a buswatch at address <i>addr</i> with an optional <i>delay</i> .
cont	continue execution
cpu [<i>enable</i> <i>disable</i> <i>active</i>] <i>cpuid</i>	Control processors in LEON3 multi-processor (MP) systems. Without parameters, the <i>cpu</i> command prints the processor status.
dcache [0 1]	show, enable or disable data cache
delete < <i>bpnum</i> >	delete breakpoint(s)
float	display FPU registers
gdb [<i>port</i>]	connect to GDB debugger
go [<i>addr</i>]	start execution without initialization
hbreak	print breakpoints or add hardware breakpoint (if available)
hist [<i>length</i>]	Print the trace buffer. The <i>length</i> last executed instructions or AHB transfers will be printed, default is 10.
icache [0 1]	show, enable or disable instruction cache
mmu	print the SRMMU registers (see also the <i>-srmmu</i> < <i>cpunum</i> > switch)
mmu [<i>mctrl</i> <i>ctxp</i> <i>ctx</i>] <i>val</i>	write value to mmu register
profile [0 1]	enable/disable/show simple profiling
register	show/set integer registers
run [<i>addr</i>]	reset and start execution at last entry point, or at <i>addr</i>
stack [<i>val</i>]	show/set the stack pointer
step [<i>n</i>]	single step one or [<i>n</i>] times
thread info	show thread information (RTEMS only)
tmode [<i>proc</i> <i>ahb</i> <i>both</i> <i>none</i>]	Select tracing mode between none, processor-only, AHB only or both.
walk [<i>address</i>]	do address translation for a virtual address
wash	Clear all SRAM and SDRAM memory
watch [<i>addr</i>]	print or add data watchpoint

5.1.2 Command line switches

The following command line switches are accepted:

- mp** Start-up in MP mode (all processors enabled)
- srmmu <cpunum>** Specifies that cpu <cpunum> has a SRMMU present. This will enable the SRMMU commands as well as virtual address translation if SRMMU is enabled.

5.2 Memory controller driver

The memory controller debug driver provides functions for memory probing and waitstate control. These switches are applicable to the LEON2 memory controller (MCTRL), and the FTMCTRL, SRCTRL and FTSRCTRL memory controller cores.

5.2.1 Internal commands

- mcfg1 [value]** Set the default value for memory configuration register 1. When the 'run' or 'load' command is given, MCFG1, 2&3 are initialized with their default values to provide the application with a clean startup environment. If no value is given, the current default value is printed.
- mcfg2 [value]** As mcfg1 above, but setting the default value of the MCFG2 register.
- mcfg3 [value]** As mcfg1 above, but setting the default value of the MCFG3 register.

5.2.2 Command line switches

The following start-up switches are recognized:

- rambanks *ram_banks*** Overrides the auto-probed number of populated ram banks.
- cas *delay*** Programs SDRAM to either 2 or 3 cycles CAS delay. Default is 2.
- mcfg1, -mcfg2, -mcfg3** Set the default value for memory configuration register 1, 2 or 3 respectively.
- noflash** Do not probe for flash memory at start-up
- nosram** Disable sram and map sdram from address 0x40000000
- ram *ram_size*** Overrides the auto-probed amount of static ram. Size is given in Kbytes.
- romrws *waitstates*** Set *waitstates* number of waitstates for rom reads.
- romwws *waitstates*** Set *waitstates* number of waitstates for rom writes.
- romws *waitstates*** Set *waitstates* number of waitstates for both rom reads and writes.
- ramrws *waitstates*** Set *waitstates* number of waitstates for ram reads.
- ramwws *waitstates*** Set *waitstates* number of waitstates for ram writes.
- ramws *waitstates*** Set *waitstates* number of waitstates for both ram reads and writes.
- trp3** Programs the SDRAM trp timing to 3 (sets bit 30 in mcfg2). Default is 2.
- trfc *val*** Programs the SDRAM trfc field in mcfg2 to represent *val* ns.

5.3 On-chip logic analyser driver (LOGAN)

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**.

5.3.1 Internal commands

- la status** Reports status of logan (equivalent with writing just la).
- la arm** Arms the logan. Begins the operation of the analyzer and sampling starts.
- la reset** Stop the operation of the logan. Logic Analyzer returns to idle state.
- la pm** [*trig level*] [*pattern*] [*mask*]
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 pat** [*trig level*] [*bit*] [*0 / 1*]
Sets/displays the specified bit in the pattern of the specified trig level to 0/1.
- la mask** [*trig level*] [*bit*] [*0 / 1*]
Sets/displays the specified bit in the mask of the specified trig level to 0/1.

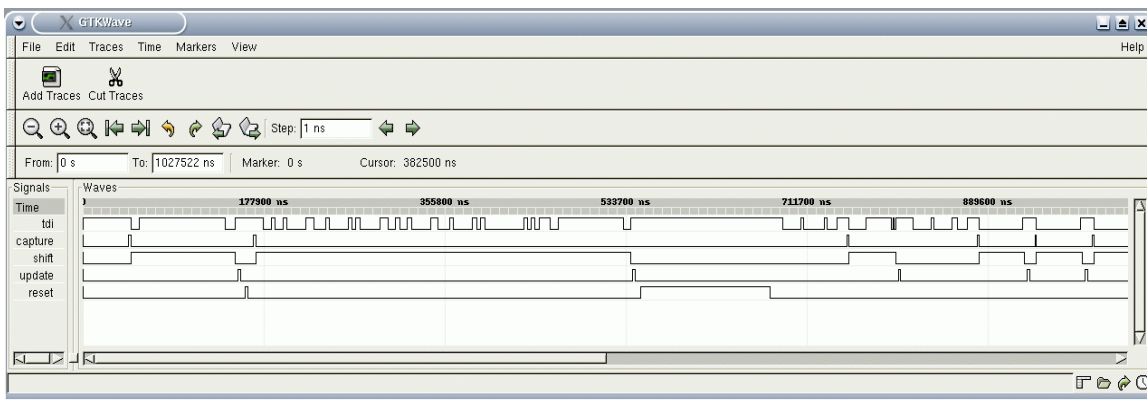
- la trigctrl** [*trig level*] [*match counter*] [*trig condition*]
Sets/displays the match counter and the trigger condition (1 = trig on equal, 0 = trig on not equal) for the specified trig level.
- la count** [*value*]Set/displays the trigger counter. The value should be between zero and depth-1 and specifies how many samples that should be taken after the triggering event.
- la div** [*value*] 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 qual** [*bit*] [*value*]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 dump** [*filename*]This dumps the trace buffer in VCD format to the file specified (default is log.vcd).
- la view** [*start index*] [*stop index*] [*filename*]
Prints the specified range of the trace buffer in list format. If no filename is specified the commands prints to the screen.
- la page** [*page*] 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.

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.

Example:

```
count31_16 16  
count15_6 10  
count5_0 6
```

This configuration has a total of 32 traced signals and they will be displayed as three different signals being 16, 10 and 6 bits wide. The first signal in the configuration file maps to the most significant bits of the vector with the traced signals. The created VCD file can be opened by waveform viewers such as GTKWave or Dinotrace. Below is an example trace displayed in GTKWave.



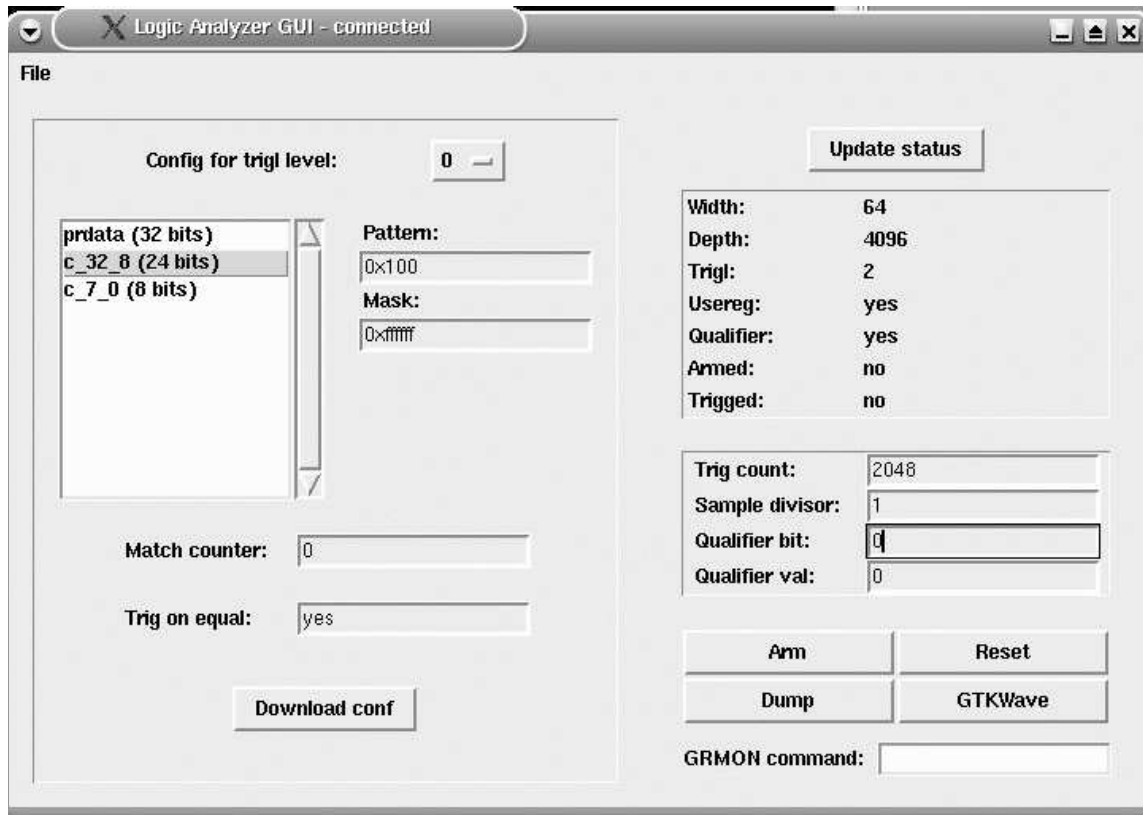
To simplify the operation of the logic analyzer, a graphical GUI is available. The GUI is written in Tcl/Tk and can be connected to GRMON using the GDB interface. To use the GUI, enter the `gdb` command in GRMON and then launch the `tcl` file `logan.tcl`.

The left side of the GUI window has the settings for the different trig levels, i.e. pattern, mask, match counter and trigger condition. Which trig level the settings apply to is chosen from an option menu. The “Download conf” button transfers the values to the on-chip logic analyzer. The pattern and mask is padded with zeroes from the left if not fully specified. They can be entered either in hexadecimal or decimal but there is a limitation that no signal can be wider than 64 bits.

The right side of the window shows the status the settings which control the trace buffer. These settings are sent to the logic analyzer when the user presses enter. The ‘armed’ and ‘triggered’ fields of the status can be re-read by pressing the “Update status” button.

There are also buttons to arm and reset the logic analyzer as well as to dump the vcd-file and launch GTKWave. Any GRMON command can be issued from the entry below these buttons.

From the file menu the current configuration can be saved and a new one can be loaded. The GUI defaults to the same configuration file as the GRMON debug driver. If the configuration is saved it adds information about the setup which is ignored by GRMON. When saving/loading any filename may be specified but during startup the GUI reads the “setup.logan” file. Only files previously saved by the GUI can be loaded from this menu option.



5.4 AMBA wrapper for Xilinx System Monitor

The debug driver driver for the AMBA wrapper for Xilinx System Monitor allows reading of device temperature and voltages. The debug driver does not perform any calibration of the System Monitor.

5.4.1 Internal commands

- | | |
|-----------------------------|---|
| grsysmon calibrate | Initialize and calibrate out any offset and gain errors |
| grsysmon temperature | Displays temperature readings |
| grsysmon voltage | Displays voltage readings |

5.5 ATA/IDE controller

The ATA/IDE controller is used when the target system contains the ATACTRL core. The driver implements one command, **ata**, which displays the types of disk drives that are attached the IDE controller. Attached disks are not automatically probed when GRMON is started, but probing can be forced by starting GRMON with **-ata**. The command **'info sys'** will then display the attached disks.

5.6 DDR memory controller (DDRSPA)

The driver for the DDRSPA memory controller performs the DDR initialization sequence, and probes the amount of available memory. To skip the probing and use the pre-configured values, add **-nosdinit** at start up. The 'info sys' command will show the DDR timing and the amount of detected memory:

```
04.01:025 Gaisler Research DDR266 Controller (ver 0)
          ahb: 40000000 - 50000000
          ahb: fff00100 - fff00200
          16-bit DDR : 1 * 64 Mbyte @ 0x40000000
                   100 MHz, col 10, ref 7.8 us
```

5.6.1 Command line switches

The following start-up switches are recognized:

- trp3** Programs the SDRAM trp timing to 3 (sets bit 30 in ddrcfg1). Default is 2.
- trcd val** Programs the SDRAM trcd timing to *val* - 2 (sets bit 26 in ddrcfg1). Default is 2.
- trfc val** Programs the SDRAM trfc field in ddrcfg1 to represent *val* ns.
- nosdinit** To skip memory probing and use pre-configured values.

5.7 DDR2 memory controller (DDR2SPA)

The driver for the DDR2SPA memory controller performs the DDR2 initialization sequence, and probes the amount of available memory. To skip the probing and use the pre-configured values, add **-nosdinit** at start up. The 'info sys' command will show the DDR2 timing and the amount of detected memory:

```
00.01:02e Gaisler Research DDR2 Controller (ver 0x0)
          ahb: 40000000 - 60000000
          ahb: fff00100 - fff00200
          64-bit DDR2 : 1 * 256 Mbyte @ 0x40000000
                   200 MHz, col 10, ref 7.8 us
```

5.7.1 Internal commands

- ddr2cfg1** [*value*] Set the default value for DDR2 control register 1.
- ddr2cfg2** Print the value of DDR2 control register 2.
- ddr2cfg3** [*value*] Set the default value for DDR2 control register 3.
- ddr2cfg4** [*value*] Set the default value for DDR2 control register 4, if available.
- ddr2delay** [*inc* | *dec* | *reset* | *value*] Change read data input delay. Use **inc** to increment the delay with one tap-delay for all data bytes. Use **dec** to decrement all delays. **Reset** will set the delay to default value. 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.
- ddr2skew** [*inc* <*steps*> | *dec* <*steps*> | *reset*] Change read skew. Use **inc** to increment the delay with one step. Use **dec** to decrement the delay with one step. **reset** will set the skew to the default value. **inc** and **dec** can optionally be given the number of steps to increment/decrement as an argument.

5.7.2 Command line switches

The following start-up switches are recognized:

-trp3	Programs the SDRAM trp timing to 3 (sets bit 28 in ddrcfg3). Default is 2.
-trcd <i>val</i>	Programs the SDRAM trcd timing to <i>val</i> - 2 (sets bit 26 in ddrcfg1). Default is 2.
-trfc <i>val</i>	Programs the SDRAM trfc field in ddrcfg3 to represent <i>val</i> ns.
-twr <i>val</i>	Programs the SDRAM twr field in ddrcfg3 to represent <i>val</i> ns.
-nosdinit	To skip memory probing and use pre-configured values.

5.8 I²C-master (I2CMST)

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

5.8.1 Internal commands

i2c bitrate <i>[normal / fast]</i>	Initializes the prescale register for Normal or Fast operation
i2c enable	Enables core
i2c disable	Disables core
i2c dvi	Interaction with DVI transmitters. Issue i2c dvi help for further info.
i2c read <i>[i2cadd]</i>	Performs a simple read from slave with I ² C-address <i>i2cadd</i>
i2c read <i>[i2cadd] [addr]</i>	Reads memory location <i>addr</i> from slave with I ² C-address <i>i2cadd</i>
i2c read <i>[i2cadd] [addr] [cnt]</i>	Performs <i>cnt</i> sequential reads starting at <i>addr</i> from slave with <i>i2cadd</i>
i2c scan	Scans the bus for devices.
i2c status	Displays some status information about the core and the bus.
i2c write <i>[i2cadd] [data]</i>	Writes <i>data</i> to slave with I ² C-address <i>i2cadd</i>
i2c write <i>[i2cadd] [addr] [data]</i>	Writes <i>data</i> to memory location <i>addr</i> on slave with address <i>i2cadd</i>

5.9 GRPCI master/target (PCI_MTF)

The debug driver for the GRPCI is mainly useful for PCI host systems. The 'grpci init' command initialises the host BAR1 to point to RAM (PCI address 0x40000000 -> AHB address 0x4000000) and enables PCI memory space and bus masterering. 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.

5.9.1 Internal commands

grpci init	Initialises the GRPCI core as described above
grpci info	Displays information about the GRPCI core
grpci scan	Scans all PCI slots for available devices
grpci conf	Configures the BARs of the found devices

grpci disable bt Disables the GRPCI byte twisting

The **grpci conf** command can fail to configure all found devices if the PCI address space addressable by the GRPCI core is smaller than the amount of memory needed by the devices.

5.10 PCIF master/target (PCIF)

The debug driver for the PCIF core is mainly useful for PCI host systems. The 'apcif init' command initialises the host BAR1 to point to RAM (PCI address 0x40000000 -> AHB address 0x40000000) and enables PCI memory space and bus masterering. Commands are provided for scanning the bus, configuring the found resources, and displaying information. Note that on non-host systems only the **info** command has any effect.

5.10.1 Internal commands

apcif init	Initialises the PCIF core as described above
apcif info	Displays information about the PCIF core
apcif scan	Scans all PCI slots for available devices
apcif conf	Configures the BARs of the found devices

The **apcif conf** command can fail to configure all found devices if the PCI address space addressable by the PCIF core is smaller than the amount of memory needed by the devices.

5.11 On-chip PCI trace buffer driver (PCITRACE)

The PCITRACE debug driver contains commands to control the PCI trace buffer core. It allows to set various triggering conditions, and to generate VCD waveform files from trace buffer data. All PCI trace buffer commands are prefixed with **pci** :

pci	Reports current trace buffer settings and status
pci arm	Arms the trace buffer and starts sampling.
pci log	Prints the trace buffer data and saves the VCD data to 'log.vcd'
pci address[<i>val</i>]	Get/set the address pattern register
pci amask[<i>val</i>]	Get/set the address mask register
pci sig[<i>val</i>]	Get/set the signal pattern register
pci smask[<i>val</i>]	Get/set the signal mask register
pci tcount[<i>val</i>]	Get/set the trigger count register
pci tpoint[<i>val</i>]	Get/set the trigger point register

The pci log command save the current trace buffer data in VCD format to the file log.vcd. The data can then be displayed in a VCD viewer such as GTKWave.

5.12 SPI Controller (SPICTRL)

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:

```
0a.01:029  Gaisler Research  SPI Controller (ver 0x0)
           irq 10
           apb: 80000a00 - 80000b00
           FIFO depth: 2, 3 slave select signals
```

The following commands are supported by the driver:

spi am	Commands for automated transfers, issue spi help am for more information.
spi aslvsel <value>	Set automatic slave select register
spi disable	Disable core
spi enable	Enable core
spi rx	Read Receive register
spi selftest	Place the core in loop mode and perform a simple self test.
spi set <field>	Set specified field in the Mode register. See comments below.
spi slvsel <value>	Set slave select register
spi status	Display core status information
spi tx <data>	Write data to Transmit register. GRMON aligns the written data.
spi unset <field>	Unset specified field in the Mode register. Se comments below.

The **spi set** command takes one or more field names as arguments and modifies the Mode register accordingly. Valid arguments to the set command are; **cg <value>**, **cpol**, **cpha**, **div16**, **len <value>**, **amen**, **loop**, **ms**, **pm <value>**, **tw**, **asel**, **fact**, **od**, **tac** and **rev**. To configure the core for master operation, with a clock phase of 1, 8 bit word length and MSb first transmission the arguments to set are: **spi set ms cpha len 7 rev**

The **spi unset** command sets the specified field to zero, valid arguments to the unset command are; **cpol**, **cpha**, **div16**, **amen**, **loop**, **ms**, **tw**, **asel**, **fact**, **od**, **tac** and **rev**. Note that the unset command can not be used to modify the CG, LEN or PM fields.

The debug driver has bindings to the SPI memory device layer. These commands are accessed via **spi flash**. Please see appendix 6.2 for more information.

5.13 SPI Memory Controller (SPIMCTRL)

The SPIMCTRL debug driver provides commands 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 following commands are supported by the driver:

spim altscaler	Toggle use of alternate scaler
spim reset	Core reset
spim status	Displays core status information
spim tx <data>	Shift a byte to the memory device
spim sd csd	Displays and decodes CSD register of SD Card
spim sd reinit	Re-initializes SD Card

The debug driver has bindings to the SPI memory device layer. These commands are accessed via **spim flash**. Please see appendix 6.2 for more information.

5.14 SVGA Frame buffer (SVGACTRL)

The SVGACTRL debug driver implements functions to report the available video clocks in the SVGA frame buffer, and to display screen patterns for testing. The **info sys** command will display the available video clocks:

```
03.01:063  Gaisler Research  SVGA frame buffer (ver 0)
          ahb master 3
          apb: 80000600 - 80000700
          clk0: 25.00 MHz  clk1: 50.00 MHz  clk2: 65.00 MHz
```

The driver implements the following commands:

svga custom	Show/set custom format, see description below.
svga draw test_screen <format> <bitdepth>	Draw test screen
svga draw <file> <bitdepth>	Display file in ASCII (PPM) format
svga frame <address>	Show or set start address of framebuffer memory
svga formats	Show available display formats
svga formatsdetailed	Show detailed view of available display formats
svga help	Show available commands

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.

5.15 USB Host Controller (GRUSBHC)

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:

```
02.01:026  Gaisler Research  USB EHCI controller (ver 0x0)
          ahb master 2, irq 9
          apb: 80000e00 - 80000f00
          1 ports, byte swapped registers
```

GRMON enumerates the universal host controllers present in the system and **info sys** will, in addition to number of ports and register setting, display the number assigned to the universal host controller:

```
03.01:027  Gaisler Research  USB UHCI controller (ver 0x0)
          ahb master 3, irq 10
          ahb: fffa0000 - fffa0100
          cc#: 1, 1 ports, byte swapped registers
```

5.15.1 Internal commands

The enhanced host controller's debug driver provides the following internal commands:

ehci endian	Displays the endian conversion setting
ehci capregs	Displays contents of the capability registers
ehci opregs	Displays contents of the operational registers
ehci reset	Performs a Host Controller Reset

The internal commands provided by the universal host controller's debug driver are:

uhci endian [<i>cc#</i>]	Displays the endian conversion setting
uhci ioregs [<i>cc#</i>]	Displays contents of the I/O registers
uhci reset [<i>cc#</i>]	Performs a Global reset

The *cc#* argument is only processed if there is more than one universal host controller present in the system.

5.15.2 Command line switches

Both the debug driver for the enhanced host controller and the universal host controller support the following command line switch:

-nousbrst	Do not perform a host controller reset when GRMON is started
------------------	--

5.16 AMBA AHB trace buffer driver (AHBTRACE)

The AHBTRACE debug driver contains commands 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.

5.16.1 Internal Commands

at [<i>entries</i>]	Prints [<i>entries</i>] of the current trace buffer log
at log	Prints all entries of the current trace buffer log
at reg	Reports current trace buffer settings and status
at enable	Arms the trace buffer and starts recording
at disable	Stops the trace buffer recording
at delay [<i>val</i>]	Delay the stops the trace buffer recording after match
at bp1 address mask	Sets break point 1, address and mask
at bp2 address mask	Sets break point 2, address and mask

5.17 10/100 Mbit/s Ethernet Controller (GRETH)

The GRETH debug driver provides commands to configure the GRETH 10/100 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:

```
0a.01:029  Gaisler Research  SPI Controller (ver 0x0)
           irq 10
           apb: 80000a00 - 80000b00
           FIFO depth: 2, 3 slave select signals
```

5.17.1 Internal commands

edcl [ip]	Show or set the EDCL IP number.
mdio <paddr <raddr>	Show PHY register. paddr = PHY address, raddr = register address
wmdio <paddr> <raddr>	Write PHY address. paddr = PHY address, raddr = register address
phyaddr <addr>	Set the default PHY address to <addr>

When setting the EDCL IP number using the **edcl** command, the IP number should have the numeric format:

```
edcl 192.168.0.66
```

5.17.2 Command line switches

The following command line switches are accepted:

-10m	Start-up in 10 Mbit/s mode
-phyaddr <addr>	Program the GRETH MDIO register to use the PHY with address <addr>.

5.18 USB 2.0 Device Controller (GRUSBDC)

The **info sys** command displays the core's number of IN and OUT endpoints. If the core is configured with separate interrupts then the three different interrupt numbers will be displayed as well.

```
04.01:021  Gaisler Research  GR USB 2.0 Device Controller (ver 0x0)
           ahb master 4, irq 9
           ahb: fff00400 - fff00800
           2 IN endpts, 2 OUT endpts
```

5.18.1 Internal commands

The debug driver provides the following internal command:

usbdc info	Prints information about AHB interface, number of IN and OUT endpoints, interrupt configuration, and endpoint buffer sizes.
-------------------	---

5.19 L2-Cache Controller (L2C)

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

```
01.01:04b  Gaisler Research  L2-Cache Controller (ver 0x0)
           ahb master 1
```

```
ahb: 40000000 - 80000000
ahb: ff400000 - ff800000
ahb: f0000000 - f0100000
L2C: 2-ways, setsize = 256kbytes
```

5.19.1 Internal commands

The debug driver provides the following internal command:

l2cache lookupaddr	Prints the data and status of a cache line if addr generates a cache hit.
l2cache showdata [way] [count] [start]	Prints the data of count cache line starting at cache line start .
l2cache showtag [count] [start]	Prints the tag of count cache line starting at cache line start .
l2cache [enable disable]	To enable or disable the cache.
l2cache flushall [mode]	Perform a cache flush to all cache lines using a flush mode .
l2cache flushmem [addr] [mode]	Perform a cache flush to the cache lines with a cache hit for addr using a flush mode .
l2cache flushdirect [addr] [mode]	Perform a cache flush to the cache lines addressed with addr using a flush mode .
l2cache hit	Prints the hit rate statistics.
l2cache status	Prints the status and configuration register.
l2cache checktag	Perform a check for illegal tags.
l2cache checkdata [dirty]	Perform a check for unmatched data in cache and memory. With the dirty switch, all dirty cache lines is printed.

6 FLASH programming

6.1 CFI compatible Flash PROM

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.

The following flash programming commands are provided:

flash	Print the on-board flash memory configuration
flash blank [addr all]	Check that the flash memory is blank, i.e. can be re-programmed. The complete flash memory (all) can be checked, or an address range can be given (e.g. 'flash erase 0x1000 0x8000').
flash erase [addr all]	Erase a flash block at address addr , or the complete flash memory (all). An address range is also support,e.g. 'flash erase 0x1000 0x8000'.
flash load <file>	Program the flash memory with the contents <i>file</i> . Recognized file formats are ELF and srecord.
flash lock [addr all]	Lock a flash block at address addr , or the complete flash memory (all). An address range is also support, e.g. 'flash lock 0x1000 0x8000'.
flash lockdown [addr all]	Lock-down a flash block at address addr , or the complete flash memory (all). An address range is also support, e.g. 'flash lockdown 0x1000 0x8000'.
flash query	Print the flash query registers
flash status	Print the flash lock status register
flash unlock [addr all]	Unlock a flash block at address addr , or the complete flash memory (all). An address range is also support, e.g. 'flash unlock 0x1000 0x8000'.
flash write <addr> <data>	Write a 32-bit data word to the flash at address addr .

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

```
flash unlock all
flash erase all
flash load file.exe
flash lock all
```

6.2 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. To issue a SPI flash command the commands listed below must be prefixed with the core debug driver name. If the memory device is attached to a SPICTRL core, all flash commands start with **spi flash**. If the memory device is attached to a SPIMCTRL core, all flash commands start with **spim flash**, see below for examples.

flash	Prints a list of available commands.
flash detect	Tries to auto detect type of SPI memory device
flash dump <addr> <len> <file>	Dumps len bytes starting at addr to a file. addr is the address presented to the device and does not correspond to a possible AMBA mapping of the memory device.
flash erase	Erases all of memory device. All devices do not require/support an erase operation.
flash fast	Enables or disables FAST READ command (memory device may not support this).
flash load <file>	Load the contents of file into the memory device. Any address ranges in the file are fed used to directly address the memory device.
flash select [index]	Selects memory device type. If index is not specified, a list of the supported devices is displayed.
flash set [...]	Specifies custom set of parameters for memory device. Issue flash set to see a list of the required parameters.
flash show	Displays current memory device configuration
flash status	Show memory device status
flash ssva [value]	Show/set slave select value to be used when communicating via SPICTRL.
flash strict [on off]	Show/set strict communication mode. Set to "on" if programming fails.
flash verify <file>	Verifies that data in file file matches data in memory device.
flash wrdi	Issue write disable instruction
flash wren	Issue write enable instruction
flash help <command>	Displays command list or additional information about a specific command.

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 dump the data of a SPI memory device connected via SPICTRL could be:

```
spi set div16
spi flash select 1
spi flash dump 0 32 32bytes.srec
```

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*. A command sequence to dump the data of a SPI memory device connected via SPIMCTRL could be:

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

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*.

7 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 a one of the memory blocks, effectively emulating an SEU. The user defines error rate and chooses between two error distribution modes. GRMON can also perform error correction monitoring and report error injection statistics including number of detected and injected errors and error coverage.

Error injection is available when GRMON is connected using JTAG, PCI or Ethernet communication interface and can not be used together with “-u” switch (UART1 in loop-back mode). Error injection function is only available in the professional version of GRMON.

Following error injection commands are available:

- 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 dis** Disable error injection.
- ei log *log_file*** Enable error injection. The error injection log is saved in file *log_file*.
- ei stat** Show error injection statistics.
- ei stat en** Enable 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 stat dis** Disable error injection statistics.

```
jupiter:/tmp>grmon -i -gplib -jtag

GRMON LEON debug monitor v1.1.3

Copyright (C) 2004,2005 Gaisler Research - all rights reserved.
For latest updates, go to http://www.gaisler.com/
Comments or bug-reports to support@gaisler.com

using JTAG cable on parallel port
JTAG chain: ETH-PHY xc2v3000 xc2v3000 xc18v04 xc18v04 xc18v04

initialising .....
detected frequency: 41 MHz

Component                               Vendor
LEON3 SPARC V8 Processor                 Gaisler Research
AHB Debug UART                          Gaisler Research
AHB Debug JTAG TAP                       Gaisler Research
LEON2 Memory Controller                 European Space Agency
AHB/APB Bridge                          Gaisler Research
LEON3 Debug Support Unit                Gaisler Research
Generic APB UART                        Gaisler Research
Multi-processor Interrupt Ctrl          Gaisler Research
```

Modular Timer Unit

Gaisler Research

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

```
grmon> lo rtems-tasks
section: .text at 0x40000000, size 116640 bytes
section: .data at 0x4001c7a0, size 2720 bytes
total size: 119360 bytes (310.6 kbit/s)
read 520 symbols
entry point: 0x40000000
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
```

Program exited normally.

```
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>
```

8 Extending GRMON

8.1 Loadable command module

GRMON can be extended with custom commands by creating a loadable command module. This module will have read and write access to the AHB bus and registers. The module should export a pointer of type *UserCmd_T*, called *UserCommands*, e.g.:

```
UserCmd_T *UserCommands = &CommandExtension;
```

UserCmd_T is defined as:

```
typedef struct
{
    void *lib;

    /* Functions exported by grmon */
    int (*MemoryRead)(void *lib, unsigned int addr,
                     unsigned char *data, unsigned int length);
    int (*MemoryWrite)(void *lib, unsigned int addr,
                      unsigned char *data, unsigned int length);
    void (*GetRegisters)(void *lib, unsigned int registers[]);
    void (*SetRegisters)(void *lib, unsigned int registers[]);
    void (*dprint)(char *string);

    /* Functions provided by user */
    int (*Init)();
    int (*Exit)();
    int (*CommandParser)(int argc, char *argv[]);
    char **Commands;
    int NumCommands;
} UserCmd_T;
```

The first five entries are function pointers that are provided by GRMON when loading the module

- *MemoryRead* implements an AHB read on the target system. The AHB address is passed in *addr*, while the read data is returned in the **data* pointer. The *length* parameter defines the number of bytes to read.
- *MemoryWrite* implements an AHB write on the target system. The AHB address is passed in *addr*, while **data* should point to the data to be written. The *length* parameter defines the number of bytes to be written. The write length should be a multiple of 4.
- *GetRegisters* gets the processor registers. See the `grmon.h` include file for register definitions.
- *SetRegisters* sets the processor registers. See the `grmon.h` include file for register definitions.
- *dprint* prints a string to the GRMON console

The void pointer argument should be supplied with the pointer *lib*

```
CommandExtension.MemoryWrite(CommandExtension.lib, ...);
```

The five last entries are to be implemented by the user:

- *Init* and *Exit* are called when entering and leaving a GRMON target.
- *CommandParser* are called from GRMON before any internal parsing is done. This means that you can override internal GRMON commands. On success *CommandParser* should return 0 and on error the return value should be > 200. On error, GRMON will print out the error number for diagnostics. *argv[0]* is the command itself and *argc* is the number of tokens, including the command, that is supplied.
- *Commands* should be a list of available commands. (used for command completion)

- NumCommands should be the number of entries in Commands. It is crucial that this number matches the number of entries in Commands. If NumCommands is set to zero, no command completion will be done.

Compile the command module to a library as follows

```
gcc -fPIC -c commands.c
gcc -shared commands.o -o commands.so
```

Load the command module with the -ucmd option

```
grmon -ucmd commands.so
```

A simple example of a command module is supplied with the professional version of GRMON.

8.2 Custom DSU communications module

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. The custom interface exports a structure with the following definition:

```
struct ioif {
    int (*wmem) (unsigned int addr, 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);
};
```

The loadable module should export a variable called *DsuUserBackend* of the type *ioif*:

```
struct ioif my_io = {my_wmem, my_gmem, my_open, my_close, my_baud, my_init};

struct ioif *DsuUserBackend = &my_io;
```

When GRMON is started, the start-up switch '-dback my_io.so' should be used to load the module with the custom interface. The *DsuUserBackend* structure will then replace the built-in DSU communications interfaces. The members in the *ioif* structure are defined as follows:

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

Pointer to 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. 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);
```

Pointer to 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. 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.

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

Called on GRMON exit.

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

Called when the GRMON 'baud' command is given. The baud parameter indicates the new baud rate. Can be ignored if not applicable.

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

Called on GRMON start-up. 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.

The loadable module should be compiled into a library as follows:

```
gcc -fPIC -c my_io.c  
gcc -shared my_io.o -o my_io.so
```

GRMON can be started with:

```
grmon -dback my_io.so -dbackarg XXX
```

An example module (my_io.c) is provided with the professional version of GRMON.

APPENDIX A: GRMON Command description

A.1 GRMON built-in commands

Command	Description
batch [-echo] <batchfile>	Execute a batch file of GRMON commands from <batchfile>. Echo commands if -echo is specified.
baud <rate>	Change DSU baud rate.
disassemble [addr [cnt]]	Disassemble [cnt] instructions at [addr].
dump <start> <end> [filename]	Dump target memory in address range [start,end] to srecord file.
echo	Echo string in monitor window.
exit	Alias for 'quit', exits monitor.
help [cmd]	Show available commands or usage for specific command.
info [drv libs reg sys]	Show debug drivers, libraries, system register or system configuration
load <file>	Load a file into memory. The file should be in ELF32, srecords or a.out format.
mem [addr] [count]	Examine memory at [addr] for [count] bytes.
shell <command>	Execute a shell command.
symbols [symbol_file]	Show symbols or load symbols from file.
quit	Exit GRMON and return to invoker (the shell).
verify <file>	Verify memory contents against file.
version	Show version.
wmem <addr> <data>	Write <data> to memory at address <addr>.

Table 1: GRMON built-in commands

A.2 LEON2/3 DSU commands

Command	Description
ahb [trace_length]	Show AHB trace.
break [addr]	Print breakpoints or add breakpoint if addr is supplied. Text symbols can be used instead of an address.
bwatch <addr> [delay] [break]	Add a buswatch at address or symbol addr, with optional delay. Break will stop execution when buswatch is hit.
bt	Print backtrace.
cont	Continue execution.
cp	Show registers in co-processor (if present).
dcache	Show data cache.
delete <bp>	Delete breakpoint 'bp'.
float	Display FPU registers.
gdb [port]	Connect to the GNU debugger (GDB).
go	Start execution at <addr> without initialisation.
hbreak [addr]	Print breakpoints or add hardware breakpoint.
hist [trace_length]	Show trace history.
icache	Show instruction cache
inst [trace_length]	Show traced instructions.
leon	Show LEON registers.
mmu	Print mmu registers.
profile [0 1]	enable/disable simple profiling. No arguments shows current profiling statistics.
register [reg win] [val]	Show/set integer registers (or windows, e.g. 're w2')
run	Run loaded application.
stack <addr>	Set stack pointer for next run.
step [n]	Single step one or [n] times.
tm [ahb proc both none]	Select trace mode.
va <addr>	Performs a virtual-to-physical translation of address.
version	Show version.
watch [addr]	Print or add watchpoint.

Table 2: LEON2/3 DSU commands

A.3 FLASH programming commands

Command	Description
flash	Print the detected flash memory configuration.
flash disable	Disable writes to flash memory.
flash enable	Enable writes to flash memory.
flash erase [addr] all	Erase flash memory blocks.
flash load <file>	Program file into flash memory
flash lock [addr] all	Lock flash memory blocks.
flash lockdown [addr] all	Lockdown flash memory blocks.
flash query	Print the flash memory query register contents.
flash status	Print the flash memory block lock status.
flash unlock [addr] all	Unlock flash memory blocks.
flash write [addr] [data]	Write single data value to flash address.

Table 3: FLASH programming commands

APPENDIX B: License key installation

B.1 Installing HASP Device Driver

GRMON is licensed using a HASP USB hardware key. Before use, a device driver for the key must be installed. The latest drivers can be found at www.aladdin.com or www.gaisler.com. The installation is described below.

B.1.1 On a Windows NT/2000/XP host

The HASP device driver is installed using the installer HASPUserSetup.exe located in `hasp/windows/driver` directory on the GRMON CD. It will automatically install the required files.

Note: Administrator privileges are required to install the HASP device driver under Windows NT/2000/XP.

B.1.2 On a Linux host

The linux HASP driver consists of `aksusbd` daemon. It is contained in the `hasp/linux/driver` on the GRMON CD. The driver comes in form of RPM packages for Redhat and Suse linux distributions. The packages should be installed as follows:

Suse systems:

```
rpm -i aksusbd-suse-1.8.1-2.i386.rpm
```

Redhat systems:

```
rpm -i aksusbd-redhat-1.8.1-2.i386.rpm
```

The driver daemon can then be started by re-booting the most, or executing:

```
/etc/rc.d/init.d/aksusbd start
```

Note: All described action should be executed as root.

On other linux distributions, the driver daemon will have to be started manually. This can be done using the `HDD_Linux_dinst.tar.gz`, which also contains instruction on how to install and start the daemon.

B.2 Node-locked license file

TSIM can also be licensed using a license file (node-locked license). If a node-locked license is used, the license file should be `~/license.dat`, or `./license.dat`, or it should be pointed to by the environment variable `GRMON_LICENSE_FILE`.

APPENDIX C: Fixed Configuration file format

A fixed configuration file can be used to describe the target system instead of probing the plug&play information. The configuration file describes which IP cores are present on the target, and on which addresses they are mapped. The file contains one 11-column row per target IP core. The columns in each row define the following parameters: vendor ID, device ID, start address of AHB area 1, end address of AHB area1, start address of AHB area 2, end address of AHB area 2, start address of AHB area 3, end address of AHB area 3, start address of APB area, end address of APB area, assigned interrupt.

The vendor and device ID for the various LEON2 and LEON3 IP cores can be found in grcommon.h which is supplied with the GRMON package.

Below is an example configuration file for a typical LEON2 system. To use a fixed configuration file, GRMON should be started with `-cfg <file>`.

```
# GRMON LEON2 config file

#vendor dev  ahbstart  ahbend                apbstart  apbend  irq
  4   f   0          20000000 20000000 40000000 40000000 80000000 80000000 80000010 0
  4   2   0           0         0         0         0         0         80000014 80000014 0
  4   8   0           0         0         0         0         0         80000024 80000028 0
  4   6   0           0         0         0         0         0         80000040 80000070 0
  4   7   0           0         0         0         0         0         80000070 80000080 3
  4   7   0           0         0         0         0         0         80000080 80000090 2
  4   5   0           0         0         0         0         0         80000090 800000A0 0
  4   9   0           0         0         0         0         0         800000A0 800000AC 0
  1   7   0           0         0         0         0         0         800000c0 800000d0 0
  1   2   90000000 A0000000 0         0         0         0         0         0         0
```

APPENDIX D: JTAG Configuration File

Unknown JTAG devices causing GRMON JTAG chain initialization to fail can be defined in JTAG Configuration File. GRMON is started with `-jtagcfg <JTAG configuration file>` switch. JTAG Configuration file can not be used with Altera download cable (`-altjtag`). An example of JTAG configuration file is shown below:

```
# JTAG Configuration file
# Device name      Device id      Device id mask  Instr. reg. length  Dbg Interface 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 interface and user instruction 1 and 2 fields, where:

- Device name String with device name
- Device id Device identification code
- Device id 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.
- Instruction register length Length of the instruction register in bits
- Debug interface Set to 1 if the device implements JTAG Debug Link, otherwise 0.
- User Instr 1 Code of the instruction used to access JTAG Debug Link Address/Command register (default is 0x2). Used only if Debug interface is set to 1.
- User Instr 2 Code of the instruction used to access JTAG Debug Link Data register (default is 0x3). Used only if Debug interface is set to 1.

APPENDIX E: USB-Blaster Driver setup for Linux

The Altera Quartus® II software uses the built-in USB drivers (usbfs) on Linux to access the USB-Blaster download cable. By default, root is the only user allowed to use usbfs. You must change the permissions on the ports before you can use the USB-Blaster download cable to access the JTAG DSU port via GRMON.

You must have system administration (root) privileges to configure the USB-Blaster download cable drivers.

E.1 Driver Setup on Linux

1. Add the following lines to the `/etc/hotplug/usb.usermap` file.

```
#
# Altera USB-Blaster
#
usbblaster 0x03 0x09fb 0x6001 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
usbblaster 0x03 0x09fb 0x6002 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
usbblaster 0x03 0x09fb 0x6003 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0
```

2. Create a file named `/etc/hotplug/usb/usbblaster` and add the following lines to it.

```
#!/bin/sh

# USB-Blaster hotplug script

# Allow any user to access the cable
chmod 666 $DEVICE
```

3. Make the `/etc/hotplug/usb/usbblaster` file executable.

```
$ chmod a+rx /etc/hotplug/usb/usbblaster
```